Small Languages in an Undergraduate PL/Compiler Course

Rastislav Bodik
Computer Science Division
University of California, Berkeley
bodik@cs.berkeley.edu

Summary

I propose to revive the undergraduate PL/compiler course by making it relevant to software engineers – the bulk of our audience. My premise is that developers are not mere users of programming languages; the better developers regularly design small languages, whether they are frameworks with rich APIs, code generators, configuration and data processing languages, or full scripting languages. Our recent experience at Berkeley shows that we may indeed be able to awaken budding language designers while developing at least a better language "taste" in the remaining students. The proposed approach is to go through the development of small languages: the choice of a programming model (abstractions), design of an intuitive concrete syntax, effective static and dynamic type checking, and efficient implementation. It's hard to fit all this into a single course, of course, especially if the course is to cover enough about both design and implementation of languages. One trick may be to use the language implementation itself as a case study for small language design. For example, regular expressions deserve more attention that we have been giving them in lexical analysis: you could argue that their semantics, as defined in most languages is broken and so is their implementation. Revisiting their semantics is a case study in how to embed a language and their implementation offers an exercise in how to parallelize an algorithm that has been considered inherently sequential. Finally, we can increase the relevance to software developers by focusing on web languages, where new programming models are likely to appear: a well-designed course will prepare students for new web languages, and may encourage them to design these languages themselves. After all, PHP, javascript, Ruby and perl were all designed by language amateurs: the more we teach our undergrads, the better our future languages will be.

1 Goals

This white paper describes a PL course I have been developing with my students at University of California, Berkeley. The course, an upper-level elective, has roughly two goals:

2008 SIGPLAN Workshop on Programming Language Curriculum, May 29–30, 2008, Cambridge, Massachusetts, USA.

- Excite undergraduates about programming languages with a broad view that links languages to real-world problems rather than isolates them by stopping at distilling the essentials. Include highlights of how languages enabled *successes* of computing and also point to the many *unsolved problems*, neither of which is known to our students.

- Infuse PL technology into practicing developers. Make them thoughtful language users and encourage them to be language designers. Make them more productive and make them advance computing by judiciously increasing the levels of abstraction.

2 A Broad View

Learning how languages arise and why they fail requires that the course be equally about languages, compilers, and programming. Admittedly, the respective communities inhabit distinct planets, but I want my students to cross the border line comfortably. Prominent voices caution against polluting language education with concerns of implementation, and I agree with their arguments. Yet, as an interface between the programmer and the machine, shouldn't a language abstraction be judged in terms of how it eases the programming problem and how efficiently it can be implemented.

Obviously, this course is broad by design. It does not want to prepare a student for graduate career in PL. (Can a single course teach foundations for all the entire PL research?) Instead, I want to uncover non-trivial insights in several topics, each spanning from programming to implementation.

3 Origins of this Course

My students and I started working on a new course in 2003 when I realized that I was preparing my students for a career path that few of them follow. The compiler course was educating compiler engineers even though most our students end up as software designers or developers. Since one needs grad school experience to hack on a modern backend compiler, anyway, should the undergraduate compiler be scratched altogether? After all, even though building a Java-to-x86 compiler provides a sense of achievement, does one really need to implement 23 semantic checks to have the compiler demystified?

How to fix the problem? My first reaction was again fairly typical: compress the paleo material like parsing and make room for more modern topics such as static and dynamic analysis for bug finding and JIT compilation. While these topics were relevant to the changes in the industry, the problem was that I picked them mainly because they were on my research agenda. I did not try to come up with a set of topics optimal for a software developer.

In the end, this was perhaps my greatest revelation: Do not let your research agenda and biases be your sole guide when designing an undergraduate course. Once I realized this, I started listening to how good programmers talk about languages, what PL technology they used, and what PL technology they designed. Informal but thorough survey by my research group was surprising: programmers use languages more than we expected, but they use the technologies that we were planning to eliminate, like parsing and syntax-directed translation, or techniques that we never taught, such as compilation of high-level declarative descriptions in the context of various Web frameworks.

(This analysis above concerns a typical *compiler* course. I am not qualified to critique the typical upper-level *PL* course. I realize that the course prepares well for grad school and, if it covers models of computation, it probably helps in designing practical DSLs. Still, it would be an interesting exercise to poll master programmers as to if and how they would like to adjust the course. I want to note, however, that my unscientific sample of blogs by the former and syllabi from the latter shows that, when it comes to issues of programming languages, they don't talk about the same thing.)

4 What Should Student Learn

It helps to set learning objectives. In terms of *doing*, what should students be able to design and implement? Recent industrial practice brought many useful languages and PL-inspired tools—we want our graduates to design similar ones:

- *small interpreters*: Google Calculator
- *unstructured data extraction*: web scraping, which parses and
- *custom code generators*: server-side web templates
- *simple meta-programming*: GreaseMonkey
- *mashups*: Yahoo Pipes
- *scripting DSL:* Second Life
- *embedded DSL:* mashups in Ruby
- *source-to-source translation:* Google Web Toolkit

In terms of *understanding*, this partial list may lead to lifetime of further learning:

- *Quickly learn a new language*, by understanding essential language principles and by being able to looking for subtleties that distinguish similar languages.
- *Critique a language design*, by pointing out good and bad language design decisions.
- *Match programming domains to common models of computation*, including key flavors of dataflow and constraint programming models.
- *Understand invariants* enforced with static types and why even statically typed languages rely on dynamic checks.
- *Understand properties of abstractions* and why they influence *static analyzability* of the program.
- *Consider human aspects in language* design, including estimating cognitive load of programming abstractions.

5 The Pedagogical Approach

The philosophy of the course is to get students into the mindset of a language designer. Whenever possible, we replay the birth of a language, to illustrate the forces that shape the language, from the point of view of both programmability and implementability.

Here is a typical sequence of steps that take the students through the insights in developing a language, and through the design decisions taken from a programming problem to an implementation of a language.

- We start from a programming challenge (e.g., a domain-specific problem such as string processing, or a problem of scale, security, or performance). We then show unsuitability of a "default" language for either expressing the desired program.

- Next, we work on indentifying the model of computation suitable for the programming problem (e.g., for string processing, it may be string partitioning with finite automata matching). It is sometimes possible to guide this step by partitioning the code in the default language into logic and plumbing; the goal is then to hide the plumbing under the abstractions of the new model.

- The next step is the decision as to whether the language is an embedded language or a standalone one. This decision is usually decided on whether a custom concrete syntax is convenient and/or whether complex semantic processing is needed.

- Next comes the question of implementation; typically introduced by why the naïve scheme is too slow. This will lead to asking why a custom implementation, derived manually by students, is so much faster.

- In turn, this asks what invariants need to be proven to guarantee legality of the custom implementation, among other questions. Establishing and discovery of invariants takes us back to language design (e.g., what are the implications of allowing to dynamically remove a method in prototype language?) and also to static analysis.

This progressive development, supplemented by using the Socratic Method in the classroom, guides student to rediscover known results. Rediscovery is good: students remember better what they invent themselves and the sense of accomplishment will hopefully encourage them to develop small languages for their own programming needs.

To illustrate this approach in more detail, let me next describe just the lectures on regular expressions. Suggesting that PL courses should still pay any measurable attention to regular expressions is controversial, so let me argue why something so seemingly tedious is a good introductory vehicle for teaching small language design, from discovery of a the programming models, to its semantic foundation, embedding the DSL into a language, compilation, compositionality, and to even parallelization.

The domain programming challenge. The programming challenge that we present to students is building mashups, with GreaseMonkey, by parsing web pages and combining ad hoc web services. This string-processing problem not only stresses compositionality of regular expressions (e.g., in building a reasonably robust street address recognizer) but it also exposes to students the limitations in parsing HTML with regular expressions, motivating context free grammars.

The programming model. We first implement a string pattern matcher in imperative style (although functional would probably work equally well). This matcher operates over a list of characters, building matches character by character. Students are surprised to discover how much code they can label as *plumbing* (i.e., the often repeated mechanics code, in this case buffer management and backtracking). The code left after plumbing is labeled is the program *logic*, and its extensive pollution with plumbing motivates a need for abstraction that goes beyond the procedural. Once students see the logic fragment labeled in the imperative code, discovering the programming model in terms of a fixed set of states and transitions comes naturally, even for those who have not seen finite automata before (those identify the programming model as character-labeled directed graph, and struggle a bit with defining its semantics).

Connections to computer science foundations. At this point the instructor can quickly rehash well-known connections formal language theory, in particular non-determinism in finite automata. (Non-determinism will come handy also later, e.g., when introducing program analysis.) This connection (or lack thereof) will become more interesting shortly as we consider regular expressions as a programming model.

A good interlude might be a bit of history on how regular expressions were originally developed (before Kleene cleaned them up).

From automata to convenient concrete syntax. At this point, our programming model is a finite automaton. This poses a language design question: how will programmers conveniently enter the automaton as text? This leads to rediscovery, under instructor guidance, of regular expressions.

Compilation. Ken Thomspon's compositional algorithm for translating regular expressions is presented next, as the first compilation algorithm in the course.

DSL semantics. We now face another language design question. The regular expression matching in formal languages answer the question "does the (entire) string match the regular expression?". Our programming challenge poses a different question. Essentially, it asks "partition the string into substrings such that each matches the regular expression in its entirety". The ambiguity is clear: there could be many such partitioning, and the resolution of the ambiguity is a neglected reason why there are "flavors" of regular expressions in the wild.

Resolving this ambiguity is an instance of defining the semantics of a DSL, and regular expressions resolve it in two interesting ways: lex and family partition according to the leftmost *longest* match among all possible matches. In contrast, most existing implementations define semantics by operationally describing the backtracking taken by the * operator (the *greedy* behavior). These two definitions mostly but not always produce the same behavior. Not only they differ, the second one is not compositional for concatenation, $L(re_1).L(re_2)$! $L(re_1.re_2)$. This complicates compositional programming and prevents the use of Thomspon's algorithm. It's unclear the greedy flavor was a lucky choice; with it, we lost connection to formal regular expressions and probably also made programmability harder. What a nic example of how subtle differences in semantics matter.

Building a lexer generator. To practice implementation of DSLs, students implement a simple lexer generator out of an existing regex packages.

Language design and scanning. Perhaps surprisingly, even the lexical structure of the language influences its implementation. In JavaScript, /a/g is a regex literal, but not when this string appears in, say, c/a/g

which is an expression with two divisions. The need for such context-sensitive scanning prevents the classical compiler pattern of the lexer generating a stream of tokens. Also, it makes parallelization of scanning harder.

Parallelization. With the single-thread performance improvement disappearing, even traditionally sequential algorithms like scanning will have to be parallelized, in order to keep up with increasing data sets and program sizes. Finite automata are considered hard to parallelize, but an undergraduate student in my course showed that this domain offers an opportunity to parallelize speculative with great efficiency. The reader may disagree with me that parallel algorithms belong to a PL course; in fact, I am not certain myself. However, there is no doubt that considering the implications of a language semantics *on* parallelization should be included, since a language should permit a parallel compiler (see *Language design and scanning* above).

Note that I am not advocating that we spend half a semester on regular expressions. Two or three lectures will suffice.

Our course is not yet fully implemented but most topics taught in Fall 2007 already followed this approach to some extent. We expect the course to be fully completed by May 2009 and tested in classroom in Fall 2009.