Improving Undergraduate Programming Language Curriculum

Kathleen Fisher
AT&T Labs Research
SIGPLAN Chair

A Common Perception...

“The programming language problem has been solved; the name of the solution is C.”

- Dean of a top-3 research university, circa 1991.

“Well, Java”

- Same dean, several years later.

The Reality...

bash F# XSLT XQuery

perl Python javascript

ActionScript C Haskell S

Ruby Ocaml Java SQL

awk PHP C++ C# R latex

Tcl/TK postscript make ML

Visual Basic regular expressions

Explosive Growth

- In the past 20 years, computer science has deepened intellectually, broadened in scope, and penetrated all areas of modern life.
  - The web, electronic commerce, social networks, graphics, gaming, image processing, security, natural language processing, bio informatics, machine learning, low power processing, multi-core processors, sensor networks, Google-style distributed processing, XML, software transactions, databases, operating systems, networks...
- This growth puts enormous pressure on undergraduate curriculum.

Curriculum Revision

- In response to expansion, ACM/IEEE revised its CS Curriculum in 2001 (CC 2001).
  - Cut required units
  - Enabled more flexibility
- CC 2001 was widely influential.
  - One of the most often downloaded documents from the ACM Digital Library.

http://www.sigcse.org/cc2001/cs-overview-bok.html#BOKTable

CC 2001 Curriculum

- PL went from 46 required units to 21:

	Description	Units, if required
PL1	Overview of PL	2
PL2	Virtual Machines	1
PL3	Introduction to Machine 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	

- Required units ended up in introductory programing and project courses. Many institutions dropped a separate programming language course.

Possible Explanations

- Almost everything was cut to some extent.
- PL community doesn’t explain itself well.
  - Did not participate in 2001 curriculum review until very late in the process.
  - Often undervalues collaborations across areas as “applications” are not of fundamental interest.
- Broader community perceives PL as
  - a “solved problem” (only need C, err..., Java!)
  - lacking in principles (arising from taxonomy courses: if it’s Tuesday, it must be COBOL... ?)

How should we respond?

- Ignore the situation.
- Accept it as inevitable and get back to doing our day jobs (“real work”).
- Accept it as the right state for undergraduate education.
- Reject it and work towards a better standard curriculum.

ACM SIGPLAN
Workshop on
Programming Language Curriculum

- Motivation: Initiate discourse on the role of programming languages in the undergraduate curriculum.
- Co-chaired by Kathleen Fisher and Chandra Krintz
- Held May 29 & 30, 2008 at Harvard
- Sponsored by NSF, NSA, and SIGPLAN Thanks!!
- 30 participants
  - 16 steering committee members, 13 authors of selected white paper contributions, NSF and ACM Ed Board representatives

Participants

Eric Allen (Sun Microsystems)
Mark Bailey (Hamilton College)
Ras Bodik (UC Berkeley)
Kim Bruce (Pomona College)
William Cook (UT Austin)
Matthias Felleisen (Northeastern Univ.)
Kathleen Fisher (AT&T Research)
Kathi Fisler (WPI)
Daniel Friedman (Indiana Univ.)
Stephen Freund (Williams College)
Sol Greenspan (NSF)
Robert Harper (CMU)
Michael Hind (IBM Research)
John Hughes (Chalmers)
Chandra Krintz (UC Santa Barbara)
Shriram Krishnamurthi (Brown)

Jim Larus (Microsoft Research)
Doug Lea (SUNY Oswego)
Gary Leavens (Univ. of Central Florida)
Greg Morrisett (Harvard Univ.)
Benjamin Pierce (Univ. of Pennsylvania)
Lori Pollock (Univ. of Delaware)
Stuart Reges (Univ. of Washington)
John Reynolds (CMU)
Martin Rinard (MIT)
Olin Shivers (Northeastern Univ.)
Peter Sestoft (ITU)
Mark Sheldon (Wellesley College)
Larry Snyder (Univ. of Washington)
Franklyn Turbak (Wellesley College)
Mitchell Wand (Northeastern Univ.)

Mission Statement

- Explosive growth in CS in general and PL in particular
  - Internet, multi-core, managed runtime systems, etc.
- Most curricula have not kept pace
  - Some curricula no longer include a PL course at all
  - ACM/IEEE curriculum only minimally covers PL concepts
- Need to consider as a community
  - WHY PL should be included in the CS curriculum
    - Clear articulation for non-PL academics of why every computer science undergraduate should have a solid PL knowledge base
  - WHAT topics and concepts should be taught
    - Broad audience, many constraints, range of career paths and goals
    - To every student and to those choosing to study PL.
  - HOW it should be taught
    - Recommended practices for range of venues, audiences, constraints

Why should we teach PL?

A Plethora of Languages

- From early on, there have been many programming languages.
- As the field advances, more and more languages are created.
- Today, programmers need to be fluent in multiple languages.

Python Java Cryptol ActionScript

Tcl/TK Ruby bash postscript Haskell C# make SQL latex

PHP Visual Basic Ocaml C XQuery R S ML

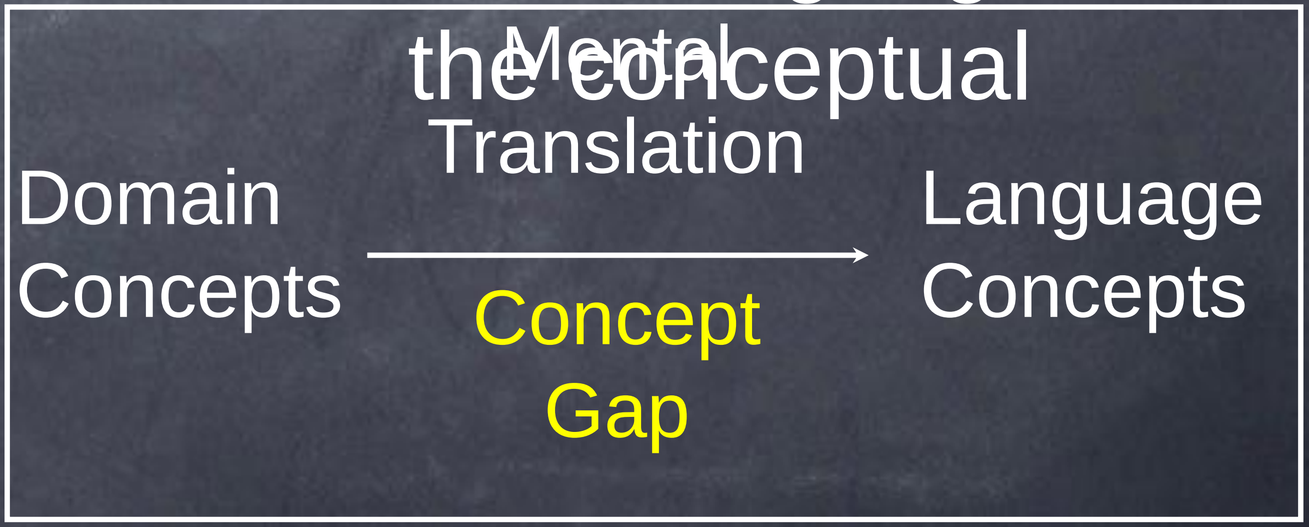
C++ perl XSLT regular expressions awk F# javascript

1) Learning New Languages

- We want to teach our students enduring knowledge, not the latest fads.
- Hence, it is more important that students learn how to learn a language rather than the particulars of a single language.
- Understanding PL fundamentals makes learning languages much easier.

Hammers and Wrenches

- Each language embodies a computational model.
- A task is most easily solved in a language whose model matches the domain of the task.
  - C: low-level systems code
  - Java: user-interfaces, modeling
  - ML: writing compilers
- Choosing a good language minimizes the conceptual gap.



2) Using the right tool

- What language(s) are best suited to the my task?
- Is it reasonable to expect my team to accomplish this task in this language?
- If I am stuck in a particular language, can I import ideas from another language?
  - Mimicking objects in C.
  - Mimicking continuations in Java
- Knowing PL fundamentals encourages programmers to ask these questions and then helps to answer them.

Better Wrenches

- Sometimes the best way to solve a problem is to build a language tailored to it.
- Often, such languages are written by domain experts whose only PL training came as an undergraduate.

API Design

Configuration Files

Little Languages

3) Building Languages

- Not many people design new general-purpose languages.
- But many people design APIs, configuration languages, and domain-specific languages.
- Understanding PL fundamentals allows students to know when and how to build such “little languages”.
  - What are the abstractions? How do they interact? What should be expressible? What should not be?

Language and Thought

- Programming languages are relatively simple, consciously-designed means for communication.
- Yet, they embody concepts common to all human communication:
  - abstraction
  - automation and generality
  - existence of multiple perspectives

4) Understanding Thought

- Studying PL provides a unique opportunity for studying interactions between language and thought because the languages are very precise and they can be changed.
- For example:
  - No one perspective is suited to all tasks.
  - Adopting a particular perspective can make a huge difference in one’s experience and success.
  - Incorporating diverse perspectives can increase chances of success.

Why teach PL?

- Students should be able to:
  - Learn languages quickly.
  - Evaluate suitability of a language for a task.
  - Know when and how to design little languages.
  - Understand the effects of language on thought and communication.
- Other arguments?

What should we teach?

The Traditional Approach

- A survey of well-known programming languages.
  - A taste of Java, Lisp, Prolog, ...
  - Selection based on current fads.
- A taxonomy of "paradigms."
  - x-oriented programming languages, for various values of x.
  - "declarative" vs "non-declarative"
  - "scripting" languages, "domain-specific" languages.
  - ... and various other distinctions according to taste.
- There are many textbooks written in these styles.

Credit: Bob Harper drafted the what slides.

What’s Wrong With This?

- Students accumulate vitae items, but little understanding.
  - How to transfer ideas to a new setting?
  - Fad-driven, quickly obsolete.
- More importantly, it is unscientific.
  - Based on a superficial morphology (Gould’s Zebra).
  - No theory of what is going on.
  - No permanent results, but lots of opinions.

What We’re Advocating

- A scientific theory of programming languages based on abstract models of computation.
  - Supports precise definitions and rigorous analysis.
  - Transfers to new problem domains.
  - Separates abstraction from implementation.
  - Enables verification.
  - Emphasizes enduring principles, not current trends.
- The field of programming languages has changed dramatically over the past 25 years!

A Formalism-Based Approach

- A programming language model is specified by
  - A static semantics defining the structure of the model.
  - A dynamic semantics defining the execution steps.
- The key to such a model is
  - Abstract: highlights essence, not accident.
  - Analyzable: supports rigorous proof.

Example: Parallel Computing

- Theme: parallelism is about performance, not concurrency.
  - Deterministic: same meaning as sequential
  - Efficient: makes better use of processors.
- Two ingredients:
  - Abstract cost semantics: sequential and parallel cost measures.
  - Concrete realization: communication and scheduling costs.

Example: Parallel Computing

- Sequential execution:

$$\frac{e_1 \mapsto_s e_1'}{e_1 + e_2 \mapsto_s e_1' + e_2} \qquad \frac{e_2 \mapsto_s e_2'}{n_1 + e_2 \mapsto_s n_1 + e_2'} \qquad \frac{}{17 + 18 \mapsto_s 35}$$

- Idealized parallel execution:

$$\frac{e_1 \mapsto_p e_1' \quad e_2 \mapsto_p e_2'}{e_1 + e_2 \mapsto_p e_1' + e_2'} \qquad \frac{}{17 + 18 \mapsto_p 35}$$

- Theorem [Implicit Parallelism]: $e \mapsto_s^* n$ iff $e \mapsto_p^* n$.

Example: Parallel Computing

- Time complexity = number of steps:
  - Sequential: $T_s(e) = k \text{ s.t. } e \mapsto_s^{(k)} n.$
  - Parallel: $T_p(e) = k \text{ s.t. } e \mapsto_p^{(k)} n.$

- Theorem [Efficient Implementation] If sequential complexity of e is w and parallel complexity is w, then e runs on a p-processor RAM in time O(w/p + d).

A Selection of Topics

- Abstract and concrete syntax, binding and scope.
- Semantic specification of models: static and dynamic.
- Finite and infinite data structures.
- Modularity, genericity, and data abstraction.
- Time and space complexity, cost semantics.
- Laziness, speculative parallelism.
- Deterministic parallelism.
- Indeterminism and concurrency.
- Mutable storage
- Continuations, exceptions, and coroutines.
- Run-time systems, storage management, scheduling.

An interim approach to what we should teach

Can we do something now?

- We were in a heated debate about curriculum, with Matthias Felleisen, Shriram Krishnamurthi, and Stuart Reges arguing.
- Stuart said “I wish we could agree on something simple, like asking the ACM to require functional programming in the CS core.
- Matthias hugged Stuart (really).
- Matthias, Shriram, and Stuart wrote a proposal.
- It was unanimously approved by workshop participants.
- We all sang “Kumbaya”

(Okay, that part is not true).

Credit: Stuart Reges drafted the ACM proposal slides.

What did we mean by “functional programming”?

“I shall not today attempt further to define the kinds of material I understand to be embraced within that shorthand description; and perhaps I could never succeed in intelligibly doing so. But I know it when I see it.”

–Justice Potter Stewart on “obscenity”

These are a few of my functional things

- Functions as first-class entities in a language (e.g., can be passed as parameters)
- Higher-order functions (map, filter, reduce)
- Avoidance of mutable state
- Closures (combination of code and context)
- Anonymous functions (lambdas)
- Use of lambda calculus for formal definitions

Why functional programming?

- FP stretches students’ understanding of programming.
- FP constructs are appearing in popular programming languages (Python, Ruby, JavaScript).
- FP is showing up in industry
  - Microsoft: F#, LINQ
  - Google: MapReduce
  - Yahoo: Hadoop
- FP has deep intellectual roots dating back to Alonzo Church (1936) and with significant interest ever since (LISP-1958, ML-1973, Scheme-1975, Erlang-1987, Haskell-1990)

Proposal: Revenue Neutral Adjustment to ACM/IEEE CC 2001

Affected Knowledge Units (of 59 in PF/PL)	Current	Proposed
PF4 Recursion	5	2
PF5 Event-driven programming	4	2
PL1 Overview of PL	2	0
PL2 Virtual Machines	1	0
PL3 Language Translation	2	0
PL6 Object-oriented programming	10	10
PL7 Functional Programming	0	10
Total Number of Units	24	24

Community Comments

- ACM/IEEE CC Interim Review Committee accepted proposal for inclusion in draft revision.
- Committee ran an open comment period from 6/9/08 to 7/16/08.
- The FP10 proposal received the most comments (by an order of magnitude).
- Of the 135 people who commented, at least 130 were supportive.

Where do we stand now?

“The review committee was divided in its response to the SIGPLAN proposal. On the positive side, the committee was convinced that students need exposure to more than one programming paradigm, for precisely the reasons outlined in the proposal. At the same time, there was no consensus within the review committee that the functional programming paradigm needed to be required in all undergraduate computer science curricula…We did not believe that we could justify making so far-reaching a change, particularly at the level of an interim review. Our consensus recommendation is therefore to add a new requirement that students acquire facility with more than one programming paradigm…The review committee also plans to forward both the SIGPLAN proposal and the notes from our discussions to the next full-scale curriculum committee.”

How should we teach PL?

Answer

Teach a dedicated, modern, programming language course that includes the core unit recommendations

Credit: Mark Bailey drafted the how slides.

Not Always the Answer

- Course slots are scare resources:
  - Expansion of the field
  - Growing importance of other subfields
  - Emphasis on “trendy” topics
  - Development of “track” curricula
  - Small departments, limited resources

Charge

How can an institution provide core units of programming languages in a curriculum that does not require a dedicated programming languages course for all students?

Alternatives

- Advanced programming in CS3
- Targeting Courses
- Sprinkling topics throughout curriculum

Advanced Programming in CS3

- Introduce a third course in the intro sequence:
  - Focus: programming-in-the-large, or advanced programming techniques
  - Topics: threading, event-driven programming, hot current languages
  - Core units: interpreters, exceptions, concurrency, managing state, security, etc.
- Advantages:
  - Easy integration
  - Acknowledges key role of PL in advanced programming
- Shortcomings: May conflict with other plans for CS3.

Targeting Courses

- Teach related courses using a PL-centric approach
- Examples:
  - Web services, software engineering, formal methods, virtual machines, databases, compilers.
- Advantages
  - Efficient
  - Builds connections
- Shortcomings
  - Difficult to get other faculty to do
  - Unlikely to give sufficient coverage of core units

Sprinkling

- Integrate core units into required courses
- Advantages
  - Highly efficient
  - Demonstrates PL impact throughout curriculum
- Shortcomings
  - Requires careful planning and coordination
  - Makes curriculum “brittle”
- Conclusion: Unlikely to be effective
  - Departments with large faculties
  - Programs with very few courses

Recommendations

- Recommendations, in order of preference:

  1. Dedicate a course to programming languages

  2. Include a CS3 Advanced Programming Course

  3. Provide one or more targeted courses

- We recommend against sprinkling because of the faculty coordination requirement.

Workshop Outcomes

- Published findings in Nov. 2008 issue of SIGPLAN Notices.
  - Why, What, and How Summaries
  - White papers from each workshop participant.
- Initiated interactions with ACM/IEEE Curriculum Review Committee.
  - FP 10 proposal
- Presented panel at SIGCSE 2009 to engage computer science education community.
- Recommended creation of SIGPLAN Education Board.

SIGPLAN Ed Board: Charter

- Create “Why” document for non-PL people.
- Create detailed “How” documents
    - Sample curricula
- Encourage authors to write appropriate texts.
- Create web page to present material and encourage discussion.
- Eventually expand focus from undergraduate courses to graduate courses.
- Encourage inclusion of appropriate PL material in curricula standards.

SIGPLAN Ed Board: Members

- Kim Bruce (chair), Pomona College
- Kathleen Fisher (ex officio), AT&T
- Kathi Fisler, WPI
- Steve Freund, Williams College
- Dan Grossman, University of Washington
- Matthew Hertz, Canisius College
- Gary Leavens, University of Central Florida
- Andrew Myers, Cornell University
- Larry Snyder, University of Washington

Your Turn

Feedback, stories, and suggestions are much appreciated!