2008 SIGPLAN Programming Language Curriculum Workshop Report

Kathleen Fisher (co-chair), Chandra Krintz (co-chair), Eric Allen (Sun Microsystems), Mark Bailey (Hamilton College), Ras Bodik (UC Berkeley), Kim Bruce (Pomona College), William Cook (UT Austin), Matthias Felleisen (Northeastern University), Kathi Fisler (WPI), Stephen Freund (Williams College), Daniel Friedman (Indiana University), Robert Harper (CMU), Michael Hind (IBM Research), John Hughes (Chalmers), Shriram Krishnamurthi (Brown), Jim Larus (Microsoft Research), Doug Lea (SUNY Oswego), Gary Leavens (University of Central Florida), Greg Morrisett (Harvard), Lori Pollock (University of Delaware), Stuart Reges (University of Washington), John Reynolds (CMU), Martin Rinard (MIT), Peter Sestoft (ITU), Mark Sheldon (Wellesley College), Olin Shivers (Northeastern University), Larry Snyder (University of Washington), Lynn Stein (Olin College), Franklyn Turbak (Wellesley College), and Mitchell Wand (Northeastern University)

Abstract

In the past decade, the field of computer science has experienced explosive growth. Unfortunately, undergraduate curriculum has not tracked these advances, and it therefore urgently requires revision. In response to this need, SIGPLAN and NSF sponsored a workshop on May 29-30, 2008 at Harvard University to address why teaching undergraduates about programming languages is important, what material should be in the curriculum, and how that material should be taught. This report describes the motivation for the workshop, its organization, and the recommendations of the participants.

Overview

This report summarizes the discussions and conclusions of a two-day workshop on undergraduate programming language curriculum held in May 2008 under the sponsorship of NSF and SIGPLAN. The report is divided into four parts. The first summarizes the organization of the workshop. The second describes the discussions that took place at the workshop, which included 1) why undergraduate computer science students should learn about programming languages, 2) what they should learn about them, and 3) how they might be taught that material. The second part also describes various recommendations made by the workshop participants, including 1) the ACM/IEEE curriculum should be modified to require ten classroom hours of instruction on functional programming and 2) SIGPLAN should constitute an Education Board to continue the work on improving programming language instruction. The third part includes all of the white papers contributed by workshop participants on the question of undergraduate programming language instruction. The final part offers a brief summary and calls on members of the community to contribute to the on-going effort to modernize undergraduate programming language instruction.

Contents

Part I
Workshop Organization

1. Motivation

Programming languages play a critical role in computer science in that they provide a flexible and robust means by which human beings interact with and control computer systems. Programming language design and implementation has advanced significantly in the recent past in response to the increasing pervasiveness and diversity of computer science and technology. Unfortunately, computer science undergraduate curriculum in the United States has not kept pace and does not adequately reflect these recent advances. To prepare a globally competitive workforce and to produce the next generation of computer science leaders, the programming language community must identify, critically evaluate, and promote the necessary transformational curricular changes.

The goal of the Programming Language Workshop was to address this challenge by bringing together leaders in the programming languages community with expertise in research, teaching, and industrial use in order to discuss the role of programming language design, implementation, and application in modern undergraduate computer science education. In particular, the workshop provided a forum to

- Evaluate recent changes and likely trends in computing technology and their impact on programming language design, implementation, and application (and vice versa),
- Discuss the implications of these changes on programming language curricula, and
- Explore strategies for designing new curricula.

For the first of these tasks, participants explored relevant trends, including the looming ubiquity of multiprocessing systems; the proliferation of domain-specific languages; the increasing diversity of programming languages, infrastructures, and support tools in active use; the growing heterogeneity of device architectures such as high-performance computing systems, desktops, game consoles, mobile phones, and hand-held devices; and the increasing complexity of operating systems, runtime systems, and applications. For the second task, participants debated the implications of these trends, including why, what, and how we should teach undergraduates about programming languages. Finally, for the third task, participants explored various tactics for designing new programming language curricula, striving to incorporate the insights from the previous discussions while satisfying the constraints imposed by the diversity of existing undergraduate programs and by the needs of other areas of computer science.

2. Participants

The workshop, which was sponsored by SIGPLAN and the NSF, took place May 29-30, 2008 at Harvard University. Kathleen Fisher (AT&T Labs Research) and Chandra Krintz (UC Santa Barbara) co-chaired the meeting, with the help of a steering committee comprised of the following fourteen individuals, selected by the chairs to represent a cross-section of the programming language community:

- Eric Allen (Sun Microsystems)
- Ras Bodik (UC Berkeley)
- Kim Bruce (Pomona College)
- Matthias Felleisen (Northeastern University)
- Stephen Freund (Williams College)
- Robert Harper (CMU)
- Michael Hind (IBM Research)

- Jim Larus (Microsoft Research)
- Doug Lea (SUNY Oswego)
- Greg Morrisett (Harvard)
- Lori Pollock (University of Delaware)
- Stuart Reges (University of Washington)
- Martin Rinard (MIT)
- Olin Shivers (Northeastern University)

All steering committee members attended the workshop. In addition, the steering committee selected another thirteen participants on the basis of three-page white papers submitted in response to a public call for contributions, available from the workshop web site (`http://www.sigplan.org/pl-workshop`). We received 34 submissions, each of which was evaluated by at least four steering committee members. On the basis of these reviews, the steering committee decided to invite thirteen authors corresponding to eleven white papers (two accepted submissions had two authors). Only one of these invitees was unable to attend the workshop. The authors of accepted white papers who attended the workshop were:

- Mark Bailey (Hamilton College)
- William Cook (UT Austin)
- Kathi Fisler (WPI)
- Daniel Friedman (Indiana University)
- John Hughes (Chalmers)
- Shriram Krishnamurthi (Brown)
- Gary Leavens (University of Central Florida)
- John Reynolds (CMU)
- Peter Sestoft (ITU)
- Mark Sheldon (Wellesley College)
- Franklyn Turbak (Wellesley College)
- Mitchell Wand (Northeastern University)

In addition, Lynn Stein (Olin College) attended the workshop at the invitation of the local arrangements chair (Greg Morrisett). Finally, Sol Greenspan (NSF) attended as a representative of NSF and Larry Snyder (University of Washington) as a representative of the ACM Education Board. In total, there were thirty-one workshop attendees.

3. Mission Statement

In advance of the workshop, participants drafted and approved the following “Mission Statement” to ensure everyone agreed on the purpose of the workshop and to make the group more effective and focused during the relatively brief meeting.

3.1 Background

In the past decade, the field of computer science has grown explosively. Entirely new areas of investigation have emerged. Existing fields have experienced exciting advances. Changing circumstances have led to new challenges. As part of this explosive growth, the state of the art and the practice of programming languages have also changed dramatically, with the industry-wide adoption of high-level programming languages with managed runtime systems, the increasing importance of programming language support for effective parallel programming, the wide-spread use of scripting and domain-specific languages, important advances in type systems, *etc.*

Unfortunately, undergraduate curricula have not tracked recent advances and is therefore severely in need of revision. It is therefore crucial that we in the programming language community do our part and revisit the question of what we teach undergraduate computer science majors about the field of programming languages, why that material is important to a broad audience, and how we recommend it should be taught. Note that when we refer to programming languages, we mean the field of programming languages and not the mechanics of programming in a particular language.

3.2 Goal

The goal of the Programming Language Curriculum (PLC) Workshop is to facilitate the adoption of curricula that ensure every computer science undergraduate learns the programming language topics essential for a productive career as a computer scientist. We note that most such computer scientists are practitioners, not researchers. Consequently, it is particularly important that we prepare our students well for jobs in industry, giving them the skills necessary for success in the field throughout the entirety of their careers. At the same time, undergraduate programming language instruction must also appeal to students interested in pursuing academic studies and research in the field.

Our goal requires influencing the curricula of the vast diversity of colleges and universities that teach computer science. Such institutions have widely varying resources, class sizes, and expertise, and face competing instructional demands from other computer science sub-fields. To maximize our potential for success, the programming language community must reach consensus on what should be taught, why these topics are important to teach, and how such instruction can be carried out successfully. We must then document our findings and rationale carefully, disseminate the resulting materials widely, and encourage their use for curriculum development.

3.3 Strategy

The PLC workshop is the first step toward achieving this goal. In particular, as workshop participants, we will strive to produce the following items:

The Why A clear articulation of why every computer science undergraduate should develop a solid understanding of programming languages before graduation, directed to an audience of people *outside* of the field of programming languages.

The What A partially ordered list of programming language topics and experiences that every computer science undergraduate should know or have before graduation. Each item on the list should have a description and a rationale for why it is important.

The How A variety of recommended practices for teaching the identified topics. After the workshop, working groups will summarize the conclusions reached at the workshop in each of these three areas. These summaries, along with the white papers written by workshop participants, will be published as an issue of SIGPLAN Notices.

To reach community-wide consensus, workshop participants will also decide how to proceed after the workshop, including the identification of working groups to continue the tasks started during the workshop. These groups will be responsible for:

- Collecting feedback from the community on the workshop report and follow-on efforts.
- Revising the materials as appropriate.
- Maintaining a portion of the SIGPLAN web site devoted to this topic with the goal of providing resources to individuals working on computer science curricula to help such individuals determine what

programming language material to include, how to teach it, and how to persuade their colleagues of its import.

- Using the developed materials and the community consensus as input to the various official curriculum standardization efforts, including ACM and IEEE.
- Recruiting additional community members to the task, as appropriate.

3.4 Scope

The focus of the workshop is on developing a modern undergraduate programming language curriculum. Given the limited time-frame of the workshop, we will not be able to consider curricular questions related to compilers, software engineering, or other related fields except insofar as they directly relate to the design and motivation of programming language curricula. However, we believe that exploring curricula for these areas is a highly worthwhile activity and hope that appropriate individuals will pursue these questions in other forums.

3.5 A Note on How

We did not have sufficient time to explore fully the wide range of approaches people have used to teach programming languages. As part of the workshop, we have simply initiated discussion on this question and leave further work to the follow-on activities.

4. Workshop Organization

The workshop was a working meeting spread over the course of two days. The full agenda for the meeting is available on the web `http://www.sigplan.org/pl-workshop/schedule.html`. We began with presentations by Kim Bruce and Gary Leavens that reviewed the history of the ACM/IEEE Computer Science Curriculum and the Liberal Arts Computer Science Curriculum, as well as previous efforts in the programming language community to define a curriculum. We then collectively brain-stormed for an hour on the "What" and "Why" topics, using a silent brain-storming technique in which participants wrote their ideas on sticky notes and posted those notes on white boards for others to see, respond to, and group into categories. We then broke into four focus groups, two to address the "What" question and two to address the "Why" question. At the end of the first day, these focus groups summarized their discussions and views to the entire group.

On the morning of the second day, we continued the discussion from the previous evening, and then again broke into four focus groups, one to address the "What," one to address the "Why," and two to address the "How." After lunch, each of these focus groups presented their findings, and we had a group discussion on each of the three broad topics. We ended with a group discussion on follow-up activities. Slides and notes for each of these sessions are available on the workshop web site.

5. Outcomes

During the final session of the workshop, participants decided to take the following actions:

- Recommend specific short-term curriculum changes in the ACM/IEEE Curriculum.
- Encourage SIGPLAN to form a standing Education Board to focus on programming language curriculum issues.
- Publish a report (this document) describing the conclusions of the workshop participants.

Changes to Current ACM/IEEE Curriculum. One of the challenges in making curriculum changes is that it is a zero-sum game. Any additional units assigned to programming language topics must be taken away from some other area. Needless to say, taking units away from other areas is an extremely difficult and controversial task that requires input and support from a number of different computer science communities. With this constraint in mind, workshop participants developed a plan to replace ten classroom hours of programming language material from the current ACM/IEEE curriculum with ten hours on functional programming. Workshop participants felt that devoting a significant amount of classroom time to functional languages would help students develop a more sophisticated understanding of computational models specifically and programming language concepts in general. We provide a detailed report of this proposal and our rationalization for the change in Section 8.

Formation of a SIGPLAN Education Board. Workshop participants felt that the tasks of 1) developing a good programming language curriculum, 2) justifying that curriculum to people outside the community, and 3) showing how to design curriculum-compliant courses for a wide range of higher educational institutions require on-going attention. As a result, we recommended that SIGPLAN form a standing Education Board with the charter of working on precisely these tasks. Matthias Felleisen submitted a proposal to form such a board to SIGPLAN, and the SIGPLAN Executive Committee adopted the proposal in June of 2008. The SIGPLAN Executive Committee is currently working on forming the board. Among its other tasks, the Ed Board will maintain a portion of the SIGPLAN web site with continuously updated documents on 1) why we should teach programming languages to computer science undergraduates, 2) what material we think undergraduates should know about programming languages before graduation, and 3) how various courses teach that material.

The Report. Workshop participants also generated this report, which is divided into two parts. The first part summarizes the discussions at the workshop and the resulting recommendations. The second part is the collection of white papers written by each of the participants. Part 1 is further divided into sections:

- **Why.** This section, written by Martin Rinard, discusses why it is important for undergraduates to learn programming language material.
- **What.** This section describes the material workshop participants felt undergraduate computer science majors should learn about programming languages. A committee, chaired by Gary Leavens and Stephen Freund and including Kim Bruce, Robert Harper, and James Larus, contributed this section.
- **How: Modify ACM Curriculum.** This section, written by Stuart Reges, Shriram Krishnamurthi, Kathleen Fisher, and Chandra Krintz, describes the recommended changes to the existing ACM/IEEE curriculum.
- **How: No Programming Language Course.** This section, written by a committee chaired by Mark Bailey, explores how the necessary programming language topics might be covered in a curriculum with no programming language course.
- **How: The Single PL Course.** This brief section, written by Shriram Krishnamurthi, Robert Harper, Matthias Felleisen, and Greg Morrisett, outlines the structure of a course devoted to teaching programming language material.

Each of these sections reflects the consensus of the workshop participants.

This report is intended to provide to the community a timely summary of the discussions and recommendations of the workshop participants. It is not intended to be the final recommendation on the topic of undergraduate programming language curriculum. In particular, workshop participants strongly felt that we should produce several more documents:

- **Why: For non-programming language people.** There is a concern among workshop participants that the current "Why" document targets an audience that already supports teaching programming languages. We would like to produce another "Why" document addresssed to people outside of the programming language community, people who are skeptical about the utility of programming language instruction.
- **How: A Programming Language Course.** The existing "How" documents do not describe how the topics identified in the "What" section should be covered in a programming language course, which is particularly unfortunate because the committee strongly felt that such a course was by far the best approach.

Working on these tasks will be one of the jobs of the SIGPLAN Education Board. In addition, the Ed Board will actively and continously seek input and contributions from the community to improve and evolve this collection of documents and the suggestions contained within.

Part II
Discussion Summaries and Recommendations

6. Why Teach Programming Languages

Lead Author: Martin Rinard

A college education has two goals. First, to produce intellectually mature, sophisticated leaders who can think deeply and productively in a range of fields and contexts. Second, to provide students with skills that they can apply successfully throughout a long career in their chosen profession or professions. Programming language concepts and ways of thinking can be a critically important part of the successful education of virtually any college student, regardless of discipline. For computer scientists these concepts and ways of thinking are indispensible.

6.1 Programming Languages and the Practicing Computer Scientist

Each programming language embodies a model of computation — a perspective one uses to perceive, formulate, and solve problems that arise in the domain at hand. Each model of computation promotes a set of concepts and corresponding constructs for understanding and structuring computations. An understanding of these basic concepts is necessary for a practicing computer scientist to function effectively.

6.1.1 Learning New Languages

Over the course of a career, a successful computer scientist must reasonably expect to learn new programming languages — many of the mainstream languages in current use did not exist even two decades ago, and new languages are continually under development. Moreover, new mainstream languages often incorporate concepts from experimental languages once those concepts have proven their value in prior development efforts. Java, for example, incorporates generic types with sophisticated subtyping rules, iterators, anonymous inner classes, and reflection. The concepts behind these constructs were originally developed in various experimental programming language design projects before they appeared in any mainstream language. Yet today even a young computer scientist educated some ten years ago (before Java became widely adopted in undergraduate computer science curricula) must understand and use these concepts. A working knowledge of a range of programming language concepts (including concepts that have yet to make their way into existing mainstream languages) provides the intellectual foundation required to acquire and use new languages quickly and effectively.

6.1.2 Designing Systems

A software designer formulates the basic problem the software system must solve, decomposes the problem into tasks, then designs a solution for each task, all while ensuring that the tasks can successfully combine their solutions to make the system operate satisfactorily. A designer's knowledge of models of computation (as realized in the many available programming languages), in combination with the ability to match each task with an appropriate model of computation, is critical to his or her ability to deliver a competitive working design that can serve as the basis for a successful system development. Indeed, the need to understand the capabilities of different models of computation is present in virtually every aspect of this process:

- **Problem Formulation:** In general, the problem formulation depends on the tools available to solve the problem. So of course knowledge of the strengths and weaknesses of the different available languages affects the ambition and form of the problem that the designer aspires to solve. This knowledge is therefore a basic prerequisite for optimal problem formulation and system design.
- **Language Selection:** Developers have a choice of programming languages. An understanding of the characteristics of each language is essential when selecting a language or languages in which to develop the system. Potential alternatives include using a single language for the entire system as opposed different languages for different components, using a mainstream general-purpose language such as Java or C++ as opposed to less popular languages such as Smalltalk or ML, and whether or not to use scripting languages such as Perl or Ruby for part or even all of the system. Because language choice can have a significant impact on developer productivity, the ability to match languages effectively with task characteristics is an important component of the designer's overall skill set.
- **Task Decomposition:** In a multilingual development environment, it is often beneficial for all of the activities in a single task to match well with the language chosen for that task's implementation. Knowledge of the capabilities of each language is therefore necessary to obtain a productive decomposition of the problem into tasks.
- **Implementation Strategies:** The field of programming languages has also produced several specific approaches that can help designers formulate problems in a more general way to obtain more flexible, resilient, and successful designs. For example, a designer may choose to include a configuration language that system deployers or administrators can use to customize the operation of the system for their own particular needs or even to extend the system to support additional functionality. The resulting system design (a core execution engine programmed to support the desired behavior) has been used successfully in a wide variety of contexts, from low-level systems software to interactive multiplayer games. Because much of the functionality is expressed in a language designed specifically for that purpose, this approach can substantially reduce both the development time and the number of errors in the final system or systems. If the configuration language is general enough, it may support the very efficient construction of multiple distinct systems. And because the configuration language makes it possible to quickly and easily change the system behavior, it is possible to respond rapidly to user feedback or extend the system to include missing functionality.

6.1.3 Designing New Languages

Many new languages are designed to meet the needs of a certain domain. For example, Perl was originally designed for text processing, while JavaScript was designed to support the development of client-side Internet applications. As in the past, many new languages will be designed by domain experts (who understand the need for the language and the computations that it must support) rather than programming language specialists. But in addition to domain expertise, a good working knowledge of programming language concepts and the corresponding ability to effectively select and incorporate appropriate concepts into the language design is necessary to obtain a strong language that can successfully support the full range of (often unanticipated) uses.

General programming language design principles that may be useful in this context include constructs (such as classes, continuations, constraints, and higher-order functions) that support the decomposition of programs into appropriately modularized components, specific language restrictions that eliminate broad classes of errors, language design choices that enable static analyses to increase performance or find programming errors, and constructs such as regular expressions, grammars, and matrix operations that support the concise expression of complex computations.

Of course, very few computer scientists will ever design a complete programming language. But many will engage in design activities that benefit from programming language design concepts. Configuration files and procedure call interfaces to shared components, for example, are often essentially embryonic languages. The same design principles that apply to full-blown programming languages can often be productively applied to guide the successful development of these kinds of embryonic languages.

Small domain-specific languages are the next logical step beyond configuration files and procedure call interfaces. Developers who are able to design and implement such languages can substantially enhance the productivity of an entire development team. By minimizing the distance between the concepts of the application domain and the concepts of the implementation language, domain-specific languages provide the following benefits:

- **Rapid Development:** Domain-specific languages eliminate much of the need to manually translate domain concepts into language concepts. The elimination of this translation can significantly reduce both the size and the development time of the system.
- **Fewer Errors:** By matching the concepts of the domain directly in the programming language, domain-specific languages make it easier for the developer to think as they design and implement. A well-designed domain-specific language also limits the expressive power of the language to eliminate large classes of errors. The overall result is a system with fewer errors.
- **Easier Maintenance:** All of the advantages that domain specific languages deliver during development carry over into the maintenance phase. The improvements include fewer errors to fix and easier incorporation of new functionality.

Note that even if the developer is not directly involved in the development of the language or its implementation, the intellectual depth that comes from understanding how to effectively treat programs as data pays off in a more productive understanding and ability to effectively use those parts of the system that do. Knowledge of programming language concepts and implementation techniques is required to obtain the substantial productivity and reliability benefits that the use of small domain-specific languages can deliver.

6.1.4 Multilingual Development Environments

Different models of computation are appropriate for different problems. Indeed, a developer using a language that is well-suited to the problem at hand can be dramatically more productive than a developer using a less appropriate language. Organizations have recognized this reality by deploying multilingual development environments, with different languages used as appropriate for different tasks. Consider, for example, a standard database-backed web service. A standard approach is to use a combination of JavaScript for client-side development, Java for the server, and an SQL interface to a database written in C. To function effectively in such an environment, developers must understand the basic concepts of each language, how those concepts interact with the requirements of different subcomputations, and how to best match languages with specific tasks in the context of the overall system structure. All of these activities require a good working knowledge of concepts from a variety of programming languages.

6.1.5 Monolingual Development Environments

Despite the advantages of multilingual environments, there are countervailing reasons that can motivate organizations to use monolingual environments. Some of these advantages include more straightforward staffing (including the ability to move developers more freely within the project), a simpler development environment with fewer development tools, and (in some cases) easier integration of multiple components.

But even within a monolingual environment, the ability to appropriately select and employ concepts from different programming languages can substantially increase developer productivity and effectiveness. In a monolingual evironment there are always mismatches between the language's model of computation and the tasks that the system must perform. These mismatches can often be ameliorated by importing structuring concepts from other, more appropriate languages, then using these concepts to solve the problem within the standard development language. Specific instances in which this approach has delivered important improvements include the standardized use of structures and function pointer tables to obtain many of the benefits of object-oriented languages in standard procedural languages and the combination of objects and dynamic dispatch to simulate continuations in standard object-oriented languages. In both of these cases the importation of concepts from other languages enabled developers to deliver solutions superior to those that the model of computation of the original language was designed to promote.

6.1.6 Managing Development Activities

An effective manager knows what is and is not possible to deliver with the languages at the team's disposal. An effective manager can also recognize when the current language has become inadequate for a task at hand. Both of these activities require knowledge of a range of programming language concepts.

6.1.7 General Concepts, not Surface Details

There are obviously many different ways to teach these concepts. Nevertheless, it is important for the educational experience to emphasize a deep understanding of the general concepts behind different models of computation rather than the surface details of specific programming languages. Indeed, it is even possible to teach the full range of such concepts effectively while operating within a single base language (potentially augmented with small special-purpose languages designed for specific tasks that arise during the course of the instruction) if the language is suitably general and flexible. The goal is to provide students with an understanding of fundamental concepts that they can apply fluidly in many different contexts. Because some of the most important uses will involve applying concepts originally developed in one language to systems built in other languages (including languages that did not exist at the time of instruction), the surface details of specific languages are largely irrelevant.

6.2 Programming Languages as Part of a Liberal Education

Computers present the human race with an opportunity that is unique in the history of the species — the opportunity to harness vast amounts of computational power, but only if every aspect of the computation is specified explicitly and completely. Because of the need to make everything explicit and precise, several concepts that arise in almost all human endeavors (specifically abstraction, automation and generality, and the benefits of multiple perspectives) are present with unique precision, sophistication, clarity, and therefore accessibility. The availability of immediate, automated feedback through appropriate interaction with a computer enhances the accessibility of the concepts and promotes a fully rigorous approach that makes it difficult for a student to slide by with only a partial understanding.

Programming languages provide the single most sophisticated and intellectually challenging medium for interacting with a computer. Faced with the challenge of communicating with an entity starkly different from any our species have ever previously encountered, researchers have developed many different models of computation, each with its own perspective on how to approach and solve problems, and each with its own strengths and weaknesses in different contexts. Studying the basic concepts behind different models of computation can provide insight into not just the field of computer science, but in virtually every modern field of human endeavor. It can therefore increase students' intellectual depth and help them to think productively in many different contexts. We focus on several specific concepts that are particularly prominent in the field of programming languages.

6.2.1 Abstraction

Abstraction is the principled discarding of information to promote the clear exposure of certain relevant aspects of a phenomenon or entity. The strength of abstraction is its ability to make important aspects clearly visible. Its weakness is that it may hide other important aspects and thereby promote damagingly narrow reasoning. In the worst case a counterproductive abstraction can even lead to reasoning with a seriously skewed version of reality.

Programming languages, in general, provide two distinct kinds of abstraction. First, they hide many of the details regarding the execution of the computation on the machine at hand. These details are provided by the language implementation (typically a compiler or interpreter). Second, many languages contain metalanguages that developers use to describe different aspects of the computation. While the most prominent metalanguage is the type system present in all mainstream programming languages, partial specifications and the information extracted from program analysis systems are also usually expressed in a metalanguage.

These two kinds of abstractions precisely and clearly show how abstractions are useful for certain kinds of reasoning. They also show how abstraction can eliminate information required to reason successfully about certain aspects of the system. For many students the precision of these examples makes the concept of abstraction clearly accessible. These students can then build on the insight they gain from these examples to to more clearly perceive and understand less precise and less apparent abstractions in other contexts.

6.2.2 Automation and Generality

An important area of the field of programming languages focuses on the automated translation of computations from one model of computation to another, typically in the context of compilation, but more generally in translations between different languages. Successful automation requires discovering and considering every possible case up front before the automated activity takes place. Automating translation therefore provides an ideal field for learning how to think ahead and plan for contingencies in a general way. Because the starting and ending points are precisely defined, it is possible to precisely determine if an envisioned solution fails to consider a specific possibility.

Developing a successful automation in a challenging domain requires students to think deeply to develop insight into the nature of the phenomenon they are attempting to automate. The precision and clarity of automation tasks in programming languages can help make the concept clear to students. The experience of developing the comprehensive general understanding required for automation develops thinking skills and approaches that students can use in a wide variety of fields and situations. Finally, experience with automation helps students develop an understanding of both the capabilities and the limits of formalization and automation. In a society increasingly subject to pervasive computerized automation, a better understanding of this subject will help future leaders make wise decisions about what to automate, what not to automate, and, when automation is appropriate, how to automate. In particular, it can help students who focus on less technical disciplines understand the overwhelming power of a formal and precise approach when it is appropriate.

6.2.3 Multiple Perspectives

In general, there are many different perspectives one can use when considering a problem or situation. But most people adopt their perspectives unconsciously, to the point that much of the time they are not explicitly aware that they have a perspective at all. Knowing that alternate and potentially different or better perspectives are always available is a key stage in a student’s intellectual development. It helps people better understand the limitations of their reasoning, more readily consider other perspectives, and integrate a multiple perspective approach explicitly into the way they approach the world.

Working with models of computation can be a productive way to develop an understanding of the value of multiple perspectives. Presenting a student with concrete problems whose differing suitability for different models of computation is manifest can help the student understand that 1) different perspectives exist and can lead to dramatically different experiences and results, 2) no one perspective is optimal for all situations, and 3) explicitly incorporating multiple perspectives can provide otherwise unavailable insight into the situation. Once again, the precision and immediate feedback characteristic of the use of programming languages as an instructional playing field can make the concepts available to students.

6.3 Why Report Summary

Within the past several decades the field of programming languages has developed a set of basic concepts that are essential for the long-term success of any practicing computer scientist. A thorough and precise treatment of these basic concepts should therefore be part of the education of any computer scientist.

More generally, the field provides a uniquely favorable context for teaching several important concepts that are crucial to the preparation of all students, and not just computer scientists. Educational institutions should therefore encourage all students to participate in an appropriate educational experience that teaches these concepts.

7. What a Programming Languages Curriculum Should Include

Lead Authors: Kim Bruce, Stephen Freund, Robert Harper, Jim Larus, and Gary Leavens

During the workshop two groups discussed ideas about what college undergraduates majoring in computer science should learn about programming languages. These groups were charged with constructing [4]:

> "a partially ordered list of programming language topics and experiences that every computer science undergraduate should know or have before graduation. Each item on the list should have a description and a rationale for why it is important."

The workshop participants believe that there are core programming languages concepts that all undergraduates should learn, but recognize that programming languages may be an elective course for some students. Thus, both groups divided their lists into two sections, presenting:

- objectives for all students receiving a bachelor's degree in Computer Science or Computer Engineering, and
- strongly recommended objectives for students who study programming languages in more depth, by, for example, taking a (possibly elective) course in programming languages.

After presentation and discussion of these lists, the groups revised and discussed the list with all workshop participants. We next overview the objectives and outcomes that emerged with consensus from these discussions.

7.1 Objectives and Outcomes

The participants at the workshop strongly believe that *all* computer science undergraduates should learn the essential concepts and principles of programming languages regardless of whether they take a specific course on programming languages. Section 6 gives the overall justification for this requirement, while Sections 8, 9, and 10 present a variety of possible strategies for incorporating this material into an undergraduate curriculum, so these topics are not discussed in this section.

Section 7.1.1 states the overall objectives for learning about programming languages, as skills that could be measured after graduation (e.g., 2 or 5 years after graduation). Section 7.1.2 describes the desired outcomes, including specific ways in which undergraduates should be able to apply their knowledge of programming languages. Outcomes could be measured during or at the end of a course in which programming language material is presented.

Some parts of these objectives and outcomes are core and some are optional. We believe that the core objectives and outcomes are necessary for all computer science undergraduates. The optional material is strongly recommended, but not core. We realize that in many colleges and universities, some of the optional material may not be taught, due to various resource limitations.

7.1.1 Objectives

Programming languages are fundamental to program design, analysis, and implementation, and all computer science undergraduate students should acquire sufficient familiarity with programming language concepts to fulfill the core objectives outlined in this section. These objectives are common to virtually every computer-related career, be it software engineer, program manager, or researcher. For some objectives, we also present recommended extensions to the basic objective.

Every student graduating with a bachelor's degree in computer science or computer engineering should be able to:

1. **Quickly learn programming languages, and how to apply them to effectively solve programming problems.**

   Justification: Programming languages are the fundamental basis for programming, but trends in programming language usage change dramatically over time. Practitioners will not use the same programming language, or even the same programming model, for their entire professional career. Moreover, well-informed language choices can make an enormous difference in terms of programmer productivity and program quality. It is therefore crucial that students master the essential concepts of programming languages so that they may choose and use languages based on a deep understanding of the abstractions they express and their suitability for solving programming problems.

2. **Rigorously specify, analyze, and reason about the behavior of a software system using a formally defined model of the system's behavior.**

   Justification: Modern software development methods employ design, verification, and validation tools based on rigorously specified computational models, such as those defined by programming languages. Students must acquire a mastery of a useful set of such models and techniques for specifying and reasoning about systems.

   For example, students should be able to specify a language's behavior using a simple evaluation semantics, specify possible or guaranteed program behavior, and be able to specify system correctness properties using a type system or Hoare logic. In addition, students should be able to use models to rigorously reason about implementation correctness and performance (time and space behavior). This includes using analysis tools (such as type checkers), and sensibly interpreting their output.

   Specification, analysis, and reasoning skills are widely useful for designing and constructing any complex system, be it an API for printers, avionics control software, secure web service, etc.

   Recommended: In order to deepen student understanding of this material, we recommend that students be able to prove standard theorems about small computational models (i.e., small languages). For example, we recommend that students be able to prove type soundness for the simply typed lambda calculus or PCF [7]. Such skills are important for being able to design faithful abstractions that produce more tractable (smaller) models, or that express more interesting properties.

3. **Realize a precisely specified model by correctly implementing it as a program, set of program components, or a programming language.**

   Justification: Computational models are abstractions of programming languages, which are themselves very general software systems. Thus a precise specification of a computational model forms a specification of a system's intended behavior; implementations realize this behavior in a software system. Students must understand this relationship between specification and implementation and be able to build systems that correctly implement specifications. Doing this for computational models allows learning to focus on interesting capabilities (such as naming, control, parallelism, etc.) that are often ignored when specifications are not formulated as computational models. Thus the process of implementation of a computational model not only reinforces the connection between specification and implementation, but also demonstrates the power of formal specification in systems design and deepens one's understanding of the formal models being implemented. In addition, by working with precise abstractions, students may more easily move from one programming language for implementation to another, and devise and employ new languages for building systems.

Recommended: We recommend that students be able to implement an interpreter or compiler for a small programming language, given its precise formal definition. This is a fundamental experience that unifies all the ideas in programming languages and can have a profound impact on student understanding. It also conveys to students the power and potential of programming languages for enhancing the ease and reliability of programming.

7.1.2 Outcomes

In order to achieve the objectives listed above, students must acquire modeling and programming skills, as well as an understanding of computational models, programming language concepts, design tradeoffs, and implementation techniques. The outcomes described in this section are designed to fulfill these requirements.

We group the outcomes into mastery of concepts and skills in three areas, corresponding to the three core objectives:

1. **Ability to learn and effectively use new programming languages.**
2. **Ability to define and analyze rigorous computational models.**
3. **Ability to define and implement abstractions.**

For each area, we outline the core ideas and skills, describe why they are important, and list measurable outcomes for learning that topic. We also include some recommended topics, as well as topics that are important to our view of programming languages, but which are traditionally taught as parts of other areas in computer science.

Table 1 on the following page summarizes the core and recommended course hours for each outcome, as well as lecture hours for topics typically covered elsewhere in the curriculum. In this table we assume that other courses cover outcomes related to both complexity analysis and object-oriented programming. However, in some cases students may have other courses that, instead of teaching object-oriented programming, teach functional programming. In such cases we assume that 7 hours can be subtracted from the time allocated to core functional programming outcomes and used to cover those object-oriented outcomes that were assumed to be covered outside of programming languages.

1. Ability to learn and effectively use new programming languages. To be able to use a wide variety of new and existing languages, students must master the fundamental concepts of programming languages independently of their realization in a particular language. This enables students to recognize the core abstractions of programming languages within any given language, and to understand how to relate languages to one another. Students will also be able to express the same underlying concept in many different languages, and to understand the strengths and weaknesses of languages measured by their ability to express these concepts. Key topics in this area are outlined below.

- **Binding and Scope (1.5 core, 0 recommended)** describe how names are mapped to definitions and how name resolution works.

  Topics: (1.5 hours)

  1. Name declaration and binding.
  2. Free and bound occurrences of identifiers, renaming of bound identifiers, and capture-avoiding substitution.

  Justification: Despite the seemingly natural intuition behind how declarations associate names with definitions, there are in fact many ways to implement binding and scoping, some with adverse effects on system usability and understanding. For example, various Unix shells can exhibit quite unintuitive behavior due to their use of dynamic scoping for environment variables. Students must understand

Topic	Lecture Hours		
	Core	Recommended	Outside
1. Learn & effectively use new languages			
Binding and Scope	1.5	0	0
Control Structures	3	1.5	0
Data Structures	1.5	1	0
Modularity and Data Abstraction	3	0	0
Mutation and State	2	1	0
Deterministic Parallelism	1	0	0
Concurrency	1	0	0
Run-Time Implementation	3	0	0
2. Define & analyze computational models			
Concrete and Abstract Syntax	2	0	0
Dynamic Semantics	2	0	0
Static Semantics	4	3	0
Language Features as Types	0	3	0
Time and Space Complexity	0	0	2
3. Define & implement abstractions			
Functional Programming	10	0	0
Object-Oriented Programming	3	0	7
Parallel Programming	1.5	1	0
Concurrent Programming	1.5	1	0
Implementing Programming Languages	0	3	0
Total	**40**	**14.5**	**9**

Figure 1. An estimate of lecture hours for the core and recommended material, as well as lecture hours for topics typically covered elsewhere in the curriculum.

variable naming principles to recognize how existing languages and systems name values and to use these them correctly.

In addition, naming concepts are prevalent in many domains beyond programming languages. Operating systems provide numerous mechanisms for naming objects, such as files, processes, synchronization devices, other computers, etc. Namespaces are essential in managing and identifying resources in distributed systems, as illustrated by the work on Plan 9 [6] and later projects. The notion of a name representing an unguessable cryptographic secret (or nonce) is the foundation of the Dolev-Yao security model [3], and the Java security model's stack inspection mechanism uses dynamic scoping rules to enforce access control.

Skill details:

- Describe scope and binding design choices for some computation model.
- Recognize and appropriately use binding and scoping concepts in the design of a program managing named resources.

- **Control Structures (3 core, 1.5 recommended)** provide the mechanisms used to sequence program operations, thereby determining the order of a program's effects.

  Topics: (3 hours)

  1. Recursion and iteration, including the definition of and use of iterators.

2. Exception handling, including the difference between fatal and recoverable exceptions.
3. Dynamic dispatch, and the design and implementation of virtual methods in object-oriented languages.
4. Continuations.

Recommended Topics: (1.5 hours)

1. Fixed points.
2. Use of higher-order functions (functionals) as control abstractions.

Justification: As the basic building blocks of programs, the ability to select and apply appropriate control structures is necessary to fully master basic program design techniques. In addition, their understanding is essential for recognizing and properly implementing concepts central to the design of many systems. Dynamic dispatch and continuations illustrate this connection.

Dynamic dispatch serves not only as the foundation for virtual methods, but also as a useful programming idiom whenever an operation's behavior will vary with program state. For this reason it is the foundation of many design patterns [5].

Many software systems must capture the notion of the remainder of a computation, given its current state. For example, a process context in an operating system captures the computation still to be performed by a process. Similarly, a web service must somehow capture the remaining steps of a session with a client (which can be quite subtle, given the limitations of the stateless HTTP protocol). This notion of "remaining computation" has been well-studied in the context of continuations, a control structure appearing in a number of languages.

Skill details:

- Describe the results and effects of computations involving the above control structures.
- Know when and how to use control structures in the design, construction, and analysis of software.

• **Data Structures (1.5 core, 1 recommended)** reflect the basic types of data defined and manipulated in programs.

Topics: (1.5 hours)

1. Product and sum types, including records (structures) and variants (unions).
2. Recursive types, including inductive and coinductive types.

Recommended Topics: (1 hour)

1. Collection types, such as arrays and vectors.
2. Generated types, such as streams.

Justification: Elementary data structures are among the most basic abstractions provided by programming languages. While they appear in different forms, understanding their intended uses and how they are fundamentally related to each other is essential for learning new languages and critically comparing them. This includes recursive types for describing inductive structures, such as lists or trees, and coinductive structures, such as infinite streams.

Skill details:

- Model recursively structured data (such as web pages or expressions) using appropriate combinations of data structures.
- Write programs to correctly process and transform data of such types as those described above.

- **Modularity and Data Abstraction (3 core, 0 recommended)** features reduce the complexity of designing and writing software by allowing programs to be broken into separate pieces that can be implemented and verified independently.

  Topics: (3 hours)

  1. Separate compilation.
  2. Interfaces and implementations, representation independence, and compositionality.
  3. Abstraction mechanisms in programming languages, including procedures, classes, and modules.

  Justification: Abstraction is a fundamental concept in computer science, and complex software systems could not be created without extensive use of abstraction. Abstraction comes in many forms: programming languages themselves are abstractions of the underlying computing architectures, procedures are abstractions for sequences of actions, and module systems and classes in object-oriented languages are abstractions that combine data and operations together.

  Given the increasing importance of libraries and reusable components, students must have a deep understanding of modularity, and its various manifestations in programming languages, to be able to construct abstractions properly. Moreover, studying various forms of abstractions and their tradeoffs in the domain of programming languages will better prepare students to apply this concept in all domains.

  Skill details:

  - Decompose non-trivial programming problems into appropriate abstractions (and client code that uses these abstractions).
  - Use programming language features (procedures, classes, modules, etc.) to implement those abstractions.
  - Modify the implementation of an abstraction, preserving the correctness of client code.

  Note that higher-order functions can also be considered a modularity mechanism, but we have placed it under the topic of control structures above.

- **Mutation and State (2 core, 1 recommended)** describe changeable (time-varying) program storage.

  Topics: (2 hours)

  1. Mutable cells, references, and equality.
  2. Persistent and ephemeral data structures.

  Recommended Topics: (1 hour)

  1. Laziness and memoization.

  Justification: Students must understand the concept of mutable program state (such as mutable variables or structures, or mutable objects in object-oriented languages) in order to recognize the impacts of side effects, aliasing, and other subtle issues related to state mutation on the ability to implement robust systems. In addition, students should recognize the difference between persistent structures (in which previous versions are still accessible after modification) and ephemeral data structures (in which changes are directly visible via all references), and they should be able to effectively restrict the use of state in situations where persistence is useful or necessary.

  Skill details:

  - Design and implement persistent and ephemeral variants of a data structure.
  - Understand the costs and benefits of using mutable state, in terms of efficiency, complexity, and reasoning about programs.

- **Deterministic Parallelism (1 core, 0 recommended)** describes computational models that can perform multiple computations simultaneously with or without explicit indications (such as forking).

  Topics: (1 hour)

  1. Deterministic parallelism.
  2. Stream-based functional programming.

  Justification: In deterministic parallelism, race conditions do not exist, because all subcomputations are deterministic, i.e., they have a single value regardless of how other subcomputations are scheduled. This allows programmers to focus on performance and efficiency issues, without worrying about non-deterministic behavior (race conditions). It is important for students to understand the conditions under which such determinism holds, such as in stream-based functional programming or parallel lazy functional programming.

  With the rapid rise of multiprocessor and multicore systems, programming languages (including mainstream languages like Java, C#, and Python) are increasingly providing direct support for parallel computation. Students must understand the concepts behind these features to learn and effectively use these languages.

  Skill details:

  ▪ Identify computations that can be deterministically performed in parallel.

  ▪ Explain the parallelism in a pipelined computation built using streams.

- **Concurrency (1 core, 0 recommended)** describes computational models that can perform multiple computations nondeterministically.

  Topics: (1 hour)

  1. Non-deterministic composition and concurrency.
  2. Locking and synchronization mechanisms.

  Justification: Concurrency is a technique for composing together program fragments that may nondeterministically cooperate with each other. The nondeterminism and the resulting race conditions may be controlled by locking and synchronization mechanisms, to help make reasoning more tractable. Students should understand the basic techniques for eliminating nondeterminism (by, for example, using mutual exclusion locks or other explicit synchronization operations).

  With the rapid rise of multiprocessor and multicore systems, programming languages (including mainstream languages like Java, C#, and Python) are increasingly providing direct support for concurrency. Students must understand the concepts behind these features to learn and effectively use these languages.

  Skill details:

  ▪ Explain how various language mechanisms can be used to prevent race conditions or other non-deterministic behavior.

- **Run-time Implementations (3 core, 0 recommended)** describe how a programming language is evaluated on the underlying machine, including how data is stored in memory and how a program interoperates with the rest of the computing environment.

  Topics: (3 hours)

  1. Stack-based allocation for procedure and function calls.
  2. Heap-based memory management, explicit allocation and deallocation, and garbage collection.
  3. Basic optimization principles, including tail recursion elimination.

Justification: Understanding how languages are implemented is essential for effectively designing, implementing, and debugging programs. For example, the choice between explicit management versus garbage collection introduces many performance and correctness issues that must be understood. Memory management and other run-time choices can also impact system portability and security.

Additionally, it is important that students understand and appreciate the basic issues in performance analysis and program optimization, such as tail recursion. With a proper understanding of language implementation details, optimizations like recursion elimination can be used to dramatically improve a program's time and space behavior.

Skill details:

- Describe the basics of how data are allocated and managed and how function calls are processed.
- Recognize and evaluate the relative costs of language feature implementations, such as explicit memory management versus garbage collection.
- Rewrite a recursive procedure to be tail recursive.

2. Ability to define and analyze rigorous computational models. To be able to specify and analyze software systems, students must master the ability to reason about precise formal models. This skill is essential for effectively expressing algorithms and reasoning (either formally or informally) about program behavior. Semantic modeling techniques also enable students to more quickly learn new languages and to more effectively specify, analyze, and implement many types of systems. Key modeling concepts and skills include the following items.

- **Concrete and Abstract Syntax (2 core, 0 recommended)** specify the syntactic structure of a programming language.

  Topics: (2 hours)

  1. Regular expressions, context-free grammars, and their use in specifying programming languages.
  2. Abstract syntax trees.

  Justification: Regular expressions and context-free grammars are widely used to describe both programming languages, as well as protocols between computer systems and the structure of other forms of data. Being able to understand and use regular expressions and grammars is crucial for any programmer attempting to learn a new language. In addition, students should understand the relationship between the syntax of a programming language and the structures used to represent programs during compilation or formal modeling.

  Skill details:

  - Read and write regular and context-free grammars that describe small languages.
  - Use regular expressions to describe languages (sets of strings, as in pattern matching).
  - Use the lexical and context-free grammar of a new programming language as a reference when writing programs.
  - Write programs that perform simple manipulation of abstract syntax trees, such as substitution of names, type checking, or interpretation.

- **Dynamic Semantics (2 core, 0 recommended)** formally describes how programs are to be executed.

  Topics: (2 hours)

  1. Abstract machines, states, and transition systems.
  2. Evaluation semantics.

Justification: The dynamic semantics of a language precisely characterizes program behavior. Students must understand how to describe and use the dynamic semantics of a language to rigorously reason about the behavior of programs and to effectively compare the computation models of different languages.

The ability to rigorously model system behavior is widely-applicable beyond just programming languages. Typically, dynamic semantics is expressed as a transition system for an idealized abstract machine, and similar transition systems can be used to model API usage, communication protocols, various forms of algorithms, and so on.

Skill details:

- Define an abstract machine and transition system to model part of a complex system.
- Use the evaluation semantics for a small programming language to explain the behavior of a small program.

- **Static Semantics (4 core, 3 recommended)** describe constraints on the formation of valid programs.

  Topics: (4 hours)

  1. Type systems.
     (a) Type safety.
     (b) Static and dynamic type checking.
     (c) Parametric polymorphism and subtype polymorphism.
  2. Loop and data invariants.

  Recommended Topics: (3 hours)

  1. Judgments, rules, and derivations.
  2. Typing rules.
  3. Coherence of static and dynamic semantics.
  4. Formalization and proof of type safety as preservation and progress.

  Justification: Type systems capture important static properties of programs, and a language's type structure dictates how its features may be used. Moreover, types are an important source of abstraction in helping the programmer think about problems. Thus, type systems have been and continue to be central to both programming and to the study of programming languages.

  Students must understand the flexibility and limitations of type systems, as well as type system concepts in common use, such as parametric polymorphism (generics). Students should also understand how static type systems express properties that summarize all executions in an easily checkable manner (i.e., without flow sensitivity) and the limitations that this imposes on what programs can be written. These skills are important both for learning new programming languages and for working productively in existing languages.

  Students should understand how type systems help prevent programming errors and enforce abstraction boundaries. Given the prevalence of dynamic type systems in scripting and other commonly-used languages, students must also understand the difference between static and dynamic type checking. More broadly, the distinction between *dynamic* and *static* properties is fundamental to understanding the nature of computer systems, as well as to various kinds of algorithmic improvements. Type systems are the paradigmatic example of this distinction.

  Loop invariants and data structure invariants can capture deeper program properties that are not easily expressed in a type system. They can be checked either dynamically with assertions or statically with advanced verification tools. Invariants are important for understanding how one can reliably program and to analyze loops and data abstractions (such as object-oriented classes). These concepts are also important for correctness of algorithm design and for correctness of data structure design.

Skill details:

- Determine whether a given property is statically or dynamically checkable.
- Explain the difference between static and dynamic type systems and when each would be appropriate.
- During implementation, exploiting static properties to optimize algorithms and write code to check for dynamic properties.
- Determine the types of expressions, including expressions involving polymorphism.
- Show how to use type systems to prevent simple user errors.
- Write pre- and post-conditions for methods, and invariants for data structures, and enforce them with dynamic checks.
- Verify the correctness of a small program.

• **Language Features as Types (0 core, 3 recommended)** describe how the type structure of a language determines its primitive computational elements (e.g., data) and its means of computation.

Recommended Topics: (3 hours)

1. Principle of introduction and elimination.
2. Types as language features, languages as collections of types.
3. Propositions as types.

Justification: We recommend this topic because the types of a computational model (or programming language) determine most of its "features" by describing what kinds of data exist in a program, and how those data can be manipulated by a program. Introduction rules are type checking rules that say when an expression has a given type (for example, that `99` has type **int**); such rules describe the data used in the model. Elimination rules describe how an expression with a given type can be used (for example, by comparing two **int** expressions for equality). It is thus recommended that students understand the connection between types and language features, and how these ideas can be used in language design.

The idea of "propositions as types" (or the Curry-Howard isomorphism) connects type structure to behavioral reasoning. Students should understand that type systems can embody propositions (i.e., assertions) about the values of computations. For example, the proposition "it has an integer value" can be represented by the type **int**. Conversely, compound types (such as pairs) correspond to logical connectives (such as conjunction) in constructive logic (since a constructive proof of $A \wedge B$ involves proving both A and B). This idea has deep implications for language design and semantics, and is of practical value when considering how to check system properties.

• **Time and Space Complexity (0 core, 0 recommended, 2 outside)**

Topics: (2 hours, covered elsewhere in curriculum)

1. Asymptotic analysis of time and space usage.
2. Cost models for time and space.

Justification: A number of previous topics have focused on techniques to model and reason about the behavior of a language or system. However, students must also be able to rigorously reason about the performance of a system in terms of time or space complexity. To this end, students must learn how to define the cost of computation in a model and how to perform asymptotic analysis. (We list this topic here because, in our view, it is an essential skill for defining and reasoning about computation models and in the study of programming as a central tool for any form of algorithmic analysis. However, for historical reasons this topic is covered in other parts of the curriculum.)

Skill details:

- Use a cost model for a language to analyze the time and space complexity of algorithms expressed in that language.

3. Ability to define and implement abstractions. Students must obtain experience designing and implementing programs in a variety of programming models. Those who learn multiple ways to analyze and solve programming problems gain an understanding of a wider range of possible designs and implementations. Moreover, students who learn multiple programming models gain an appreciation for how underlying programming models affect analysis and implementation. For example, the absence of mutable state in purely functional computation models often makes it easier to reason about program correctness. These experiences are essential, not only for effective programming, but also for modeling, specification, and analysis skills.

Note that for functional programming and object-oriented programming, we label topics likely to be covered in introductory computer science or programming courses as "*Programming Topics*." We expect that the programming topics for one of either functional or object-oriented programming will be covered in these other courses and will not need to be repeated in the context of programming languages. When allocating lecture hours below, we assume that these other courses cover 7 hours of topics related to our object-oriented programming outcomes. However, the overall time allocated to programming topics would not change if instead students were previously exposed to functional programming.

- **Functional Programming (10 core, 0 recommended)** organizes computations around expressions, functions, and immutable data values (including functions themselves).

  Programming Topics: (7 hours)

  1. Recursion over lists, natural numbers, trees, and other recursively-defined data structures.
  2. Functional programming pragmatics, debugging by divide and conquer, and persistence of data structures.
  3. Functions as data and higher-order functions.

  Topics: (3 hours)

  1. Immutability and statelessness.
  2. Determinism.
  3. Laziness.
  4. Amortized efficiency for functional data structures.

  Justification: Functional programming is a well-developed, alternative model of computation quite different from procedural and object-oriented models. It illustrates alternate ways of expressing computation and thinking about problems, and it reinforces many basic computer science concepts. In addition to learning the basic principles and mechanics of functional programming, students should learn how to reason about the amortized performance of functional data structures, both to reinforce basic asymptotic performance analysis skills and to demonstrate how to adapt such techniques to a different computational model.

  Since most programming languages have expressions and immutable data (e.g., strings in Java), functional programming also enables students to be more expert in current languages. Many concepts from functional programming are migrating into other languages (for example, closures in C# and Java 7, and many features of Javascript and XSLT). One reason for this migration is that the handling of XML and other recursively-structured data in web services lends itself to functional programming techniques.

Functional programming techniques will likely become even more important with the prevalence of multicore processors, since immutable data can eliminate many race conditions in parallel programs. Ideas from functional programming also lead to simple, scalable system designs for large-scale distributed computations, as in Google's MapReduce system [2].

Skill details:

- Design, write, test, and debug programs using functional programming.
- Recognize tradeoffs in algorithm design when using functional programming.
- Perform amortized efficiency analysis for immutable data structures.
- Use higher-order functions to implement a program manipulating streams, infinite sets, or a similar structure.

• **Object-Oriented Programming (3 core, 0 recommended, 7 outside)** organizes computations around (possibly mutable) objects, which are data structures that encapsulate several fields and methods.

Programming Topics: (7 hours, covered elsewhere in curriculum)

1. Object-oriented design.
2. Encapsulation of state.
3. Message-passing and dynamic dispatch.
4. Classes, inheritance, and interfaces.

Topics: (3 hours)

1. Dynamic dispatch.
2. Inheritance, and the distinction between subtyping and inheritance.
3. Run-time representation of objects.

Justification: Object-oriented programming and object-oriented languages are widely-used, and many students are likely to learn some amount of object-oriented programming in their introductory courses. However, object-oriented languages contain many interesting design tradeoffs and potentially confusing issues that must be explored in relation to programming languages. Students must have a clear understanding of the fundamental concepts and semantics of such languages to use them effectively.

Skill details:

- Design, write, test, and debug programs using object-oriented programming.
- Implement a hierarchy of related classes.
- Recognize tradeoffs in systems design when using object-oriented programming.
- Explain the meaning of small programs involving inheritance and dynamic dispatch.

• **Parallel Programming (1.5 core, 1 recommended)** organizes deterministic computations to enable simultaneous execution of multiple subparts of a computation.

Programming Topics: (1.5 hours)

1. Data-parallel and task-parallel programming models.

Recommended Topics: (1 hour)

1. Speculative concurrency techniques, including futures.

Justification: Parallel programming techniques will grow in importance, due to the rapid rise of multi-processor and multi-core computers. This makes parallelism more likely to be used in everyday programming. Studying this subject from a programming language viewpoint enables students to see how a programming model affects their ability to analyze and construct systems for efficiently exploiting various hardware capabilities.

Skill details:

- Write a program to solve a programming problem using data parallelism in a particular programming model (e.g., with stream-based functional programming), and similarly for task parallelism.

- **Concurrent Programming (1.5 core, 1 recommended)** organizes computations to enable nondeterminisic interleaving of multiple subparts of a computation.

  Programming Topics: (1.5 hours)

  1. shared memory, threads and locking.
  2. asynchronous message passing.

  Recommended Topics: (1 hour)

  1. Serializability and weak correctness guarantees.

  Justification: Concurrent programming techniques are important for client-server systems and for designs in which multiple actors cooperate to incrementally build a system state.

  Since the language features that best support concurrency are still a subject of very active research and are likely to evolve over time, students must understand the principles behind concurrent programming models. Studying this subject from a programming language viewpoint enables students to see how a programming model affects their ability to analyze and construct systems for solving different kinds of problems.

  Skill details:

  - Write a program to solve a programming problem using shared memory, threads, and locking, and give an argument as to why the program is correct (avoids race conditions and deadlocks).
  - Write a program to solve a programming problem using asynchronous message passing.
  - Directly compare the strengths and weaknesses of different programming models by writing programs in those models to solve the same programming problem.

- **Programming Language Implementation (0 core, 3 recommended)** refers to the construction of language tools based on core programming language concepts. This includes being able to write a parser, type checker, interpreter, or compiler for a small programming language, given its precise formal definition.

  Recommended Topics: (3 hours)

  1. Representations for programs as data, such as abstract syntax trees.
  2. Algorithms for interpreting and manipulating program representations.

  Justification: We recommend this topic because translating a precise formal definition into an executable program deepens and tests one's understanding of that formal definition, and further reinforces the benefits of developing rigorous specification skills.

  Programming language systems are also excellent examples of programs that take complex input data and perform symbolic computation. Other examples include web browsers, printer drivers, PDF renderers, scripted robot control systems, spreadsheets, video and audio players, and so on [1]. Even if students never design a tool to manipulate programs written in a large, "general purpose" programming language, they will inevitably work on systems like these. To properly implement any form of symbolic computation, which can be quite subtle, students must understand the underlying theory and design principles, and they must be able to translate those principles into a working artifact.

For those students with a particular interest in language design, this experience will provide foundational skills for enabling one to encapsulate reusable knowledge in any area through the design and implementation of a domain-specific language.

Skill details:

- Translate a non-trivial formal specification for a small programming language into a correct implementation.
- Implement a type checker, interpreter, or translator, given the specification of a small programming language.
- Explain how to adapt the basic implementation techniques for programming language implementations to build other programming systems, such as debuggers or verification tools.

8. How We Should Teach Programming Languages: Changing the ACM Curriculum Unit Allocation

Lead Authors: Stuart Reges, Shriram Krishnamurthi, Kathleen Fisher, and Chandra Krintz

8.1 Proposal

In this section, we focus on how to teach programming language material effectively. One way to have an impact in this area is to influence the ACM/IEEE recommended curriculum. Many institutions worldwide use this recommendation in the design of their curricula. Consequently, we consider here how to modify the ACM/IEEE curriculum recommendation in small but high-impact ways. We start with a brief review of the background of the ACM/IEEE curriculum standards and then present our proposed changes. In the section that follows, we consider other ways of addressing the "How" question.

Since 1968 ACM and IEEE have published curriculum standards for computer science. Major revisions appeared in 1968, 1978, 1991, and 2001. The 2001 ACM/IEEE Computer Science Curricula report is available on the web:

`http://www.acm.org/education/education/education/curric_vols/cc2001.pdf`

Because the field has been changing so rapidly, the most recent standard advocates shifting to an on-going review process rather than the previous once-a-decade model.

The 2001 curriculum standard identifies the body of knowledge comprising computer science and divides it into fourteen areas. Each area is subdivided into *knowledge units*, which represent “thematic modules” within the area. The standard identifies each unit as either *core* or *elective* and associates a number of hours of classroom instruction with each core unit. The standard defines the major to include 280 hours of core material plus additional advanced courses. The standard is flexible in that departments can decide which advanced courses to include; however, the standard offers a number of possibilities. The authors of the 2001 standard adopted this more flexible structure in response to the explosion of topics in computer science and to allow each department to decide how best to complete the major.

The ACM/IEEE curriculum identifies programming languages as one of the fourteen areas in computer science, named using the acronym PL. It divides the area into eleven knowledge units, of which the first six are core with the required number of hours in parenthesis:

- PL. Programming Languages (21 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

The curriculum also defines Programming Fundamentals (PF) as an area with five knowledge units, all of which are considered core:

- PL. Programming Fundamentals (38 core hours)
- PF1. Fundamental programming constructs (9)
- PF2. Algorithms and problem-solving (6)
- PF3. Fundamental data structures (14)
- PF4. Recursion (5)
- PF5. Event-driven programming (4)

The knowledge units that relate directly to programming languages (and are thus commonly included in programming language curricula) are PF4 and PF5.

The ACM/IEEE curriculum recommendation suggests a variety of courses that cover the core knowledge units from the identified areas. In these courses, programming language material falls primarily in introductory courses that focus on programming (*e.g.*, CS111i, CS112i, CS101O, CS102O) or in courses that use object-oriented programming techniques. Other than in such courses, programming languages and compilers fall only in advanced courses that are electives in the curriculum.

Workshop participants (and others in the community) have expressed broad dissatisfaction with the position of programming languages in the undergraduate curriculum as reflected in the 2001 ACM/IEEE curriculum. In particular, there is concern that

- The curriculum does not sufficiently distinguish programming from programming languages.
- The knowledge units place too much weight on object-oriented techniques.
- Many key programming language ideas appear only in elective courses.

As an initial step in improving the ACM/IEEE curriculum, workshop participants proposed a change in the allocation of hours that affects *only* programming language knowledge units. The proposal does not change the knowledge units in the PL area nor the total number of required hours under PL, PF4, and PF5, only the knowledge units to which those hours were allocated. Table 1 overviews the proposed changes. The changes have two key design goals:

1. To make the functional programming unit (FP), PL7 in the curriculum, required rather than elective.
2. To account for this modification without changing the total number of required knowledge units.

In the following paragraphs, we first discuss the motivation for this change and then detail the reasoning behind the specific unit changes.

Shifting to functional programming is not merely about a change in syntax; rather, it forces students to approach problems in a novel way. This change increases their mental agility and prepares them for a life of practice in a world where languages continuously grow, morph, and sometimes shift their perspective. This

Knowledge Unit to be Changed		Current	Proposed	Change
PF4	Recursion	5	2	-3
PF5	Event-driven programming	4	2	-2
PL1	Overview of PL	2	0	-2
PL2	Virtual Machines	1	0	-1
PL3	Language Translation	2	0	-2
PL7	Functional Programming	0	10	+10
Total		14	14	0

Table 1. Proposed Changes to the 2001 ACM/IEEE Curriculum Required Knowledge Units

observation comes not only from the academic community: this advice comes from influential industrial practitioners including Joel Spolsky, Steve Yegge, Paul Graham, Eric Raymond, and Peter Norvig, who have all written extensively about it.

Moreover, the growth and change of languages, and the influence of functional programming, is not hypothetical. Virtually every significant mainstream language that has been designed, has gained prominence, or has been improved in the past decade, from JavaScript to Ruby to Java to C# to Visual Basic, incorporates notable features that used to be associated with functional programming. Datacenter programming techniques such as MapReduce grow directly out of functional programming. Microsoft has championed the use of functional constructs like closures in the .NET framework as the best way to express database and XML queries. The rise of multicore architectures is imposing new pressures on programmers to avoid the use of shared state whenever possible. Learning functional programming takes students directly to the source of these ideas, without the overhead and pain of sometimes unwieldy encodings of these ideas.

In addition, by exposing students to multiple languages before they graduate, we correctly convey what they can expect to see in practice. History shows that the dominant programming language changes roughly every seven years, and over two or three of these changes little stays the same other than syntax. Furthermore, modern systems are rarely built in a single language alone; developers find it advantageous, and sometimes necessary, to use a combination of languages of widely different styles. Therefore, exposing students to a variety of styles is an essential part of their education.

Given this motivation, we next discuss the rationale for the changes to specific knowledge units.

First, the topics in PF4 (Recursion) and PF5 (Event-driven programming) lend themselves naturally to coverage in both object-oriented and functional programming. Functional programming has traditionally made paradigmatic use of recursion, while the callbacks that guide event-driven programming are a natural fit when discussing closures. Furthermore, by seeing these topics in both contexts, students will be in a better position to compare and contrast their expression in different styles. As a result, we see both of these topics being covered naturally within the newly-expanded PF7.

Secondly, the material in PL1 (Overview of programming languages), PL2 (Virtual machines) and PL3 (Language translation) are important topics, but the small number of required hours allocated to them makes it difficult to say much of use within the allocated time. The result is an enumeration of jargon without any deep study of concepts. Students will be better served by a substantial exposure to a functional language that serves as a contrast to whatever object-oriented language they will also learn. This experience of learning two different languages is more important than a superficial coverage of these other PL topics.

Finally, one of our major goals is to bring parity between the number of required hours devoted to PL6 (Object-oriented programming), for which there are 10 knowledge units, and PL7 (Functional programming). The modified curriculum requires substantial experience with each approach, roughly 3 weeks of classroom time.

The proposal is neutral about how to implement the proposed change, Some schools will chose to incorporate functional programming into their introductory sequence. Other schools might include it in an advanced programming course or in a discrete structures course. And other schools will fit this material into a required programming languages course.

Note that the proposed change does *not* require that students use functional languages just as much as they use object-oriented languages during their studies. We anticipate that many courses will continue to use object-oriented languages for implementation work, and that as a result, students may end up with significantly more exposure to object-oriented languages.

Note also that this change is not lobbying for an untested change to programming language curriculum. As far back as Curriculum 68, the ACM has required a course in “programming languages,” which was generally interpreted to include a variety of programming paradigms. As recently as Curriculum 91, there were still 46 hours of required instruction in programming languages. The big shift came with the dramatic reduction of this material in Curriculum 2001, when the material was cut to 21 core hours, of which 10 were for object-oriented programming. This proposal is thus a modest change to restore some of the most important material that was cut in the previous revision but still has great relevance today.

8.2 Status

A joint ACM/IEEE Computer Science Curriculum Review Taskforce is currently revising the ACM/IEEE 2001 Curriculum. Because participants in the Programming Language Curriculum Workshop unanimously voted to support this proposal (with one abstention), Larry Snyder petitioned the Taskforce to incorporate the change into their revision. His description of the proposal is available on the web:

`http://wiki.acm.org/cs2001/index.php?title=SIGPLAN_Proposal`

The Taskforce accepted the petition and included the proposed change in their draft curriculum revision. They posted this draft on the web:

`http://wiki.acm.org/cs2001/index.php?title=Main_Page`

along with a request for comments from ACM/IEEE members during the period June 9th to July 1st:

`http://campus.acm.org/public/comments/comments_cs2001.cfm`

They later extended the review period to July 20th. Over 125 people posted comments on the proposed changes to the programming language curriculum, with the vast majority (>95%) expressing strong support for the change.

After reviewing the community comments, the committee decided to add the following section to Chapter 9 of the revision to the ACM/IEEE curriculum:

```
9.1.5 Exposure to different programming paradigms

Computer science professionals frequently use different programming
languages for different purposes and must be able to learn new
languages over their careers as the field evolves.  As a result,
students must recognize the benefits of learning and applying new
programming languages.  It is also important for students to recognize
that the choice of programming paradigm can significantly influence
the way one thinks about problems and expresses solutions of these
problems.  To this end, we believe that all students must learn
to program in more that one paradigm.

Although we believe it is essential for all students to acquire
experience with more than one paradigm, we believe that the choice
of an appropriate secondary paradigm will depend significantly on
the specific character and educational goals of each institution.
```

`Universities that seek to prepare students for positions in academia, research, and advanced development would be well advised to introduce functional programming so that students can take advantage of the mental discipline that those languages encourage. Programs that seek to prepare students to develop web-based applications might choose instead to use scripting languages. Such languages seem to be growing in commercial importance and are increasingly adopting features that were typically associated with functional languages in the past.`

Informal feedback suggests the committee felt that the shift to requiring ten units of functional programming was too ambitious a change to be made during this interim review, and that the question should be revisited during the upcoming full revision.

9. How We Should Teach Programming Languages: When There is No Programming Languages Course

Lead Authors: Eric Allen, Mark Bailey, Ras Bodik, Doug Lea, Mark Sheldon, Lori Pollock, Franklyn Turbak, and Mitchell Wand.

In this section, we respond to the question, "*How can an institution support core units of programming languages in a curriculum that does not require a dedicated programming language course for all students?*". Although there has been a strong tradition in computer science curricula of dedicating an entire course to the topic of programming languages, we have already seen, and will continue to see to an increasing degree, programs of study that do not follow in this tradition. With the expansion of the field and the growing importance of other subfields, this situation is inevitable. In recognition of this challenge, the committee considered how best to support these institutions in teaching core units in programming languages.

Workshop participants identified core programming language concepts that every undergraduate student should study. These are detailed in Section 7. Covering these topics outside the structure of a formal course in programming languages is extremely challenging and not recommended by the committee. However, the committee does see the efficiency a program can gain by integrating these topics into other courses considered contemporary or trendy by faculty, students, or others who influence a program's course offerings. The committee discussed three such approaches: sprinkle topics throughout the curriculum, a CS3 course emphasizing advanced programming techniques, and integration of topics into targeted, "relevant" courses with strong connections to programming languages.

Sprinkling Topics. In this model, core units are integrated into a large number of required courses in the curriculum. This approach is highly efficient in guaranteeing coverage without the cost of an additional course. It also demonstrates the impact the programming language community has had throughout the field of computer science. In some cases, it may even encourage instructors to use techniques or introduce concepts that are more effective than those previously used in the course. However, this approach requires careful planning of the entire curriculum and coordination of the entire faculty to ensure unit coverage, and makes the curriculum brittle to modification. It is unlikely that this approach can be effective in departments with large numbers of faculty or in programs that require very few courses.

Advanced Programming in CS3. There is a growing interest in introducing a third course in the introductory programming sequence — a CS3 to expand the breadth provided by CS1 and CS2[8]. Such a course might very well focus on programming-in-the-large, advanced programming techniques, or other topics not covered in the first two courses in the sequence. The committee sees this as a particularly attractive course in which to cover many of the core programming language concepts. This course might include topics such as threading, event-driven programming, hot current languages like Python, Ruby (on rails), *etc.* Such a course could easily include many core programming language topics including interpreters, exceptions, concurrency, parallelism, transactions, managing state, security, *etc*. In many respects, the CS3 approach most closely models the pivotal role programming languages play in the advanced programming techniques that students will use every day in their professional careers.

Targeting Courses. Many institutions have courses in specialized topics with strong connections to programming languages. These include courses in web services, software engineering, formal methods, models of computation, virtual machines, concurrent or parallel programming, databases, and computer systems. It would be reasonable to provide a "programming language centric" approach to teaching one or more of these courses. In a web services course, problems arise where instructors can use core programming language concepts to describe and implement solutions efficiently and with elegance. For example, handling transactions with state in a stateless web server can be solved using continuations to model web browser cookies. Opportunities to use these core concepts in related courses are plentiful and illustrative.

Committee Recommendations. In considering these approaches, many committee members felt these models further supported the position that a dedicated programming language course should be required. Given only these options though, the committee recommends the inclusion of a CS3 advanced programming techniques course in a curriculum otherwise lacking a programming language course. The committee's second choice would be one or more targeted courses. The committee felt strongly that the remaining alternative — sprinkling topics — is unlikely to be successful given the faculty coordination requirement.

10. How to Teach Programming Languages: The Single PL Course

Shriram Krishnamurthi, Robert Harper, Matthias Felleisen, and Greg Morrisett

As one of the oldest and deepest disciplines in computer science, Programming Languages offers ample material for a stand-alone upper-level undergraduate course. The value of such a a course is, however, a function of the methods used to teach it. Experience leads the workshop attendees to heartily reject the long-standing practice of teaching programming languages strictly as a survey of existing programming languages or "paradigms". Modern languages have long exceeded the grasp of these paradigms, which are tedious, uninformative, and moribund. The complex trade-offs that guide their design are simply not amenable to description and evaluation through prose or sample programs alone.

A modern approach demands a balance between intuition and rigor. While concrete languages provide intuition, more mathematical means are necessary for rigor. Various techniques such as definitional interpreters, formal semantics, type theory, and more demonstrate not only the meaning of existing languages but also the constraints that are necessary to generate coherent designs for new ones.

A thorough understanding of these constraints is critical for our students, who will have to constantly make decisions about new languages and novel application contexts. Without a rigorous understanding of this material, students will be unable to appreciate the trade-offs present in language design, and therefore cannot soundly evaluate and apply language technology. Some of these students will even go on to design (at least) little languages of their own, in the guise of protocols, specification formats, and large libraries. It is therefore critical for programming languages curricula to modernize.

Part III
Workshop Report Summary

In response to the concern that undergraduate computer science curriculum in general and programming language curriculum in particular are out of date, workshop participants met for two days to discuss why undergraduate computer science majors should learn about programming languages, what they should learn, and how they might be taught this material. This report summarizes the discussions during the workshop and includes specific recommendations to improve undergraduate programming language education, as described in the previous sections. Given the size of the task of modernizing curriculum and the limited time frame of the workshop, the task is necessarily not complete. In fact, it has only just begun. With this consideration in mind, workshop participants recommended that SIGPLAN create an Education Board to continue the task. The SIGPLAN Executive Committee has approved this idea, and such a board will be constituted shortly. The charter of the board will include:

- Soliciting community feedback on an on-going basis as to programming language curriculum.
- Producing a concise description of why undergraduate computer science majors should learn about programming languages for a non-programming languages audience.
- Producing concrete course descriptions and syllabi covering the identified topics.
- Working with the ACM Education Board to modernize the ACM/IEEE Curriculum and the role of programming language instruction in that curriculum.
- Exploring whether and how to conduct educational studies on the efficacy of instruction in various programming languages.
- Identifying any other activities that will enhance programming language education.

The goal of modernizing undergraduate programming language curriculum can only be successful with the active participation of a wide array of experts in the field, from educators to practitioners to researchers. If you are such an expert, please consider participating in future activities!

References

[1] Eric Allen. Some things that computer science majors should know. `http://www.sigplan.org/pl-workshop/`, May 2008. Checked June 16, 2008.

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

[3] Danny Dolev and Andrew Chi-Chih Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–207, 1983.

[4] Kathleen Fisher and Chandra Krintz. Mission statement of the first SIGPLAN workshop on undergraduate programming language curricula. `http://www.sigplan.org/pl-workshop/mission.txt`, May 2008. Checked June 15, 2008.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1995.

[6] 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.

[7] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[8] T. Vasiga. What Comes After CS 1 + 2: A Deep Breadth Before Specializing. In *Proc. of the SIGCSE Technical Symposium on Computer Science Education*, pages 28–32, 2002.