JFP **29**, e13, 13 pages, 2019. © Cambridge University Press 2019
doi:10.1017/S095679681900011X

Emerging languages: An alternative approach to teaching programming languages

SAVERIO PERUGINI
University of Dayton, 45469–2160 USA
(*e-mail:* saverio@udayton.edu)

Abstract

We challenge the idea that a course intended to convey principles of languages should be structured according to those principles, and present an alternate approach to teaching a programming language course. The approach involves teaching emerging programming languages. This approach results in a variety of course desiderata including scope for instructor customization; alignment with current trends in language evolution, practice, and research; and congruence with industrial needs. We discuss the rationale for, the course mechanics supporting, and the consequences of this approach.

1 Introduction

While the learning outcomes of a course in programming languages are well established (Adams *et al.*, 2006; Pombrio *et al.*, 2017), the most effective approach to teaching a language course is not a settled matter. There are a myriad of interpreter-based (Friedman *et al.*, 2001; Queinnec, 2003; Krishnamurthi, 2012), survey-based (Louden, 2002; Scott, 2009; Sebesta, 2015), and other approaches (Kumar, 2005; Adams *et al.*, 2006; Fraser *et al.*, 2015; Lee, 2015; Lewis *et al.*, 2016). Over the past 14 years, we have tried various approaches toward teaching programming languages, including the two predominant approaches—the comparative/survey approach and the interpreter approach—which both involve challenges. In Spring 2016, we sought to teach programming languages from a different perspective. The approach involves using emerging programming languages (e.g., Lua, Elixir, and Python) as a conduit through which students incidentally bump into concepts of languages, the implementation options available for them, and their compelling consequences in programs. The focus is on an exploration of the explosion of languages, which share the adoption of functional features to some extent, that has happened in the last 20 years. The central thesis of this article is that this alternative approach results in a variety of course desiderata (see Section 4), including scope for instructor customization; alignment with current trends in language evolution, practice, and research; and congruence with industrial needs. We discuss the rationale for, the course mechanics supporting, and the consequences of this approach.

2 Rationale for the emerging languages approach

We challenge the idea that a course intended to convey concepts of languages should, therefore, be structured according to those concepts. The rationale for our approach is based on two simple ideas:

1. Recognizing that students think in terms of *languages* not *concepts*.
2. Given that perspective, use course mechanics that leverage how they learn *languages* to teach them *concepts*.

Let us unpack both ideas.

2.1 The student language-centric perspective

The approach is based on the belief that a typical student thinks in terms of *languages* not *concepts* and, moreover, places more importance on languages than concepts. This student perspective was prevalent in qualitative responses collected in a span of five semesters in a longitudinal study we conducted to investigate the effectiveness of the use of Go and Elixir for teaching the concepts of concurrency and synchronization. In over 550 anonymous student responses collected to a variety of questions, students used terms referring to languages when addressing concepts (i.e., they referred to the languages Go and Elixir rather than CSP and the Actor model, respectively). For instance, "Elixir improved my learning because it forced me to draw design diagrams." The diagrams of actors and messages passed between them are an artifact of the Actor model, not Elixir.

Learning *concepts* of programming languages sounds dull and academic; learning *languages*, especially new, emerging ones, and how they apply to domains in which students have passion, like game programming, Internet of Things, or web frameworks, sounds like fun. Most students with an interest in gaming are motivated to learn Lua—which also has an attractive side effect of being an appropriate language for distilling concepts of programming languages. So why not take advantage of that juxtaposition of circumstances. If we tell students we are going to learn Lua, Python, and Ruby, that is something they can get excited about (and to which they can relate given the pervasive presence of these languages in online fora). Let's get students excited about what they are studying and then most effectively harness that motivating spirit to meet the learning outcomes of the course—exploring concepts, implementation options, and the implication of those options on programming.

2.2 The student relevance-centric perspective

Students typically only want to invest time and effort in studying topics that they perceive as relevant and useful to the world outside of academia (Pintrich, 2003). Since "[s]tudents often learn a particular programming language to get themselves jobs" (Guzdial & Landau, 2018), and since more code is written in, e.g., Java and C++ than LISP (Ray *et al.*, 2017), students are typically not going to get excited about required concepts of languages course whose description indicates the use LISP as an implementation language (in either the survey- or interpreter-oriented approach), which they perceive as not being used in industry, as not building their résumé, and as not helping them get a job.

Students are, however, motivated to learn new, hot languages and novel emerging technologies. Students perceive emerging languages like Python and Ruby as less esoteric, more accessible, and more practically applicable to real-world problems than languages like LISP and ML, which they perceive as archaic and arcane. Why? We believe the answer lies in the idea that students are products of the online ecosystem in which they function. Some students probe new languages and technologies on their own (McCartney *et al.*, 2016) by taking online courses, watching YouTube videos, reading Stack Overflow and other similar fora, and perusing enthusiast programming blogs. The increased use of languages incorporating functional programming features in recent times (Savage, 2018), especially in web frameworks, has caused them to experience increased presence in these online fora, and this influences how students perceive these emerging languages. For instance, our students reported an awareness of Elixir through the Phoenix web framework (`http://phoenixframework.org`). Moreover, Python ranked first on average as the most popular programming language in 2018 (of 47 languages across a variety of metrics from multiple sources) (Cass, 2018). Can't we teach students first-class and higher-order functions and closures in Python rather than LISP? (We are not advocating against a principles-based approach, but rather proposing of the use of emerging languages.) Ultimately, nobody cares if a student can *program* LISP; rather they care if they can *think* like a LISP programmer (e.g., higher-order abstractions and metaprogramming), in a variety of other languages. Paul Graham notes that "Python copies even features that many Lisp hackers consider to be mistakes. You could translate simple Lisp programs into Python line for line" (Graham, 2004). We should harness this process of (programming language) acculturation/incubation in which students, perhaps unconsciously, participate.[1] Students view their academic studies in computer science and their personal, continuous growth in knowledge of computing technologies as disjoint. Let's blur these boundaries.

3 Course framework: Support for the approach

The course involved three components: a brief introduction to programming language concepts and constructs; student presentations of emerging languages; and final, culminating projects to help connect the languages back to the concepts.

3.1 Establishing language concepts as a basis

Most average undergraduate students have both little formal exposure or practical experience with language principles or the tenets of functional programming.[2] Thus, to lay a foundation from which the subsequent study of emerging languages could be deconstructed into their constituent conceptual language principles, we begin the course with a brief, approximately 4-week, systematic study of classical programming language concepts[3]. We presented first-class and higher-order functions, closures, pattern matching,

[1] This rationale for our approach is grounded in *constructivist* theories/approaches to learning (Papert, 1980; Hamer *et al.*, 2008). "Students construct mental models to conceptualize the world around them and use information they already know to acquire more knowledge" (Alesandrini & Larson, 2002).

[2] None of the students in our two offerings of the course had ever had any formal study of these topics.

[3] An alternative approach is to make use of a highly successful MOOC (Miller *et al.*, 2014).

type systems, lazy evaluation, and continuations, among other concepts through traditional lectures and in-class demonstrations. We followed two guiding principles while presenting these concepts: (i) avoid devoting more than a few weeks of class time to these topics; and (ii) use emerging languages—but not those to be used in module two of the course—as much as possible to present these concepts. For instance, we used JavaScript for first-class and high-order functions, Python for closures and lazy evaluation (e.g., list/generator comprehensions), Perl for scoping options, and Ruby for first-class continuations. We also used Haskell for pattern matching, type systems, and lazy evaluation; and Racket for some fundamental functional programming. Neither the languages used to demonstrate each concept, which varied from concept to concept, nor the emerging languages (e.g., Ruby and Python) used to demonstrate these fundamental concepts were used in module two of the course (e.g., Lua and Elixir). The required textbook used to guide students through this introductory material features multiple emerging languages, especially in the coverage of core language concepts through Python (Perugini, 2018b). Students were assigned graded, supportive homework, involving both conceptual and programming exercises, during this module of the course to evaluate their understanding of these concepts; see (Perugini, 2018a) for the details.

3.2 Student presentations of the emerging languages

With a conceptual foundation established, we embarked on a series of student presentations of emerging languages over the next 7–10 weeks. Each student was required to present a language across two consecutive full (75 min) class periods. A motivation for the "language presentations" component of the course was to emulate (within a course context) the process by which students probe and learn new languages and technologies on their own. The instructor recommended a host of emerging languages to students from which to choose; they also had the freedom to propose a language to present and multiple did. Thus, students also play a role in how the course content organically evolved (Pintrich, 2003; Hamer *et al.*, 2008). The languages involved include Lua, Elixir, Elm, Go, and Julia.[4] The series of the *Seven ...* books from the *Pragmatic Bookshelf* (Tate, 2010; Butcher, 2014; Tate *et al.*, 2014) were recommended, and made available to students through the University library, as a guide through this component of the course. Those books, in particular, help connect the emerging languages covered therein to both concepts and the evolution of programming languages catalyzed by recent developments in software development (i.e., application domains, hardware platforms, and development processes)—an important theme of which we took care to incorporate into the course.

This central component of the course involved multiple mechanisms by which to facilitate this simulation:

- students teaching other students (Pintrich, 2003; Hamer *et al.*, 2008)—an aspect of *active learning* (Freeman *et al.*, 2014)—through both the presentations themselves and the use of creative programming pearls therein;

[4] Of the emerging languages covered in the course, four appear in a recent list of "9 New Programming Languages to Learn in 2019" (Kumar, 2019).

- the creation of brief (appropriately three page) language synopses, akin to those in the (Wexelblat, 1993) collection, e.g., (Budd, 1993), or quick reference sheets (e.g., `https://tinyurl.com/LuaQuickReference`) akin to `https://tinyurl.com/ElixirQuickReference`;
- the development of a set of (approximately three) supportive programming exercises to help fellow classmates assimilate the predominant language concepts/idiosyncrasies showcased through the particular presented language;
- the creation of outline-style webpages, involving snippets of code; and
- video recordings of all in-class language presentations which were made available to students and the larger community through YouTube.

These mechanisms also help students overcome the difficulties of learning the idiosyncrasies of programming in each of these languages. A table of links, for each language presented, to the outline-style online notes; language summary or quick reference sheet; the programming exercises developed; and the YouTube video of the presentation is available at `https://tinyurl.com/LangPresMaterials`.
Aside from attempting to recreate the environment in which students learn languages, this component of the course offered ancillary advantages: (i) the creation and archival of the materials developed, and the collection of them across all students, serve as a tangible and permanent artifact of language resources to which students can later refer in subsequent courses as well as on the job in the industry. "[L]earning is most effective when part of an activity the learner experiences as constructing a meaningful product" (Papert, 1987); and (ii) it transfers ownership of course and learning, including the material and structure/form through which that material is presented, to the students (Pintrich, 2003; Hamer *et al.*, 2008);

3.3 Closing the loop: Connecting languages back to concepts

Despite the moniker for the approach, the goal is not to teach students "emerging languages" and, thus, the course must not involve an isolated, rote, investigation of a sequence of emerging languages. Rather the goal is to teach students *concepts* of languages and the *implications* of a variety of implementation options for these concepts as an intended, though unadvertised, side effect of covering these emerging languages under the guise of a survey course of a variety of new, hot languages. The introductory module of the course was indispensable in helping to establish a context and framework from which students could deconstruct, reason about, and compare the emerging languages discussed in the main (second) module of the course. The final, culminating project experience helped close the loop from this study of emerging languages back to the essential concepts studied in the first module. The final project also gives students an opportunity to apply and demonstrate mastery of the concepts covered in the first (introductory) and reinforced/distilled in the second (language presentations) modules of the course. Students used their intuition to independently discern how and where to harness and creatively integrate a subset of the language concepts covered to craft a solution to a practical computing problem. Posted project ideas were intentionally vague to provide students ample scope for individual critical thought, design, and creativity (e.g., building a game in Lua, developing a

Table 1. *Frequency of use of language concepts in final projects (Spring 2016 and 2017). Legend:* $s_1 \ldots s_{15}$ *= anonymized student identifiers*

Concepts / Projects	**Lua**				**Python**			**Julia**		**Elm**	**Haskell**	**Racket**	**Elixir**	**Go**	**PROLOG**	**S**	**S**	**Total**
	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	s_{11}	s_{12}	s_{13}	s_{14}	s_{15}	**’16**	**’17**	**Freq.**
First-class functions	✓	✓		✓	✓	✓		✓	✓	✓	✓	✓				4	6	10
Higher-order functions	✓	✓			✓	✓			✓	✓	✓	✓				3	5	8
λ expressions	✓			✓	✓	✓				✓		✓		✓		4	3	7
First-class closures			✓	✓	✓	✓										2	2	4
Pattern matching										✓	✓		✓		✓	3	1	4
Type systems					✓	✓				✓	✓					2	2	4
Strong typing										✓	✓			✓		2	1	3
Currying										✓	✓					1	1	2
First-class continuations							✓					✓				2	0	2
List/generator comprehensions						✓					✓					2	0	2
Tail recursion					✓						✓					1	1	2
Message passing						✓							✓			2	0	2
Actor model of concurrency						✓							✓			2	0	2
Homoiconic												✓			✓	2	0	2
Lazy evaluation											✓					1	0	1
Coroutines								✓								0	1	1
Channels														✓		1	0	1
Communicating Sequential Processes														✓		1	0	1
Partial function application								✓								0	1	1
λ-expr-repr. of data struct.					✓											0	1	1
Continuation-passing style																0	0	0
Frequency	3	2	1	3	7	8	1	3	2	7	10	5	3	4	2	36	25	61

functional image-manipulation and -processing library in Julia, or constructing a human–computer dialog manager in Elixir). Students were also welcome to propose their own project, of which seven did. Students were encouraged to pursue solutions which involved an integration of multiple languages and/or libraries (e.g., the Pykka library[5] for using the Actor model in Python, or use of the LOVE2D (`https://love2d.org/`) library in Lua). Students were given 1 month to complete the project, during which time no other course work, graded or otherwise, was assigned. Final projects involved a software system, a formal paper discussing it, and an in-class presentation.

Since the course offerings using this alternative approach did not involve a formal research experiment, it is challenging to ascertain the merit of this approach in helping students understand core language concepts. We can, however, offer some informal evidence. Table 1 presents the observed frequency of use of language concepts in the source code of the final projects produced by students. (The Spring 2016 and 2017 offerings involved nine and seven students, respectively, including one outlier.) Students self-reported the concepts that they used and the instructor conducted a manual analysis of the source code to both verify the self-reported concepts were actually used and self-reported accurately (i.e., identifying false positives), and identify any concepts used that were not self-reported (i.e., identifying false negatives). While the observation of the application and integration of concepts of programming languages in final projects cannot, on its own, serve as formal evidence that students understand those concepts well, it does provide anecdotal evidence to that end, especially since many projects were of high quality, some of which were published (Prince & Perugini, 2018; Perugini & Watkin, 2018). Moreover, student understanding of the concepts was evaluated through programming homeworks/exercises and through the evaluation criteria of in-class language presentations and final project presentations (Perugini, 2018a).[6] To triangulate these informal results, we report the scores

[5] The Pykka library (`https://www.pykka.org`) is based on the design of the Akka library (`https://akka.io`) for actors in Java and Scala.

[6] The requirements and evaluation criteria for the presentation/paper for the emerging languages the final project components of the course are available at `https://tinyurl.com/LangPresEvalRequirements` and `https://tinyurl.com/FinalProjectEvalRequirements`, respectively.

of student midterm projects in Spring 2017 which involved the use of multiple language concepts that we want students to understand. Each student pursued a midterm project among eight choices offered by the instructor.[7] Students completed the following midterm projects (number in parentheses indicates frequency): FORTH Interpreter in Haskell (3); Applicative-order Y-combinator in JavaScript (1); Expression evaluator in Racket (1); and Multi-paradigm Language Interpreter in Python (1). Midterm projects were evaluated for the use of tenets of functional programming (e.g., higher-order functions) as well as correctness against test cases. Five students scored 100% and one student scored 98%. A student provided the following anonymous comment on a course survey regarding connecting the languages back to the concepts:

> *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.*

4 Consequences of the approach: Resulting course desiderata

The following are desirable consequences and properties of this approach:

- **Customizable.** Instructors can use a variety of different emerging languages across course offerings. A host of other languages can be substituted for those explored in the two offerings of the course discussed here (e.g., TypeScript, Hack, Clojure, Scala, R, Qt, Kotlin, miniKanren, F$^\sharp$, and Bosque). Instructors can also use languages of their or students' preference for the foundation material.
- **Moves students away from a Java-centric worldview.** By covering a wide spectrum of languages in a relatively short window of time, this approach abruptly/radically compels students out of their safe, insulated Java bubble and dispels/shatters a Java-only perspective of problem solving and the programming landscape.
- **Fosters/promotes the idea that programmers should use a language appropriate for the problem domain.** The use of these types of languages—those that are highly targeted to certain domains and/or those supporting metaprogramming—gets students acclimated to the idea of expressing solutions in languages most closely aligned with the application domain (e.g., Julia for scientific programming) (Felleisen *et al.*, 2018). (Note, the syntactic idiosyncrasy of emerging languages can be viewed as a survival mechanism; languages both survive and succeed through their appropriateness and effectiveness in a niche application domain—and that notion of identity is key to that success. This perspective challenges the common course theme that most languages are far more similar than they are different, with a similar underlying semantics for the most common language features. This under-explored notion of language identity also supports the rationale for our approach.)
- **Promotes academic integrity.** Rotating or using different languages across offerings reduces the student incentive to resort to plagiarism (e.g., reusing/rehashing

[7] Students also had the option of proposing a project.

language materials students developed in prior offerings) without demanding an inordinate excess of work from the instructor (since students are learning through their construction of materials).

- **Does not compromise learning outcomes.** Students successfully applied and integrated fundamental language concepts in their course projects (see Table 1). A formal research study is necessary to verify use implies assimilation in this context. Moreover, 12 of the 17 topics in the *ACM/IEEE Curriculum Standards for Programming Languages in Undergraduate CS Degree Programs* (The Joint Task Force on Computing Curricula: Association for Computing Machinery (ACM) and IEEE Computer Society, December 20) [pp. 155–166] are covered by this course.
- **No substantial compromise in purity necessary.** This approach does not require compromise for language purists. Students who really wanted to use pure languages like Haskell and PROLOG—there are usually always a handful in a course—were accommodated (see Table 1).
- **Aligns well with current trends in language evolution, practice, and research**, especially the trend toward more languages with dynamic bindings and functional programming features and toward research supporting language-oriented programming and toolchains for building embedded domain-specific languages (Felleisen *et al.*, 2018). The increased momentum of this evolution in languages in these veins has exceeded the pace with which computer science courses at the university level can adapt (Krishnamurthi, 2008).
- **Aligns well with current industrial needs**, e.g., MapReduce (Dean & Ghemawat, 2004). Sustaining undergraduate student engagement with and investment in learning the material requires a concerted effort to illustrate to them that what they are learning is directly and practically applicable and will be useful beyond the end of the course. Students reported the following:

  > *[F]or my post-grad[uation] job I will be working a lot with large datasets. . . . [L]azy evaluation was very interesting because it allows you to get results back from recursive functions right away and work with infinite data structures.*

  > *I've had a lot of opportunities to use concepts from Emerging Languages in my post-graduation full-time job. I needed to configure callbacks for Bluetooth pairing. The callbacks require a certain number of arguments, however I needed to be able to pass into the callback properties on the device that I am attempting to pair to in order to be able to set the device as a trusted device. I utilize partial evaluation to bind these properties to the function because I knew these properties when I created the callback, but not when the callback was being called (different scopes).*

- **Fosters/promotes professional preparation.** The language presentations/paper component of the course and, especially, the final paper/presentation experience introduced students to the process of professional dissemination of their work. Some papers from the final projects have been published (Prince & Perugini, 2018; Perugini & Watkin, 2018); one of the language papers has been accepted for publication (Arnold & Perugini, 2019).
- **Fosters/promotes academic preparation.** This approach prepares students for future core and special topics courses (e.g., OS, AI, machine learning, game programming, numerical methods, and so on). For instance, a student who subsequently took the OS course reported the following in an anonymous survey:

The [A]ctor model is able to process the details that I wanted without all the busy work of just setting up channels and such. Elixir is able to make that syntactically clear, especially since I have taken [E]merging [Programming] [L]anguages [course].

5 Scaling the approach

A limitation of this approach, due to its constructionist and active-learning aspects, is that it does not scale well to courses with more than approximately 25 students. However, instructors of sections of large class sizes can mitigate the challenges in scaling the approach through some well-studied techniques (NCAT, 2014; CRLT, 2016). (The Center for Research on Learning and Teaching at the University of Michigan maintains both theoretical and practical material on teaching strategies for large classes (CRLT, 2016).) For instance, instructors might consider the use of team work for both the language presentations and final culminating project components of the course to mitigate the effects of this challenge. Ideally the course should use between five to ten programming languages to distill the concepts. Though used in a different context, we can generally use *Little's Law* ($L = \lambda \times W$) to informally reason about number of languages/teams (W) and number of student members per team (λ) as a function of class size (L). In a class of 30 students and using seven languages as an average yield seven teams of approximately four to five students. Scaling the approach to classes of 60 students requires either approximately eight to nine students per group, which is typically unreasonable, or increasing the number of languages/teams to 14, which is feasible in a 14-week semester. Scaling the approach to classes of 100 students requires more creativity. Using 10 languages and having two distinct groups present one language lead to teams of five students. Instructors can require each of the two teams presenting the same language to focus on distinct aspects of the language (e.g., concurrency in Elixir versus metaprogramming in Elixir) or present the same aspects so that the other students experience the presentation of language concepts from two distinct perspectives.

Additional techniques (e.g., providing clear specifications for in-class activities and group projects/presentations, maintaining high expectations, and providing prompt feedback) are discussed in (NCAT, 2014). Lastly, large universities with large class sizes will almost certainly have teaching assistants available to help with course management and grading. Utilizing and managing those TAs well are important in large sections (NCAT, 2014). Nevertheless, sustaining this approach in classes with a large number of students is an ongoing challenge.

6 Other challenges and lessons learned

Also, the approach can raise concerns that students are only gaining substantial experience programming one or two of the emerging languages—those chosen for their final project. While it is helpful for students to gain substantial programming experiences with multiple languages, that goal is secondary, in our opinion, to gaining experience with multiple concepts of programming languages. This issue can be addressed by assigning more weight to the integration of multiple languages/libraries aspect of the final project. In addition, as mentioned above, instructors might consider assigning the programming exercises students develop as part of their language presentations to the residual students for a grade. Moreover, sustaining student engagement can also be an ongoing challenge. Specifically, once students choose a language to use for their final project, they may lose interest in the other student language presentations. Instructors may want to explore incentive mechanisms such as, again, assigning and grading the programming exercises students develop. The approach can also present challenges to sustaining students' interest if, due to language rotation across offerings, instructors are not covering the languages that students really want to learn (e.g., Lua). We certainly do not want this approach to devolve into a special topics course on a variety of obscure research languages. There is nothing inherent in the approach that requires either a different set of languages to be used across course offerings or a wide spectrum of languages to be used; two to three may suffice. Lastly, some students may find language acclamation challenging because after (a maximum of) two class periods we move onto a new language. It is critical for and incumbent upon the instructor to schedule and orchestrate the sequence of language presentations so to foster connections between and themes across the languages.

Beyond these challenges in course mechanics, student inexperience with implementation details of languages, especially hands-on experience, as students in an interpreter-oriented course acquire (Friedman *et al.*, 2001), is a limitation of our approach. However, it is reasonable to expect that students completing a course using the approach explored here would be better-equipped to identify and characterize existing languages in terms of their design choices, since this is explicitly done in the course.

7 Conclusion

The alternative approach presented here is based on the idea that we should teach programming languages in a mode that meshes well with and leverages both how students think about languages (e.g., putting language before concepts) and how students learn languages on their own (e.g., through online fora and YouTube videos).

Students were given a foundation and vocabulary (in module one) from which to deconstruct and compare the emerging languages presented in the second module of the course (of which they had no prior experience), and make use of in module three. The value added here is analysis of concepts (module one), deconstruction of languages to probe concepts (module two), and reconstruction/composition/application of concepts (module three). The qualitative data collected support the effectiveness of the course design. Moreover, the active learning and constructivist techniques used in modules two and three enhanced the approach by supporting "learning as a reconstruction rather than as a transmission of knowledge" (Papert, 1987). Thus, there is a synthesis present here that is less likely to emerge in a standard survey course, and the active learning and constructivist techniques fostered, supported, and enhanced that synthesis.

A showcase of (selected) completed final course projects is available at `https://tinyurl.com/FinalProjectsSpring2016` and https://tinyurl.com/FinalProjectsSpring2017. Videos of the final presentations from the Spring 2017 offering are available on YouTube at `https://www.youtube.com/watch?v=NtPTRLdz2rE&t=208s` and `https://www.youtube.com/watch?v=MtgbeLO6ZM4&t=224s`. Lastly, a weekly syllabus and course outline involving pedagogical goals, explanatory lecture titles, and supportive homework can be found at:

http://perugini.cps.udayton.edu/teaching/courses/Spring2017/cps499/

The idea that a course intended to convey principles of languages need not be structured according to those principles has been communicated in prior work. Budd has published a book with discussion of multi-paradigm approaches to programming languages (Budd, 1995). We are optimistic that this article will generate discussion and reignite interest in such related approaches.

Acknowledgments

This material is based upon the 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 Krishnaprasad Thirunarayan in the Department of Computer Science and Engineering at Wright State University for providing comments on a draft of this paper. We thank David J. Wright in the Learning Teaching Center at the University of Dayton for identifying supportive references. We also thank the anonymous referees for their comments and recommendations which helped improve the article.

References

Adams, E., Baldwin, D., Bishop, J., English, J., Lawhead, P., & Stevenson, D. (2006). 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 (ITiCSE). New York, NY: ACM, pp. 299–300.

Alesandrini, K. & Larson, L. (2002) Teachers bridge to constructivism. *Clearing House* **75**(3), 119–121.

Arnold, Z. L. & Perugini, S. (To appear) An introduction to concatenative programming in Factor. *J. Comput. Sci. Coll.* **35**.

Budd, T. (1993) A brief introduction to Smalltalk. *ACM SIGPLAN Not.* **28**(3), 367–368.

Budd, T. (1995) *Multiparadigm Programming in Leda*. Boston, MA, USA: Addison-Wesley.

Butcher, P. (2014) *Seven Concurrency Models in Seven Weeks: When Threads Unravel*. Dallas, TX: Pragmatic Bookshelf.

Cass, S. (2018) The 2018 top programming languages. *IEEE Spectrum*. Accessed June 19, 2019. Available at: https://spectrum.ieee.org/at-work/innovation/the-2018-top-programming-languages

CRLT. (2016) *Teaching strategies: Large classes and lectures*. Accessed June 19, 2019 http://www.crlt.umich.edu/tstrategies/tsllc

Dean, J. & Ghemawat, S. (2004). MapReduce: Simplified data processing on large clusters. Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation (OSDI), vol. 6. Berkeley, CA, USA: USENIX Association.

Felleisen, M., Findler, R. B., Flatt, M., Krishnamurthi, S., Barzilay, E., McCarthy, J. & Tobin-Hochstadt, S. (2018). A programmable programming language. *Commun. ACM* **61**(3), 62–71.

Fraser, S. D., Bak, L., DeLine, R., Feamster, N., Kuper, L., Lopes, C.V. & Wu, P. (2015) The future of programming languages and programmers. In Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH). New York, NY, USA: ACM, pp. 63–66.

Freeman, S., Eddy, S. L., McDonough, M., Smith, M. K., Okoroafor, N., Jordt, H. & Wenderoth, M. P. (2014) Active learning increases student performance in science, engineering, and mathematics. *Proc. Nat. Acad. Sci. USA* **111**(23), 8410–8415.

Friedman, D. P., Wand, M. & Haynes, C. T. (2001) *Essentials of Programming Languages*. 2nd ed. Cambridge, MA, USA: MIT.

Graham, P. (2004). *Revenge of the Nerds*. Beijing: O'Reilly. Accessed June 19, 2019. Available at: http://www.paulgraham.com/icad.html

Guzdial, M. & Landau, S. (2018) Programming programming languages, and analyzing facebook's failure. *Commun. ACM* **61**(6), 8–9.

Hamer, J., Cutts, Q., Jackova, J., Luxton-Reilly, A., McCartney, R., Purchase, H., Riedesel, C., Saeli, M., Sanders, K. & Sheard, J. (2008) Contributing student pedagogy. *SIGCSE Bull.* **40**(4), 194–212.

Krishnamurthi, S. (2008) Teaching programming languages in a post-Linnaean age. *ACM SIGPLAN Not.* **43**(11), 81–83.

Krishnamurthi, S. (2012). *Programming languages: Application and interpretation*. Accessed June 19, 2019. Available at: https://cs.brown.edu/~sk/Publications/Books/ProgLangs

Kumar, A. N. (2005). Projects in the programming languages course. In Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE). New York, NY, USA: ACM, p. 395.

Kumar, V. (2019) *9 new programming languages to learn in 2019*. Accessed June 19, 2019. Available at: https://www.rankred.com/new-programming-languages-to-learn

Lee, K. D. (2015) A framework for teaching programming languages. In Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE). New York, NY: ACM, pp. 162–167.

Lewis, M. C., Blank, D., Bruce, K. & Osera, P.- M. (2016) Uncommon teaching languages. Proceedings of the 47th ACM Technical Symposium on Computer Science Education (SIGCSE). New York, NY: ACMs, pp. 492–493.

Louden, K. C. (2002) *Programming Languages: Principles and Practice*, 2nd ed. Pacific Grove, CA, USA: Brooks/Cole.

McCartney, R., Boustedt, J., Eckerdal, A., Sanders, K., Thomas, L., & Zander, C. (2016) Why computing students learn on their own: Motivation for self-directed learning of computing. *ACM Trans. Comput. Educ.* **16**(1), 2:1–2:18.

Miller, H., Haller, P., Rytz, L. & Odersky, M. (2014) Functional programming for all! Scaling a MOOC for students and professionals alike. In *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE)*. New York, NY, USA: ACM, pp. 256–263.

NCAT. (2014) *How to redesign a college course using NCAT's methodology*. Accessed June 19, 2019. Available at: http://thencat.org/Guides/AllDisciplines/How%20to%20Redesign%20A%20College%20Course.pdf. Chap. VI: How to Create Small within Large.

Papert, S. (1980) *Mindstorms: Children, Computers, and Powerful Ideas*. New York, NY, USA: Basic Book, Inc.

Papert, S. (1987) *Constructionism: A new opportunity for elementary science education*. Accessed June 19, 2019.Available at: https://nsf.gov/awardsearch/showAward?AWD_ID=8751190. National Science Foundation Award Abstract #8751190.

Perugini, S. (2018a) The design of an emerging/multi-paradigm programming languages course. *J. Comput. Sci. Coll.* **34**(1), 52–62.

Perugini, S. (2018b) *Programming languages: Concepts and implementation*. Accessed June 19, 2019. Draft. 1,240 pages. Course notes developed in conjunction with and for this textbook. Available at: http://academic.udayton.edu/SaverioPerugini/PLCI.

Perugini, S. & Watkin, J. L. (2018) ChAmElEoN: A customizable language for teaching programming languages. *J. Comput. Sci. Coll.* **34**(1), 44–53.

Pintrich, P. R. (2003). A motivational science perspective on the role of student motivation in learning and teaching contexts. *J. Educ. Psychol.* **95**(4), 667–686.

Pombrio, J., Krishnamurthi, S. & Fisler, K. (2017) Teaching programming languages by experimental and adversarial thinking. In *Proceedings of the 2nd Summit on Advances in Programming Languages (SNAPL)*, Lerner, B.S., Bodík, R., & Krishnamurthi, S. (eds), Germany: Leibniz International Proceedings in Informatics Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, pp. 13:1–13:9.

Prince, D. G. & Perugini, S. (2018). An application of the Actor model of concurrency in Python: Euclidean rhythm music sequencer. *J. Comput. Sci. Coll.* **34**(1), 35–46.

Queinnec, C. (2003). *LISP in Small Pieces*. Cambridge, UK: Cambridge University.

Ray, B., Posnett, D., Devanbu, P. & Filkov, V. (2017) A large-scale study of programming languages and code quality in GitHub. *Commun. ACM* **60**(10), 91–100.

Savage, N. (2018) Using functions for easier programming. *Commun. ACM* **61**(5), 29–30.

Scott, M. L. (2009) *Programming Language Pragmatics*. 3rd ed. Amsterdam: Morgan Kaufmann.

Sebesta, R. W. (2015) *Concepts of Programming Languages*. 11 ed. Boston, MA, USA: Addison Wesley.

Tate, B. A. (2010) *Seven Languages in Seven Weeks: A Pragmatic Guide to Learning Programming Languages*. Dallas, TX, USA: Pragmatic Bookshelf.

Tate, B. A., Daoud, F., Dees, I. & Moffit, J. (2014) *Seven More Languages in Seven Weeks: Languages That are Shaping the Future*. Dallas, TX, USA: Pragmatic Bookshelf.

The Joint Task Force on Computing Curricula: Association for Computing Machinery (ACM) and IEEE Computer Society. (December 20, 2013). *Computer science curricula 2013: Curriculum guidelines for undergraduate degree programs in computer science*. Tech. rept. Association for Computing Machinery (ACM) and IEEE Computer Society. Accessed June 19, 2019. Available at: https://www.acm.org/binaries/content/assets/education/cs2013_web_final.pdf.

Wexelblat, R. L. (ed) (1993). *ACM SIGPLAN Not.*, vol. 28. New York, NY, USA: ACM.