THE DESIGN OF AN EMERGING/MULTI-PARADIGM PROGRAMMING LANGUAGES COURSE

Saverio Perugini
Department of Computer Science
University of Dayton
300 College Park
Dayton, Ohio 45469–2160
(937) 229–4079
saverio@udayton.edu

ABSTRACT

We present the design of a new special topics course, *Emerging/Multi-paradigm Languages*, on the recent trend toward more dynamic, multi-paradigm languages. To foster course adoption, we discuss the design of the course, which includes language presentations/papers and culminating, final projects/papers. The goal of this article is to inspire and facilitate course adoption.

INTRODUCTION

Emerging/Multi-paradigm Languages is a cross-listed undergraduate and graduate, three credit hours, special topics course on the recent trend in programming languages toward more dynamic, multi-paradigm languages. It was offered at the University of Dayton in the Spring 2016 and 2017 semesters. Nine students were enrolled in the Spring 2016 offering of the course: six in the undergraduate section (all seniors) and three in the graduate section. Seven students were enrolled in the Spring 2017 offering of the course. All were in the undergraduate section and all save two were seniors. Students were from a variety of majors, including computer science, computer engineering, and mathematics. The course is an

exploratory odyssey through a variety of emerging languages, including Lua, Elm, and Elixir, with a thematic focus on showcasing and creatively harnessing the niche features in each language to solve pragmatic programming problems. Topics include new concurrency models, type systems, and lazy evaluation. The student learning outcomes include:

- An understanding of fundamental (though largely reserved to functional languages until recently) language concepts which are experiencing a rebirth in *multi-paradigm languages*.
- Professional acculturation (i.e., formal presentation and manuscript preparation)

The emerging languages are contextualized through the study of classical functional programming concepts (e.g., first-class, higher-order functions) [9], which are experiencing a revival in multi-paradigm languages, in which they are perceived as less esoteric, and more accessible and practical. This motif suggests a natural syllabus of topics: use foundational languages (e.g., LISP) as a vehicle through which to study fundamental language concepts in the first third of the course, and focus on how those concepts are re-emerging in modern, multi-paradigm languages in the final two thirds of the course.

To foster adoption, we discuss our design of the course which includes student presentations/papers of the languages and culminating, final projects/papers. The goal of this article is to inspire and facilitate course adoption.

COURSE DESIGN

The course website (`http://perugini.cps.udayton.edu/teaching/courses/Spring2017/cps499/` for the Spring 2017 offering and `http://perugini.cps.udayton.edu/teaching/courses/Spring2016/cps499/` for the Spring 2016 offering) is an

integral resource for students in the course. It contains a class-by-class, course outline annotated with links to a set of the instructor's course notes (available free online for future instructors) based on the class dates on which the topics in the notes are presented. The course site also contains references to the required and recommended books (all of which are available at the University library, and many in eBook format).

Emerging/Multi-paradigm Languages is a programming intensive course, and students are required to take an active part in class. For instance, the languages studied *emerge* as the students, play a role, through informal surveys and, especially, student-selected language presentations, in deciding which languages are covered as the course organically unfolds. This course also leverages high-impact, STEM, active-learning practices.

Evaluation Instruments: Homeworks

Homeworks involve analytical and programming exercises. The programming involved in each homework requires a fair amount of critical thought and design, and approximately 100–250 lines of code. Homework assignments involve novel programming problems and puzzles that explore the use of re-emerging language concepts in application areas such as AI and numerical methods. Some assignments involve reading and writing critical analysis essays of articles in the literature.

The following is a list of assignment synopses, from the Spring 2016 offering of the course, intended to relate the content and form of assignments that might be helpful to instructors inspired to teach a similar course.

Homework #1 involves writing an introductory course essay, and building a postfix arithmetic expression evaluator and GUI in any language. Five students used Python, three used Java, two used JavaScript, two used Racket, one used Ruby, and one used C♯.

Homework #2 involves functional programming exercises in

Racket (e.g., defining a variety of sorting, searching, and metaprogramming functions).

Homework #3 is an advanced set of Racket functional programming problems, including the construction of a boolean expression evaluator.

Homework #4 involves the use of the core language concepts of scoping (e.g., static and dynamic) and binding (e.g., deep, shallow, ad-hoc) in Racket and JavaScript, including the construction of a stack object as a vector of first-class closures.

Homework #5 involves methods of affecting program control through the use of first-class continuations and continuation-passing style (CPS). Problems include classical exercises (e.g., computing Fibonacci numbers with only one, tail, recursive call) and control programming problems providing an opportunity for creativity (e.g., jumping out of and back into the run-time stack for exception handling, and building a `while` loop control construct using `call/cc` and CPS).

Homework #6 involves implementing and experimenting with a metacircular interpreter for LISP, and writing a critical reflection of [2].

Midterm entails building an interpreter in Haskell or ML for the language FORTH for control and embedded systems applications. Eight midterm projects were posted in Spring 2017; students were also given the option to propose a midterm project.

Homework #7 covers concepts intended to promote effective system modularity, including type inference and strong typing, type systems, currying, and higher-order functions (HOFs). Students defined a `string2integer` function in one line of code, and solved a non-trivial problem of their choice by creatively using the aforementioned building blocks. Students also read and answered critical-analysis questions on [8], which creatively uses currying to dynamically create shortcuts to frequently-performed tasks in application software.

Homework #8 covers lazy evaluation. Students built a lazy iterator object from first principles and used it in a variety of problems. Students also implemented a host of numerical methods (e.g., numerical differentiation) using lazy evaluation and HOFs, inspired by [3]. Students wrote a critical analysis of [10], which posits the functional paradigm (especially in Erlang) as an approach to the multicore problem, and helped transition students to (re-)emerging models of concurrent programming—the *Actor Model of Concurrency* (in Elixir) and *Communicating Sequential Processes* (CSP; in Go)—which use non-traditional approaches to thread communication and synchronization.

Homework #9 involves a suite of concurrent programming exercises using the CSP model of concurrent programming in Go. Problems included a host of variations on managing n threads (called *goroutines* in Go) to cooperatively perform some task.

Homework #10 involves solving the classical *Sleeping-Barber* problem from operating systems using the *Actor* model in Elixir.

Language Presentations and Papers

The Spring 2017 offering of the course involved student presentations of the emerging languages after the foundational material was explored. The approximately five week student presentations of languages transferred ownership of both the instruction and learning to the students. Each student presented a language from a list of ten emerging/multi-paradigm languages. The languages presented were Elixir, Factor, Lua, Io, Julia, CLIPS, and Elm.

Each student presented one language across two consecutive 50-minute class periods. Presentations involved creative demos and programming pearls to showcase the particular language. As part of this component of the course, students were required to write either a two-page paper on the language, in the style of [11], or develop a one-page language quick reference sheet (akin to `https://media.pragprog.com/titles/elixir/ElixirCheat.pdf`).

Students were also required to develop a webpage containing technical details, syntactic and semantic details, and example programs in the language, which were linked from the course webpage. They also developed a set of representative programming exercises to help their fellow students reflect on the practical applications of the language (and included them as a section in their two-page language paper). The video of all presentations, save for the first on Elixir, was recorded and made available on YouTube. All student language video presentations, papers, and HTML notes are available at `http://perugini.cps.udayton.edu/teaching/courses/Spring2017/cps499/languages.html`.
Source code from these presentations is available as a Git repository in BitBucket at `https://bitbucket.org/sperugin/emerging-languages-spring-2017`.

A main idea behind the language presentations is to showcase several emerging/multi-paradigm languages to provide a deeper context for students from which to pursue a final project. The requirements and evaluation criteria for the final presentation and paper (Table 1b) was identical to that of the emerging/multi-paradigm language presentation and paper (Table 1a). Thus, another natural and desirable effect of the language presentation/paper component of the course is that it provides graded preparation for the final project presentation/paper.

Final, Culminating Projects

The entire first two thirds of the course is structured to prepare students for the final, culminating project experience, the goals of which is to inspire students to demonstrate the concepts learned by putting (an integration of) them into practice.

Approximately one month before the end of the semester, the instructor posted a list of ideas for possible course projects (e.g., building a game in Lua, or an application of AI using lazy evaluation to mitigate the size of the search space). Project ideas were intentionally vague and open-ended to provide students ample scope for

Table 1: Evaluation criteria and point distribution for (left) language papers/presentations and (right) final projects/papers from Spring 2017.

(a) Language presentation and papers.

Component	Points	Percentage
Detailed Presentation Evaluation Criteria: level of preparation, clarity, creativity, and originality **(2-class, in-class) Language Presentation**	50	30%
Detailed HTML Language Notes Evaluation Criteria: clarity, creativity, originality, grammar; cleanly working HTML code; adherence to provided template and style guide; quality of tables/figures **HTML Language Notes**	50	30%
Detailed Language Synopsis or Quick Reference Sheet Evaluation Criteria: content, structure, clarity, grammar; cleanly working LaTeX code; adherence to ACM SIG style; quality of tables/figures; & citations/bibliography (BibTeX) **2-page Language Synopsis or Quick Reference Sheet**	50	30%
Detailed Language Programming Exercise Evaluation Criteria: functionality, creativity, depth, & documentation (e.g., comments) **Programming Exercises**	16	10%
Total Language Presentation Points:	166	100%

(b) Final project (paper, presentation, and system).

Component	Points	Percentage
Abstract (optional, but highly recommended)		
Paper draft (optional, but highly recommended)		
Detailed Final Paper Evaluation Criteria: content, structure, clarity, grammar, cleanly working LaTeX code, adherence to ACM SIG style, quality of tables/figures, & citations/bibliography (BibTeX) **Final project term paper**	111	33.33%
Detailed Presentation Evaluation Criteria: level of preparation, clarity, creativity, and originality **Presentation**	111	33.33%
Detailed System Evaluation Criteria: functionality, creativity, depth, & documentation (e.g., comments) **Source code/running system**	111	33.33%
Total Final Project Points:	333	100%

individual critical thought, design, and creativity. The only pseudo-requirement of the project was that it creatively applies the concepts and building blocks studied in the course in a practical application in an emerging/multi-paradigm language. Students were also welcome to propose their own project, of which seven did. Students were given one month to complete the project, during which time no other course work, graded or otherwise, was assigned. Final projects involved three components: a working system, a formal paper discussing it, and an in-class presentation to classmates and the instructor during the final exam period (see Table 1b).

Final papers were required to be three pages long and, in keeping with the theme of the course, typeset in LaTeX (and BibTeX), a document-preparation language, using the ACM SIG Proceedings LaTeX Template. Each paper was required to contain one original figure and one original table, and a minimum of three references to published ACM or IEEE papers. Students were required to use the LaTeX package `lstlisting` to typeset any (snippets of) source code included in a paper, which did not count toward the total page count. Students were directed to *Overleaf*,

a synchronized, split screen (source⇔PDF) LaTeX IDE, which can be used through the cloud. Students were advised to follow detailed recommendations for writing a formal paper compiled by the instructor, and available at http://academic.udayton.edu/SaverioPerugini/documents/advice.html. They were also provided a detailed list of writing conventions to follow. Students submitted an abstract two weeks before the paper draft deadline, which was approximately one week before the deadline for the final paper and presentation, and the delivery of the final source code/running system. The final paper and presentation experience introduced students to the process of professional dissemination of their work.

A website showcasing (selected) completed final course projects from Spring 2016 is available at http://perugini.cps.udayton.edu/teaching/courses/Spring2016/cps499/projects/selectedprojects.html. The site contains a project abstract, and links to the final project paper and presentation (both in PDF format), for each project. A similar site is available for the Spring 2017 offering at http://perugini.cps.udayton.edu/teaching/courses/Spring2017/cps499/projects.html. Videos of the final presentations from Spring 2017 are available on YouTube at https://www.youtube.com/watch?v=NtPTRLdz2rE&t=208s and https://www.youtube.com/watch?v=MtgbeLO6ZM4&t=224s. These projects enhance and extend the style of projects proposed in [5].

CONCLUSION

The programming language landscape is ever-evolving to meet the demands of modern runtime environments and hardware platforms, and new problem domains. As a result, languages such as Python, C♯, and C++ now include support for many of the re-emerging functional building blocks and dynamic bindings covered in this course. The *Emerging/Multi-paradigm Languages* course is a response to this phenomenon. The course was generally well re-

ceived by the students. One student provided the following anonymous comment on a course survey:

I personally like this setup of the course with us basically taking over for the second half better than the original setup. Really made us integrate all the topics we learned in order to synthesize all the information of the languages. Was really effective at helping us understand how to choose a language for development regardless of where we head in the future.

Multiple approaches, which vary in objective and perspective, have been used for teaching a general course on programming languages [1, 4, 7]. Lewis et al. [6] explored the use of *uncommon* languages (e.g., OCaml, Grace, Jigsaw, Processing, and Scala) for CS1, but the focus is on purity, simplicity, and ease of the languages with respect to pedagogy, and Scala is the only *emerging* language in the set for non-educational, real-world applications. A host of other emerging languages can be substituted for those explored in the two offerings of the course discussed here (e.g., TypeScript, Hack, Clojure, Scala).

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant Numbers 1712406 and 1712404. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. We thank Norman Bashias in the Department of Computer Science at the University of Dayton for providing comments on a draft of this paper.

REFERENCES

[1] Adams, E., Baldwin, D., Bishop, J., English, J., Lawhead, P., and Stevenson, D. Approaches to teaching the programming languages course: A potpourri. In *Proceedings of the 11th Annual*

SIGCSE Conference on Innovation and Technology in Computer Science Education, pages 299–300, New York, NY, 2006. ACM Press.

[2] Graham, P. *Beating the Averages*. O'Reilly, Beijing, 2004. Available: `http://www.paulgraham.com/avg.html` [Last accessed: 19 July 2018].

[3] Hughes, J. Why functional programming matters. *The Computer Journal*, 32(2):98–107, 1989.

[4] Krishnamurthi, S. Teaching programming languages in a post-Linnaean age. *ACM SIGPLAN Notices*, 43(11):81–83, 2008.

[5] Kumar, A. Projects in the programming languages course. In *Proceedings of the 10^{th} Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, page 395, New York, NY, 2005. ACM Press.

[6] Lewis, M., Blank, D., Bruce, K., and Osera, P.-M. Uncommon teaching languages. In *Proceedings of the 47^{th} ACM Technical Symposium on Computer Science Education (SIGCSE)*, pages 492–493, New York, NY, 2016. ACM Press.

[7] Pombrio, J., Krishnamurthi, S., and Fisler, K. Teaching programming languages by experimental and adversarial thinking. In Lerner, B., Bodík, R., and Krishnamurthi, S., editors, *Proceedings of the 2^{nd} Summit on Advances in Programming Languages (SNAPL)*, pages 13:1–13:9, 2017.

[8] Quan, D., Huynh, D., Karger, D., and Miller, R. User interface continuations. In *Proceedings of the 16^{th} Annual ACM Symposium on User Interface Software and Technology (UIST)*, pages 145–148, New York, NY, 2003. ACM Press.

[9] Savage, N. Using functions for easier programming. *Communications of the ACM*, 61(5):29–30, 2018.

[10] Swaine, M. It's time to get good at functional programming: Is it finally functional programming's turn? *Dr. Dobb's Journal*, 34(1):14–16, 2009.

[11] Wexelblat, R., editor. *ACM SIGPLAN Notices*, volume 28. ACM Press, New York, NY, March 1993.