

Programming Languages: Fundamental Concepts for Expanding and Disciplining the Mind

Mitchell Wand

College of Computer and Information Science
Northeastern University
360 Huntington Avenue, #202WVH
Boston, MA 02115
wand@ccs.neu.edu

Daniel P. Friedman

Computer Science Department
Indiana University
101 Lindley Hall
Bloomington, IN 47405
dfried@cs.indiana.edu

Abstract

In this white paper, we propose a list of essential concepts of programming languages, and discuss the techniques we have used to teach these concepts.

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

General Terms Languages

Keywords curricula

1. Introduction

We view the undergraduate course on programming languages as a primary means of expanding the student's horizons beyond the Java monoculture. We aim to make the student dissatisfied with the status quo, and thereby create an audience for new and better programming tools.

2. Our list of essentials

We formulate our essentials mostly as a list of contrasting concepts.

- **syntax vs. semantics:** real-world entities, such as numbers, vs. their representation as values vs. their expression as program literals.
- **names vs. values.**
- **interface vs. implementation:** the all-important distinction between the outside and the inside of an abstraction.
- **environment vs. store:** possibly the most fundamental distinction in programming languages, and largely lost in the master narrative of CS curricula. Our slogan is: binding is about communicating values; assignment is about sharing. In this heading we also include static vs. dynamic binding of names, a concept that is crucial when the student gets to systems.
- **continuations:** the third leg of the classical triangle of PL notions, along with environments and stores.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLC '08 May 29-30, 2008, Cambridge, MA.
Copyright © 2008 ACM [to be supplied]...\$5.00

- **compilation vs. evaluation:** including various notions of analysis time, translation time, link time, execution time, etc.
- **types:** both as contracts for procedures and as a way of expressing abstractions.
- **invariants:** Though we customarily think of invariants in terms of imperative languages, we can profitably study invariants in a functional setting instead. This approach goes back to subgoal induction (Morris and Wegbreit 1977).
- **modules:** both as simple devices for name control, and for creating and enforcing data abstractions. Parameterized modules as a way of controlling intermodule dependencies.

We add to this list a pair of things we wish were essentials:

- **macros and meta-programming tools:** coming from the Lisp/Scheme culture, we believe macros are a fundamentally important idea because they are the right way to do metaprogramming. Alas, we recognize that this is a minority view.
- **abstractions for concurrency and distribution:** almost all serious systems involve some level of concurrency and/or distribution, even without the much-ballyhooed advent of multi-core machines. There is a growing consensus on message-passing systems, like those embodied in Actors, Erlang, or the join calculus, as a fundamental abstraction for programming such systems. Transactions are another emerging consensus abstraction.

3. Modalities for teaching these concepts

Good pedagogy demands the use of multiple modalities for conveying these concepts. These should include:

- English text, both to describe the concepts ("What's the solution?") and to provide use cases for various constructs and design choices ("What's the problem?")
- Interpreters for assorted mini-languages, to provide the student a hands-on learning experience.
- Lightweight operational semantics, along with some use of equational reasoning. Automated tools for doing this kind of reasoning, such as PLT Redex or ACL2, can be useful in making these techniques practical.

4. EOPL3

As one might expect, this list contains more than any reasonable undergraduate course can cover. We next describe the path we have taken in our new edition of Essentials of Programming Languages

(Friedman and Wand 2008). We apologize in advance if this presentation seems self-serving.

We first emphasize the connection between inductive data specification and recursive programming and introduce several notions related to the scope of variables. Next, we introduce a data type facility. This leads to a discussion of data abstraction and examples of multiple implementations of a single data type.

Next, we introduce interpreters as mechanisms for explaining the run-time behavior of languages and develop an interpreter for a simple, lexically scoped language with first-class procedures and recursion. Starting from this interpreter, we study the translation to deBruijn notation, an example of a simple program analysis and transformation.

We introduce state as a technique for sharing data between different portions of a program and present two programming models for dealing with it: an explicit-reference model like that in ML, and a more conventional implicit-reference model that uses assignment. We show how state can replace tail-recursion in both models. We introduce a language with mutable pairs, and we explore call-by-reference, call-by-name, and call-by-need parameter-passing mechanisms.

Following this, we introduce continuations to our basic interpreter. The control structure that is needed to run the interpreter thereby shifts from recursion to iteration. This exposes the control mechanisms of the interpreted language, and strengthens intuition for control issues in general. We show how to implement the continuation-passing interpreter in an imperative language, using the standard transformation from tail-recursive to imperative programs, and also using trampolining to avoid stack overflow in languages (like Java, C, and C#) that do not implement tail calls properly. We use continuations to add exception-handling and multi-threading mechanisms to our base language. We then present an algorithm for converting a larger class of programs into continuation-passing style. In keeping with our pedagogy, we present the algorithm through examples, through equational reasoning, and in the form of a program

We then turn our attention to types; including an analysis-time system to do type checking and unification-based type inference. We do only the monomorphic case in the text, leaving Hindley-Milner polymorphism to an exercise. We then introduce modules as a way of using the type checker to build and enforce abstraction boundaries. Last, we present the basic concepts of class-based object-oriented languages. We first develop an efficient run-time architecture. Then we combine the ideas of the type checker with those of the just-introduced object-oriented language, leading to a conventional typed object-oriented language. This requires introducing new concepts including interfaces, abstract methods, and casting.

Although we write our program interpretation and transformation systems in Scheme as an executable meta-language, any other language that supports both first-class procedures and assignment will do.

This is a hands-on course: the code for everything we discuss is publicly available and may be exercised within the limits of a typical undergraduate course. Because the abstraction facilities of functional programming languages are especially suited to this sort of programming, we can write substantial language-processing systems that are nevertheless compact enough that one can understand and manipulate them with reasonable effort.

The derivation of a sequence of interpreters ranging from very high- to very low-level does more than provide a solid, hands-on understanding of programming language semantics and a disciplined approach to language implementation. It also teaches an approach to programming that starts with a high-level operational specification, which also serves as a rapid prototype, and ends

with what is effectively assembly language. We believe that the program-transformation techniques used in this process should be in the toolbox of every computer scientist. In fact, sometimes we derive C code to run the interpreter tail-recursively, at once pointing out the drawbacks of using tail-recursion in C, and showing how to avoid these problems using trampolining or register architectures.

References

- Daniel P. Friedman and Mitchell Wand. *Essentials of Programming Languages*. MIT Press, Cambridge, MA, third edition, 2008.
- James H. Morris, Jr. and Ben Wegbreit. Subgoal induction. *Communications of the ACM*, 20:209–222, 1977.