Programming Languages as Part of Core Computer Science

Kim Bruce
Computer Science Department
Pomona College
Claremont, CA 91711

Stephen N. Freund
Computer Science Department
Williams College
Williamstown, MA 01267

Abstract

While the programming languages course played a key role in Curricula '68, '78, and '91, Curriculum 2001 replaced most of the content in programming languages with sections on learning to program. We argue that the need for a programming languages course has not diminished, but instead increased, especially as we move into an era of many-core computing.

Categories and Subject Descriptors K.3.2 [*Computers and Education*]: Computer and Information Science Education—Computer Science Education; D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Languages

Keywords Programming languages curriculum

1. PLs in ACM/IEEE CS Curricula

Curriculum 2001 [For01] departed from all previous ACM and IEEE CS model curricula by deemphasizing the role of programming languages, aside from its use in teaching programming. Below we have listed the Curriculum 2001 recommendations in programming languages:

PL. Programming Languages (21 core hours)
- PL1. Overview of programming languages (2)
- PL2. Virtual machines (1)
- PL3. Introduction to language translation (2)
- PL4. Declarations and types (3)
- PL5. Abstraction mechanisms (3)
- PL6. Object-oriented programming (10)
- PL7. Functional programming
- PL8. Language translation systems
- PL9. Type systems
- PL10. Programming language semantics
- PL11. Programming language design

Only PL1–PL6 are considered core topics in the curriculum. The others are recommendations for programs that wish to go beyond the core. Of the 21 hours spent on those core topics, only the five hours in PL1–PL3 were in the PL section of the 1998 draft report. PL4–PL6 were originally in the "Programming

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
2008 SIGPLAN Workshop on Programming Language Curriculum, May 29–30, 2008, Cambridge, Massachusetts, USA.
Copyright © 2008 ACM 0362-1340/2008/05...$5.00.

Fundamentals" section, but were moved to programming languages to appease the complaints from the programming languages community about this area being ignored. All of the sample curricula presented by the committee placed the material in PL4–PL6 in the introductory course sequence, rather than more advanced courses as envisaged in earlier curricula.

There were no representatives from the programming languages community on the committee, and no advisory group was originally formed for the area. Only after complaints about the earlier draft was a PL KFG (Knowledge Focus Group) formed. The recommendations of that group were subsequently mainly ignored by the full committee. (See [Bru00] for the PL KFG's recommendations.[1])

While there was never any substantive response to the PL KFG's recommendations, the impression was left that the study of programming languages was no longer important, and that it was more important to include material on more trendy topics in computer science.

In contrast, the Liberal Arts Computer Science consortium's 2007 model curriculum in Computer Science [Con07] has a Programming Languages course as part of the core curriculum as well as a segment on functional languages in another course.

In the remainder of this document, we address the importance of programming languages to an undergraduate education in Computer Science.

2. The Study of Programming Languages is Central

Today's recommended computer science curriculum exposes students to very few kinds of programming languages. Aside from the languages in the C/C++/Java/C# family, some students will see scripting languages like Python and Ruby, while even fewer will see a functional language like LISP, Scheme, ML, or Haskell.

However, exposure to these languages, and those representative of other programming paradigms, is central to a solid computer science education for several reasons:

- *Core CS Ideas.* The study of programming languages raises key issues that permeate computer science. Recurring themes include:
  1. time/space tradeoffs;
  2. the limits of computability;
  3. the flexibility obtained by putting off decisions versus the efficiency obtainable by making decisions early (late/early binding); and
  4. modularity and abstraction.

  The programming languages course is well-suited for pulling in topics from across the core of computer science and examining their connections in a rigorous way.
- *Algorithmic thinking.* It is imperative that students learn early on to be flexible in expressing their algorithmic thinking. Exposure to and practice using different programming paradigms is the best way to teach students this fundamental skill. Students must be able to adapt to languages that they will be using in the future, as well as adopt techniques originally developed for other paradigms.
- *Preparing for the future.* The computing platform and application domains are not static, and changes to them will impact the programming languages used in the future. The most used programming languages have changed regularly over the last several decades, and they will certainly change again in the coming years (due, in part, to the issues described below). C++ has been dominant, but it has lost ground to Java, just as Java will certainly be replaced by another language.

[1] An on-line version of the knowledge units is available at http://www.cs.pomona.edu/∼kim/Curric2001/PL2001.html

Students must be able to understand the impacts of new languages on how they approach computational problem solving. The programming languages course can and should provide the background for students to evaluate and adapt to future changes.

3. Visions of Programming Languages

For us, a programming languages course is as essential as a course in computer organization. A programming language creates a "virtual computer" with data structures and operations for creating algorithms. If a computer scientist were only to use a single language during his/her career, then perhaps there would be little reason for requiring a programming languages course. However, it is not atypical for computer scientists to use six or more quite different languages during their professional careers. Programming languages courses examine the dimensions in which these languages differ and provide unifying concepts (e.g., binding time) with which one can better understand the differences between languages.

Perhaps even more importantly, learning different kinds of programming languages requires different ways of thinking about algorithms. Algorithms are expressed quite differently, for example, in procedural, functional, object-oriented, and logic programming languages. Recognizing how these kinds of languages differ and are similar requires understanding the fundamental principles of programming languages. We refer the reader to the description of the proposed knowledge units in programming languages for Curricula 2001, referenced earlier, for examples of these principles.

In case there is any concern that the topics in a programming languages class are static and no longer of interest, we list below several recent trends that have substantially impacted programming languages. Our computing platforms and applications domains are not static. Other trends will arise that will require the development and adoption of new language technologies, and understanding programming languages principles is not decreasing in importance, but perhaps even more important now than ever.

3.1 Recent changes in programming languages.

Generics. Among the more recent changes in existing programming languages has been the addition of generics (type parameters) in languages such as C++, Java, and C#. Generics have been around at least since the language Clu [LSAS77], and they played an important role in Ada [US 80].

Generics were a late addition to C++ and functioned essentially via macro-expansion, leading to a long struggle for robust and efficient implementations. The Java addition incorporated a relatively new kind of bounded polymorphism, with a clever implementation [BOSW98] that enabled this feature to be compatible with the existing Java virtual machine language. C# added similar features, but with support in the intermediate language [KS01].

The "F-bounded" polymorphism mechanism incorporated by Java and C# was developed by language researchers [CCH$^+$89] in 1989 and was quickly modelled in an extension of the bounded polymorphic lambda calculus [CW85]. Because of the extensive work on the type theory of this theoretical foundational language, correct type-checking rules were quickly developed, providing a sound theoretical basis for the language and suggesting algorithmic implementations of the typing rules.

The implementations of this form of polymorphism in these three languages illustrate different important principles of programming languages. For example, the concepts of early and late binding are represented by the difference between the link-time type checking of the instantiation of generics in C++ and the compile-time checking possible in Java and C#. Careful analysis of the programming languages reveals why generics instantiation can be done at compile time in the latter two, while it must be delayed in C++.

The recent popularity of languages like Python also presents an excellent opportunity to emphasize the important differences between dynamic and static type checking (again illustrating the importance of binding time) and the trade-offs involving different levels of information hiding.

Concurrency. It is widely accepted that the rapid increases in processor speed due to Moore's law are at an end. Instead Moore's law will be preserved by increasing the number of processors on a chip. Given these

architectural advances, multithreading is likely to be the most promising way to achieve further performance improvements for many computer systems.

Quad-core and eight-core chips are already readily available, and if current trends continue, processors may have hundreds or even thousands of cores within ten years. However, building large, robust single-threaded programs is already quite difficult. The complexities of programming multiple processes on many-core computers reliably and efficiently are well beyond the capabilities of a large proportion of today's computer scientists.

Similarly, distributed computing and the provision of software services raise programming issues that are not well supported by current languages. These changes to the computational resources available to programmers require new mechanisms for managing complexity.

It is hard to avoid writing concurrent code in many modern applications. For example, event-driven programming using the Subject-Observer pattern (as exemplified in the Java event model) requires the provision of separate threads to respond to events while a computation is progressing.

If we are to provide mechanisms to support concurrent programming then we will be forced to make many decisions in the choice of an appropriate programming language. Traditionally these have included the choice of shared-memory versus message-passing, how to control memory access in shared-memory, and whether to make message passing synchronous or asynchronous.

Ad hoc attempts to add concurrency to existing languages have generally resulted in many problems. For example early implementations of concurrency in Java were problematic. See [MPA05], for example, for problems with the Java memory model. Moreover, the rather low-level support for concurrency presently found in most programming languages makes it too hard for most programmers to write correct and efficient programs running on a large number of processors.

Many researchers are examining different approaches to concurrent programming that should suggest new robust and high-level programming language constructs. In the last few years, for example, we have seen proposed languages Chapel [CCZ07], Fortress [ACH+08], and X10 [CGS+05], each of which exhibits interestingly different approaches to concurrency. The concepts of software transactional memory [ATKS07, Jon07] (taken from database transactions) and chords [BCF04], for example, represent even more intriguing new ideas for controlling concurrency.

It is hard to anticipate what language features will prove to be the most useful in writing programs for 1000-core chips. However, the analysis tools gained in a programming languages course will help us to understand the differences and evaluate the trade-offs in the various proposals that will arise.

Security. A final element of systems design that has recently become much more important in the context of programming languages is security. The wide-spread dissemination of code on the web (and from possibly untrusted sources), as well as the continually-growing list of software services in our daily life, make the robustness and security of programs increasingly important.

Various languages have begun to include mechanisms to help ensure security properties. Java, for example, includes a stack inspection mechanism (as described in [WF98]) to determine which code has access to resources. Other languages and prototypes now include mechanisms for tracking information flow to prevent security violations [Mye99, WCO00]. In the future, it is likely that programming languages will include even more features to manage trust, secrecy, and integrity.

4. Summary

The programming language course has played, and will continue to play, an important role in the education of undergraduates. As such, it should regain its status as an integral part of a computer science education in curricular standards. Not only are the topics of central interest to computer scientists today, but they also help prepare students for the inevitable changes that they will encounter over their careers. Moreover, the recurring themes in a good programming languages course reinforce those learned in other courses,

making it a good location to pull together many of the intellectual threads introduced in a computer science education.

References

[ACH+08] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Jr. Guy L. Steele, and Sam Tobin-Hochstadt. *The Fortress Language Specification 1.0*. Sun Microsystems, 2008.

[ATKS07] Ali-Reza Adl-Tabatabai, Christos Kozyrakis, and Bratin Saha. Unlocking concurrency. *Queue*, 4(10):24–33, 2007.

[BCF04] Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for C#. *ACM Trans. Program. Lang. Syst.*, 26(5):769–804, 2004.

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Vancouver, October 1998. ACM.

[Bru00] Kim B. Bruce. Curriculum 2001 draft found lacking in programming languages. *SIGPLAN Not.*, 35(4):26–28, 2000.

[CCH+89] P. Canning, W. Cook, W. Hill, J. Mitchell, and W. Olthoff. F-bounded quantification for object-oriented programming. In *Functional Prog. and Computer Architecture*, pages 273–280, 1989.

[CCZ07] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21:291–312, 2007.

[CGS+05] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 519–538, New York, NY, USA, 2005. ACM.

[Con07] Liberal Arts Computer Science Consortium. A 2007 model curriculum for a liberal arts degree in computer science. *J. Educ. Resour. Comput.*, 7(2):2, 2007.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[For01] CC2001 Joint Task Force. *Computing Curricula 2001: Computer Science*. December 15, 2001. Available online at http://www.acm.org/sigcse/cc2001/.

[Jon07] Simon Peyton Jones. Beautiful concurrency. In Greg Wilson, editor, *Beautiful code*. O'Reilly, 2007.

[KS01] Andrew Kennedy and Don Syme. Design and implementation of generics for the .NET Common Language Runtime. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 1–12, 2001.

[LSAS77] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction mechanisms in CLU. *Comm. ACM*, 20:564–576, 1977.

[MPA05] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 378–391, New York, NY, USA, 2005. ACM.

[Mye99] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 228–241, 1999.

[US 80] US Dept. of Defense. *Reference Manual for the Ada Programming Language*. GPO 008-000-00354-8, 1980.

[WCO00] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl*. O'Reilly & Associates, Inc., 2000.

[WF98] Dan S. Wallach and Edward W. Felten. Understanding Java stack inspection. In *Proceedings of IEEE Symposium on Security and Privacy*, 1998.