

# Implementing Domain-Specific Languages as the Foundation of an Honors Intro CS Course

Kathi Fisler, WPI Department of Computer Science  
kfisler@cs.wpi.edu

April 9, 2008

## 1 Introduction

While there is extensive discussion about appropriate introductory curricula for CS1, these discussions typically overlook a particular group of college-level “introductory” students: those with prior programming experience. Many schools give these students placement credit and let them start in CS2. Some have them take the normal CS1, especially if it covers more than the standard high school course. If a school has only a few such students, these are cost-effective options. They are, however, pedagogically unsatisfying. Students with prior experience are often our most passionate potential majors. Ideally, we would like to expose them to new and challenging material from the day they arrive. Furthermore, CS1 and CS2 are often both programming courses (even if CS2 nominally covers data structures). For strong potential majors, we want them to start viewing CS as more than just programming as early as possible. Finally, we want to set the tone that college-level CS has a lot to teach and offer talented students beyond what they have already seen. Thus, there are clear advantages to offering options for these students when we can afford to do so.

For schools with the luxury to offer a separate course for these students, what makes for an appropriate curriculum? This position paper argues that programming languages is an ideal foundation for such a course. By “programming languages”, I do not mean traditional courses based on paradigms or meta-circular interpreters. Rather, I want students to see the computing world linguistically. When asked to develop a new software product, do they see the language(s) embedded within the domain? Do they think to conceive of a programming task not in C++ or Java, but in a vocabulary that is native to the problem they are trying to solve? Do they understand how to create a domain-specific language as a programming abstraction for purposes of program evolution and maintenance? In short, a first course with students with prior programming experience is a terrific opportunity to introduce the mindset and core techniques of programming languages.

This vision is embodied in a course that I have developed and taught over the last four years. Concretely, the course covers functional programming, rudiments of designing domain-specific languages, implementing languages through interpreters, and writing customized front-ends through macros. The rest of this paper details the structure of the course, its design features, and our experiences with it. Recent lecture notes and assignments are all available online at <http://www.cs.wpi.edu/~cs1102/a07/>.

## 2 Course Description and Context

The course is offered in twenty-eight one-hour lectures over seven weeks (the standard course format at WPI). The course uses Scheme, partly for technical reasons (the macro system), partly for pedagogic ones (the availability of suitable tools and texts, as well as consistency with our introductory course for novices) and partly out of instructor preference. The material is divided into the following thematic units:

- *An introduction to functional programming.* In eleven lectures, we cover the basics of Scheme, programming with lists and trees, and higher-order functions. This portion of the course follows Felleisen

*et al.*'s "How to Design Programs" text [1]. This works particularly well for this course as it uses the interpreter pattern [2] throughout the lists and trees material, which nicely sets up the later units.

- *An introduction to languages as data.* Many students struggle when they first see interpreters because they are unaccustomed to viewing programs as data that other programs can process. We design a simple slideshow (ala Powerpoint) language as a running example, starting from a concrete example of a program that displays a fixed slideshow (and has no abstraction). The goal is to show students how languages arise from making abstractions similar to the function abstractions they are already comfortable writing.
- *Implementing interpreters for domain-specific languages.* After designing our slideshow abstractions, we follow the "How to Design Programs" methodology to create an interpreter that runs (ie, displays) slideshow programs using simple text I/O. This plus the previous unit span four lectures using the powerpoint example, then we spend another class, a lab, and a homework repeating the exercise on another application-specific language.
- *Writing front-ends with macros.* By the time we have written the interpreter, most students are uneasy calling our abstract syntax a "language". We spend several lectures on basic (syntax-rules style) macros, covering how we can use them to hide implementation details of our abstract syntax. We also cover two examples of using macros to compile programs from concrete syntax to Scheme.

The remaining lectures are used for exams, project design review, and other minor topics (such as quoting and quasi-quoting to turn programs into data).

Three assignments are particularly pertinent to this position statement, as they illustrate how we try to encourage students to think linguistically:

1. Students are asked to find "languages" that exist in the real world outside of programming, mathematics, and music. Their answers must indicate the data, operations, and control structures within their chosen domain. They have to describe the "users" of the language and what having a language helps them achieve. Common answers to this are sporting playbooks, recipes, road signs, and theatrical stage directions. Students have the most difficulty with the "control" portion. This assignment has proven fairly effective at identifying which students understand the distinctions between operations on data and control operations in linguistic contexts.
2. Students are given Paul Graham's essay "Beating the Averages" [3] and asked to describe questions that it inspires them ask when approaching a new programming language. Graham's essay, which explains how he used Lisp as a competitive advantage in building a startup that Yahoo! eventually purchased, has proven extremely useful for helping students put functional languages in perspective. Part of his essay explains how most programmers have a preferred level of abstraction in which to work; most of us look down on lower-levels and don't see the point to higher-levels. Since I began assigning his essay, course evaluation comments about Scheme have changed: rants about the language have been replaced with more reflective comments on students' preferred programming styles.
3. As a final course project, students must design and implement a domain-specific language for a given domain. Domains in past years have been an animation scripting language, a prototype of TurboTax (with a language for tax forms), or a language for standardized tests that supports different styles of questions (multiple choice, free response, questions with hints, etc). Students are given several sample programs (sequences of animation frames, tax forms, or exams to administer). They design a language for capturing these programs and implement interpreters that run their languages. Students wishing an A-grade on the project must also use macros to implement a concrete syntax that hides details of their data structures.

### 3 Perspective

Domain-specific languages for software applications provide an excellent framework in which to introduce students to interpreters and programming languages. Students have no trouble imagining themselves being asked someday to write an application like PowerPoint or TurboTax. Whereas the idea of implementing general programming languages appears odd to many students (even at the upper level), these application languages seem natural. Unlike with meta-circular interpreter assignments, nobody gets confused about who implements the object language. The students begin to appreciate languages and interpreters as tools for writing maintainable software. When we do TurboTax, for example, students are asked to change the tax forms while leaving the underlying interpreter unedited. Overall, this approach encourages students to think of languages not only as software tools, but as tools they have the power to design themselves.

Domain-specific application languages also give students a lot of room to be creative. When we use animations as the course project, some students try to add fairly sophisticated control constructs to their work. Many students get very concerned about the aesthetic design of their concrete syntax and work hard to design macros that keep their languages clean.

Macros are surprisingly popular with the students (though many ask each year how to use them to get rid of Scheme's parentheses). They seem to enjoy the power and control of creating their own concrete syntax. We don't cover details like hygiene by default (though several students implicitly ask for this when they experiment with using macros to add loops to Scheme). Macros are also important to the course because they don't exist in the languages students have learned before. Macros aside, students could write our assignments in Java or other languages they knew on entry. By covering a feature that does not exist in the languages they know, the course asks them to think beyond Turing-complete platitudes. Many students report that the macro segment changes how they think about programming.

In the four years we have offered this class, it has proven both successful and popular. Roughly sixty-five students per year start the course (mostly freshmen). Of these, about ten will decide that the course is too fast-paced. These students can drop back to our regular introductory course (for novices) without penalty. This design-decision has been crucial: it allows us to teach to our best students, and it allows students without prior background to attempt the course if they wish. Each year, a few students successfully complete the class starting from no prior programming background. Upper-class students generally recommend the course highly, even if they disliked Scheme. They appreciate being asked to think so differently about programming.

My greatest satisfaction with the course comes from watching the students' perspectives on languages evolve over the term. The students entering the class are generally proud of their programming skills. Many are a bit haughty that college has little to teach them given the cool programs they have built before. By focusing on language implementation as a tool for software design, we respect their background while leveling the field. We ask them to function at new levels of abstraction, both through programming functionally and through designing languages. Students come to respect that languages are far richer than what they know, whether or not they care for functional programming per se. Helping them learn that lesson as freshmen, before they dive into advanced courses, gives them powerful tools for later use. Some come back in later years asking for help designing a language for a course or project. It's a perspective I doubt they would have if their only exposure to languages was through a more conventional upper-level course on the subject.

### References

- [1] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to Design Programs*. MIT Press, 2001.
- [2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [3] Paul Graham. Beating the averages. Available at <http://paulgraham.com/avg.html>, 2001.