Why Undergraduates Should Learn the Principles of Programming Languages*

ACM SIGPLAN Education Board

Stephen N. Freund (Williams College), Kim Bruce, Chair (Pomona College), Kathi Fisler (WPI),
Dan Grossman (University of Washington), Matthew Hertz (Canisius College),
Doug Lea (SUNY Oswego), Gary T. Leavens (University of Central Florida),
Andrew Myers (Cornell University), Larry Snyder (University of Washington)

June 24, 2010

Abstract

Undergraduate students obtain important knowledge and skills by studying the pragmatics of programming in multiple languages and the principles underlying programming language design and implementation. These topics strengthen students' grasp of the power of computation, help students choose the most appropriate programming model and language for a given problem, and improve their design skills. Understanding programming languages thus helps students in ways vital to many career paths and interests.

This white paper is based on contributed articles, discussions, and presentations from the 2008 SIGPLAN Programming Language Curriculum Workshop [3, 4].

Programming languages are the medium through which we describe computations. More specifically, we use the model provided by a programming language to discuss concepts, formulate algorithms, and reason about problem solutions. Programming languages define models tailored to thinking about and solving problems in intended application areas. For example, the C language provides a model close to a computer's underlying hardware, a spreadsheet language (such as Excel with Visual Basic for Applications) provides a model of cells and constraints for solving financial problems, and so on.

The languages used in practice change continuously, as advances in our field and the broadening uses of technology change how we model and express computation. The rise of the Internet and the web, for example, fundamentally transformed the way many types of systems are designed, implemented, and deployed. The rapid adoption of multicore and distributed platforms is again transforming how we program.

*This material is based upon work supported by the National Science Foundation under Grant No. CCF-0825525. Stephen Freund was also supported, in part, by Grant No. CCF-0644130. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

At its core, the study of programming languages examines the principles and limitations of computing (or programming) models, the effective design and use of systems or languages based on these models, and methods to compare their relative strengths and weaknesses in particular contexts. Undergraduate students benefit from studying this material in substantial ways, regardless of their future career paths and interests. It strengthens "a student's understanding of computation itself, its power and its limitations" [26]. Moreover, the knowledge and skills acquired enable students to critically compare and choose the most appropriate abstractions for describing particular programs, and to adopt and develop new models as they emerge. We elaborate on the most salient benefits of studying programming languages below.

1. *Learning Widely-Applicable Design and Implementation Techniques.*

   The key to building a complex system in any domain is identifying and implementing suitable abstractions for that area's core concepts. As new ideas, domain structures, or problems emerge, additional abstractions are formulated to accommodate them effectively. Often, these abstractions eventually rise to the level of being included in new programming languages for that domain, and in many ways the evolution of programming languages reflects our increased understanding of how to develop systems. However, the converse is also true: a thorough understanding of the principles and models underlying programming languages provides important insights into how to design and implement abstractions.

   For example, Dean and Ghemawat recognized that the best way to support Google's need to process huge data sets on large distributed clusters was to create MapReduce, a system "inspired by the *map* and *reduce* primitives present in LISP and many other functional languages" [7]. More specifically, they recognized that expressing algorithms as stateless, effect-free computations over streams of data would enable their platform to automatically parallelize and distribute work across many thousands of computers efficiently and reliably [9, 22]. MapReduce exemplifies how language principles transfer to many other situations ranging from large-scale system architectures to API designs to configuration mechanisms, regardless of the language ultimately chosen for implementation.

   Even on a smaller scale, stateless and effect-free computing models have an important role. Any concurrent system using shared mutable data structures may be prone to subtle race conditions and other synchronization defects. Expressing computation in an effect-free style (without changing the contents of existing variables or data structures) whenever possible, helps to avoid these potential problems and mitigate their impacts. Texts on concurrent programming [10] often urge programmers to use this style, which is more easily understood and adopted by those familiar with functional programming.

   In addition, web browsers, printer drivers, PDF renderers, scripted robot control systems, spreadsheets, video and audio players, and many other programs all share a common structure: they take complex input data and perform symbolic computation in a way similar to how compilers or interpreters act on the source code for a program [2]. Virtually every student will at some point work on a system like these, but implementing any form of symbolic computation can be quite subtle. Students

must understand the underlying theory and design principles (such as recursions that follow the source language grammar, name binding and resolution rules, etc.) to produce working and maintainable artifacts. Studying programming languages prepares students with that knowledge.

Even seemingly isolated topics in programming languages provide valuable insight and principles for systems design. Web browsers, cell phones, web application frameworks (such as the Facebook Platform), and an increasing number of other devices execute untrusted and potentially malicious code. Platforms for these devices, such as the Java Virtual Machine and Microsoft's .NET framework, often enforce access control via mechanisms based on how programming language implementations manage function calls and map variable names to storage locations [11, 16]. Similar principles underly designs of other systems involving named resources, including operating systems that must provide numerous mechanisms for naming files, processes, synchronization devices, other computers, etc., and distributed systems [21]. Also, abstract data types, objects, and modules underly the encapsulation and abstraction principles crucial to many software engineering methodologies [24].

A common issue in web applications further demonstrates the value of understanding programming language mechanics. Any such web application that must warn the user not to use the browser's "Back" button has fundamentally flawed interactions between its underlying control flow and its management of variables for session state. Programming language principles (continuations and coroutines for control flow, and stores and environments for variable management) shed light on the subtleties of these interactions in a way that both illustrates commonly-encountered problems and how to avoid them via alternative programming models [15].

2. *Creating New Domain Specific Languages or Virtual Machines.*

   Few students will ever design a general-purpose programming language during their careers. However, many will design domain-specific APIs, languages, or virtual machines. Such systems provide a computational model for thinking about data and algorithmic structures specific to problems in one particular context. For example, the MLFi language provides a model for describing the pricing and terms of financial contracts and language primitives for computing their valuations [8, 13]. By presenting a model centered around the specific topic of financial contracts, the designers created a more intuitive framework for solving problems related to financial contracts than a general-purpose language. MLFi has been used for this purpose quite successfully. Another example can be found in modeling the processes in systems biology [6].

   Computing is in fact replete with many heavily used domain-specific languages: Mathematica [23] and MATLAB [14] for manipulating mathematical formulas, Verilog and VHDL for describing computer hardware circuit designs, Cg [18] and others for writing rendering algorithms that run directly on graphics hardware, LaTeX for typesetting documents, etc. These languages all exploit properties of their intended domains to facilitate writing specific types of algorithms. For example, Cg provides direct language support for graphics concepts, such as vertices and

textures, as well as operations that can execute efficiently on the highly data-parallel processing units present in graphics cards.

Designers of domain-specific languages must always address the same basic issues: How expressive must the language be? What abstract model does it provide? How will it support user-defined naming and abstraction? How will programs communicate with the rest of the computing environment? Will any specific features interact in undesirable ways? As discussed above, similar issues affect the design of API layers in systems work, such as a virtual machine to encapsulate a hardware interface and enhance portability, or the virtual file system layer present in many operating systems.

Lack of knowledge of programming language fundamentals can lead to domain-specific languages that are difficult to understand and use or that require later repair. For example, dynamically scoped function texts (as opposed to lexically-scoped closures for functions) make higher-order abstractions unusable in many cases and lead to problems in type checking and optimization; this problem has had to be fixed in Lisp and Smalltalk. A solid programming languages foundation enables students to effectively recognize when designing a new language is appropriate and how to avoid these problems.

Understanding programming language principles and models often provides the insights leading to new innovations as well. Features of the financial contract language MLFi, for example, were inspired by Haskell and other languages that, while not currently pervasive, are often examined in programming languages courses. Type checking and event handling models provided key insights into the development of Hancock [5], a language used successfully by AT&T to write statistical analyses for identifying patterns in huge streams of call records, such as patterns indicating fraud. Sawzall [20], Dryad [30], and Pig [19] leverage the same language principles as MapReduce to model computations for distribution across large networks of computers at high levels of abstraction.

3. *Learning New Computational Models and Speeding Language Learning.*

   The programming languages and models commonly used in practice change constantly. Witness the growth in the use of object-oriented programming over the last 20 years for building large, extensible systems, in particular for building user interfaces. While these systems could be written in other types of languages, developers have recognized that the abstractions present in object-oriented languages facilitate their design and lead to more robust implementations.

   As another example, many widely-used languages now manage memory via garbage collection because advances in processor and memory performance, as well as improved collection techniques, have made garbage collection feasible for large systems. More recently, there has been a rapid rise in the use of light-weight scripting languages (such as Ruby and Javascript) to support the new application domain of web programming. We are in the middle of even greater changes rooted in the advent of multicore computer architectures.

   Given the pace of change, students will not use a single model or a single set of languages for their entire careers. They will frequently need to learn new languages

when they change jobs, start new projects, or begin working in new areas. The most challenging aspect of using a new language is understanding how to describe data and algorithms in a way that matches the strengths of the language's underlying model.

To illustrate this point, consider the fundamental difficulties of exploiting multiprocessor and multicore computer architectures, which are becoming the most promising way to achieve further computer performance improvements. Significant gains will be realized only if developers can successfully leverage the inherent concurrency in these architectures [27, 28]. This necessity has recently begun to drive both existing and new languages to include communication and concurrency features not common in most prior production languages: data-parallelism with roots in languages for functional programming and high-performance computing, transactional programming with roots in database languages, and process-based and actor-based programming previously seen primarily in niche parallel programming languages.

Those who just learn the syntax of languages embodying these concepts but proceed to program in a style suited for older models are doomed to be ineffective. Only a thorough understanding of the concurrency models provided by new languages will enable programmers to write robust, efficient programs. For example, X10 is a new object-oriented language for concurrent and distributed programming [25]. The language has "Java-like" syntax, but if programmers write "Java-like" programs without understanding the X10 computation model, they will fail to effectively use X10's asynchronous computation mechanism for improving performance, its notion of "places" for simplifying the design of distributed algorithms, and its notion of atomicity and thread communication primitives for avoiding deadlocks and other subtle, but common, errors.

The best preparation for quickly learning and effectively using new languages is understanding the fundamentals underlying all programming languages and to have some prior experience with a variety of computational models. This knowledge will endure longer than today's "hot" languages, which will undoubtedly become obsolete and give way to new languages in the future. In addition, this knowledge will enable students to quickly look beyond an unfamiliar language's surface-level details (such as syntax) and grasp the underlying computational model's design principles.

Programming languages often evolve to include successful features from other languages. Thus, a broad, solid foundation in this area also enables students to readily recognize and take advantage of changes in languages they currently use. For example, functional programming techniques offer clean, robust ways to express specific types of computation, such as manipulation of XML data from web pages, or exploration of algorithms in computer graphics. These techniques have become so widely adopted that many languages (including recent revisions to C# and Java) now directly support them via features such as anonymous functions, iterators, and generic polymorphism.

4. *Choosing the Right Language.*

The availability of so many languages and models means that students will need to make educated choices about which to use for specific tasks. Even individual systems are now rarely built entirely in one language. Instead, they are the composition of various components, each written in a language chosen for its strengths in that component's particular problem domain. For example, a web application may include database queries written in SQL, server application logic written in Java, data transformers written in XSLT, and client-side code written in JavaScript.

The choice of programming language can dramatically influence how one thinks about the design and structure of computation, and while it may be theoretically *possible* to solve every problem in any reasonable language, some problems inherently lend themselves to specific ways of thinking and programming. For example, Twitter recently switched parts of its server infrastructure from Ruby on Rails to Scala because Scala better matched their needs for long running threads, high performance under heavy loads, and more robust code via compile-time type checking [29, 1]. Scala also allows one to write parts of a system using functional programming techniques, which is attractive because many data transformations performed by a server like Twitter may be most easily written in a language expressing computation as composable functions applied to streams of data.

Other companies have enjoyed similar benefits from specific language choices. The Wall Street firm Jane Street Capital attributes a major part of their success to adopting the language OCaml for their on-line trading, research, and management systems [17]. That language's module system helped them to avoid error-prone code duplication practices endemic in previous systems built with that domain's more traditional languages (*e.g.*, C++, Java, and Excel with Visual Basic), and it led to code that was much more readable and intuitive to discuss with business people during their stringent code review procedures.

Paul Graham notes that his company's use of the Lisp language was instrumental in the success of their online store front application, which eventually became "Yahoo Store" [12]. That language enabled them to develop and deploy new functionality more rapidly than their several dozen competitors, who were primarily using C++ and CGI scripts. Elements of the Lisp model absent in those other languages, such as meta-programming primitives enabling programs to create, modify, and execute new pieces of code, also made implementing complex features much easier. [1]

On the other hand, choosing an ill-suited model can make devising and implementing a program far more difficult, complex, and error-prone. To avoid these pitfalls, students must have the intellectual framework and skills to critically relate models to languages and to determine which choices can best solve the problem at hand.

Summary. Programming languages embody many concepts central to all of computer science, including abstraction, generalization and automation, computability, and resource management. Studying programming languages enables one to examine these

[1] In essence, these features were used to define a domain-specific programming language for describing online stores.

topics in a precisely defined, accessible domain, and the lessons learned from programming languages thus provide immediate insight into all aspects of our discipline.

In the past, some programming languages courses have focused on surveys of languages in isolation, which can lead to these connections being less well-understood and appreciated. However, courses that follow a more principled approach to the wealth concepts in programming languages, and also present them in a broader context, provide all the benefits we have described. We encourage those interested in more detail to read a short companion article describing the recommended content of an undergraduate programming languages course based on this approach [3].

2008 SIGPLAN Programming Language Curriculum Workshop Contributors. *Steering Committee Co-Chairs*: Kathleen Fisher (AT&T Research). Chandra Krintz (UC Santa Barbara) *Steering Committee Members*: Eric Allen (Sun Microsystems), Ras Bodik (UC Berkeley), Kim Bruce (Pomona College), Matthias Felleisen (Northeastern Univ.), Stephen Freund (Williams College), Robert Harper (CMU), Michael Hind (IBM Research), Jim Larus (Microsoft Research), Doug Lea (SUNY Oswego), Greg Morrisett (Harvard Univ.), Lori Pollock (Univ. of Delaware), Stuart Reges (Univ. of Washington), Martin Rinard (MIT), Olin Shivers (Northeastern Univ.). *Participants*: Mark Bailey (Hamilton College), William Cook (UT Austin), Kathi Fisler (WPI), Daniel Friedman (Indiana University), John Hughes (Chalmers), Shriram Krishnamurthi (Brown), Gary T. Leavens (University of Central Florida), John Reynolds (CMU), Peter Sestoft (ITU), Lynn Andrea Stein (Olin College of Engineering), Mark Sheldon (Wellesley College), Larry Snyder (University of Washington), Franklyn Turbak (Wellesley College), Mitchell Wand (Northeastern University).

References

[1] April 2009. `http://lambda-the-ultimate.org/node/3261`.

[2] Eric Allen. Some things that computer science majors should know. *SIGPLAN Not.*, 43(11):32–35, 2008.

[3] Eric Allen, Mark W. Bailey, Rastislav Bodík, Kim B. Bruce, Kathleen Fisher, Stephen N. Freund, Robert Harper, Chandra Krintz, Shriram Krishnamurthi, James R. Larus, Doug Lea, Gary T. Leavens, Lori L. Pollock, Stuart Reges, Martin C. Rinard, Mark Sheldon, Franklyn A. Turbak, and Mitchell Wand. SIGPLAN programming language curriculum workshop: Discussion summaries and recommendations. *SIGPLAN Notices*, 43(11):6–29, November 2008.

[4] Mark Bailey, Kim Bruce, Kathleen Fisher, Robert Harper, and Stuart Reges. Report of the 2008 SIGPLAN Programming Languages Curriculum Workshop: Preliminary Report. In *Proceedings of the ACM Technical Symposium on Computer Science Education*, pages 132–133, 2009.

[5] Corinna Cortes, Kathleen Fisher, Daryl Pregibon, Anne Rogers, and Frederick Smith. Hancock: A language for analyzing transactional data streams. *ACM Trans. Program. Lang. Syst.*, 26(2):301–338, 2004.

[6] V. Danos, J. Feret, W. Fontana, R. Harmer, and J. Krivine. Rule-based modelling and model perturbation. *Transactions on Computational Systems Biology*, 11:116–137, 2009.

[7] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[8] Jean-Marc Eber. The financial crisis, a lack of contract specification tools: What can finance learn from programming language design? In *European Symposium on Programming*, pages 205–206, 2009.

[9] The Apache Software Foundation. Hadoop: Open source implementation of MapReduce. `http://hadoop.apache.org`, 2009.

[10] Brian Goetz, Tim Peierls, Joshua Block, Joesph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency in Practice*. Addison Wesley Professional, 2006.

[11] Li Gong, Gary Ellison, and Mary Dageforde. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation (2nd Edition)*. Prentice Hall, 2003.

[12] Paul Graham. *Hackers and Painters: Big Ideas from the Computer Age*. O'Reilly, 2003.

[13] Simon L. Peyton Jones, Jean-Marc Eber, and Julian Seward. Composing contracts: an adventure in financial engineering, functional pearl. In *International Conference on Functional Programming*, pages 280–292, 2000.

[14] The Mathworks. MATLAB. `http://www.mathworks.com/`, 2009.

[15] Jacob Matthews, Robert Bruce Findler, Paul Graunke, Shriram Krishnamurthi, and Matthias Felleisen. Automatically restructuring programs for the web. *Automated Software Engineering*, 11(4):337–364, 2004.

[16] Microsoft. Microsoft .NET framework. `http://www.microsoft.com/NET/`, 2009.

[17] Yaron Minsky and Stephen Weeks. Caml trading - experiences with functional programming on wall street. *J. Funct. Program.*, 18(4):553–564, 2008.

[18] NVIDIA. Cg — the language for high-performance realtime graphics. `http://developer.nvidia.com/page/cg_main.html`, 2009.

[19] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: a not-so-foreign language for data processing. In *SIGMOD International Conference on Management of Data*, pages 1099–1110, 2008.

[20] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 13(4):277–298, 2005.

[21] Rob Pike, David L. Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(2):221–254, 1995.

[22] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary R. Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *International Conference on High-Performance Computer Architecture*, pages 13–24, 2007.

[23] Wolfram Research. Mathematica. `http://www.wolfram.com/`, 2009.

[24] Barbara G. Ryder, Mary Lou Soffa, and Margaret Burnett. The impact of software engineering research on modern progamming languages. *ACM Trans. Softw. Eng. Methodol.*, 14(4):431–477, 2005.

[25] Vijay Saraswat and Nathaniel Nystrom. Report on the experimental language X10. `http://dist.codehaus.org/x10/`, 2009.

[26] Olin Shivers. Why teach programming languages. *SIGPLAN Notices*, 43(11):130–132, 2008.

[27] H. Sutter. The free lunch is over: a fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), 2005.

[28] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.

[29] Bill Venners. Twitter on Scala: A conversation with Steve Jenson, Alex Payne, and Robey Pointer. `http://www.artima.com/scalazine/articles/twitter_on_scala.html`, 2009.

[30] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 1–14, 2008.