Proceedings of the Informing Science + Information Technology Education Conference

An Official Publication of the Informing Science Institute
InformingScience.org

InformingScience.org/Publications

June 30 – July 4, 2019, Jerusalem, Israel

DID YOU ALSO FALL ASLEEP DURING A PRINCIPLES OF PROGRAMMING LANGUAGES LECTURE? HOW DID A RE-DESIGN OF A PPL COURSE SUCCEED TO KEEP THE STUDENTS TUNED-IN? [DISCUSSION PAPER]

Moshe Goldstein*	Jerusalem College of Technology, Jerusalem, Israel	goldmosh@jct.ac.il
Ariel Stulman	Jerusalem College of Technology, Jerusalem, Israel	stulman@jct.ac.il

*Corresponding author

ABSTRACT

Aim/Purpose	In this paper we wish to present a new direction for the instruction of a Principles of Programming Languages (PPL) course.
Background	Teaching PPL using the standard curriculum found that the students do not understand the overall concepts, getting lost in the abundance of minute details. We needed a way to emphasize the higher level constructs important to this body of knowledge.
Methodology	This is a course description paper, describing how we instruct a PPL course at our college.
Contribution	To share with the CS education community the approach we developed to effectively teach the very important PPL course.
Findings	Using the integrative approach presented, we believe that • relative to the previous, and commonplace, PPL teaching approach, this is a very effective and successful way for conveying this important subject matter, and • our new teaching approach gave the students a professional maturity that they lacked before they took the course.

Accepting Editor: Eli Cohen | Received: December 31, 2018 | Revised: February 15, 2019 | Accepted: February 17, 2019.
Cite as: Goldstein, M., & Stulman, A. (2019). Did You Also Fall Asleep During a Principles of Programming Languages Lecture? How Did a Re-design of a PPL Course Succeed to Keep the Students Tuned-in? *Proceedings of the Informing Science and Information Technology Education Conference*, *Jerusalem, Israel,* pp 111-117. Santa Rosa, CA: Informing Science Institute. https://doi.org/10.28945/4329

Recommendations for Practitioners	Do not be scared to experiment with new ways of teaching. Do not think that you must teach the way the *books* tell it. If it doesn't feel right, it probably isn't.
Future Research	All our insights about the use of the presented teaching approach are non-empirical. Future research should thoroughly analyze the results from teaching/learning theories points of view using standard CSE techniques.
Keywords	principles of programming languages, CSE

INTRODUCTION AND MOTIVATION

This paper is based on our experience of how the syllabus of our principles of programming languages course evolved into one of the capstone courses in our B.Sc. curriculum. In the course presented, students not only learn principles of programming languages, but they also build them into a running two-layer compiler. This experience, albeit difficult to achieve, brings the students to their peak experience in the bachelor's degree. The paper describes the authors' efforts on creating a course which joins into one stream all the previous courses, gives students a common understanding of how the different, separate topics taught in such a course work together while providing them with a taste of a real life experience. The current course encompasses the principles of programming languages starting from the general concepts, through compilers, toward final working application.

A course on the Principles of Programming Languages (PPL) usually includes theoretical concepts of programming, such as the description of programming constructs (conditional, iteration, etc.) in a number of different programming languages, contrasting different approaches to each of these, etc. In many universities (MIT, Rice, Stonybrook, IU, CMU, ANU, BGU, etc.), the PPL course is also used as an opportunity to introduce students to functional (Lisp, Scheme, Haskell or ML) and/or logic (Prolog) programming paradigms. With most students learning to program in imperative (C) and object oriented (C++, Java, C#) based languages, these alternative paradigms broaden their knowledge. Now that Python ("Python programming language," n.d.) is being used as an introductory language (MIT, HebrewU, TAU, etc.), functional programming can even be taught without switching languages. In our college, students have a sophomore course for learning functional and logic programming, making it impossible to use this approach in our PPL course; this induced us to deviate from the standard courses found in those universities.

Until ten years ago there was no PPL course in our college. When such a course was proposed and taught for the first time by the first author, the curriculum was similar to that in use in other universities, and inspired by the two popular textbooks on the subject: Ravi Sethi's book (1989) and Sabesta's book (2012). For several years the course was completely theoretical, similar to a seminar course. Based on the theoretical concepts taught in the lectures, the students were required to write a critique about some language they did not know, and present their findings and assessment at the end of the semester. In such a course, there was no final exam.

During the 2008 academic year, the first author was on sabbatical and the course was taught by the second author. Coming from a computer security background, he was quite appalled when assigned to teach the PPL course. Not knowing what to expect, he mimicked what has been done in preceding years. This included the same theoretical concepts as in the past. The homework consisted of answering thought questions, e.g., "*what might be the benefit of a call-by-name calling convention"*, and reading classical language design debates (e.g., Dijskstra's [1968] seminal work *"GOTO statement considered harmful"*) and the discussion that followed (Dijkstra, 1987; Knuth, 1974; Moore, Musciano, Liebhaber, Lott, & Starr, 1987, and others).

Without having empirically supported evidence, our feeling was that the students were disgusted with the course. They could not understand why they needed to know this material - what difference does it make to their body of knowledge if loop invariants are really invariant (Pascal) or not (C/C++)?

Needless to say, with the students being bored without a clue as to why they are taking such a course, we felt dejected as well. It was a struggle to keep their attention, and with campus WIFI and laptops readily available, the students were only physically present. When exam time came around, it became apparent that although they could regurgitate what has been said, they still didn't understand basic concepts such as type systems, scopes, or memory management.

We realized that if we will not completely re-design the course, we shouldn't be looking forward for the next time. In order to revitalize it and make it more appealing and challenging for the students (and also for us!), we should cause the students to feel that this course is worth their time and effort.

In this paper, we describe the course that we set out to build. We describe our choice of topics, lab work, student participation and self-help. We explain how we deviated our course from the standard courses found in many universities, and the outcome of our efforts. The re-designed course went on to become the capstone of the B.Sc. curriculum (3rd year). It is our intention to share with the CS education community the approach we developed to effectively teach this important course of the curriculum of the B.Sc. in Computer Science.

COURSE METHODOLOGY

Our course has the following objectives:

1. Teach the core programming language concepts (see the ***Topics*** section, below), and
2. Have the students engage in a large self-taught and challenging semester-long experience:
    - They have to learn by themselves to program in a programming language completely unknown for them.
    - They have to learn by themselves how to get, install, and run a development environment for that language.
    - They were introduced to the principles of the two-layer architecture used in most programming language compilers today (see the ***Labs*** section, below). Based on those principles, they have to implement the compiler of a relatively simple object-based programming language by using the language they were assigned to use in the course.
    - They have to prepare a twenty-minute presentation about the language they were assigned.
    - They have to learn how to write entries for the Wikipedia site of the course (see the ***Wiki*** section, below), helping future students.

In order to accomplish these tasks, we split the lectures (three academic hours per week with a total of 13 weeks) into two alternating parts. The first (totaling 8 weeks) deals with the theoretical topics of programming languages (see the ***Topics*** section, below), and the second (totaling 5 weeks) deals with what they need to understand in order to solve the lab assignments. These parts intertwined during the semester, with part 2 being taught every week before an assignment was handed out (weeks 2, 5, 7, 9, 11).

COURSE TOPICS

When discussing which topics were to be included in the course curriculum, we made a number of key criteria that should heed the way:

- we will not dwell on syntax,
- we will choose two or three (contrasting) languages to demonstrate a point,
- we don't have to extensively discuss functional or logic programming due to the fact that the students already had a dedicated course; yet, we will bring out key points to contrast them with imperative and object-oriented language paradigms.

TOPICS

Based on the above criteria and the limited lecture time that is available (see the ***Course Methodology*** section, above), we chose the following topics to discuss:

- Introduction to language principles - syntax, BNF, semantics, language evaluation criteria (readability, writability, reliability, cost, portability, etc.).
- Type systems - the purpose of type systems, static vs. dynamic type systems (including duck-typing), weak vs. strong type systems, how specific types are implemented in some languages (based on Ben-Ari, 2006) and, pointers – their associated problems (i.e., memory leaks, dangling pointers, etc.) and solutions (e.g., locks & keys, *unique_ptr*, *shared_ptr*, etc.).
- Scopes - their purpose and meaning, dynamic vs. static binding, memory reclaiming methods (reference counting, garbage collection using mark & sweep, Cheney (1970), generational mark & sweep, etc. (Wilson, 1992).
- Sub-programs - procedural abstraction, parameter passing and binding, the semantic model for parameter passing (in, out and, in-out), the way of implementing these models (pass-by-value, pass-by-result, pass-by-value/result, pass-by-reference, pass-by-name) and demonstrating these options with specific languages, function calling conventions, the run-time calling stack, and recursion.
- Object oriented - data abstraction, abstract data types, encapsulation, information hiding, sub-typing, polymorphism, multiple vs. single inheritance (and problems and solutions for diamond inheritance), how polymorphism is efficiently achieved using vPtr and vTables.
- Functional Programming (FP) - review of the basic principles of FP: data immutability, pure functions, the substitution model order of evaluation, lambda expressions, closures, lazy evaluation.
- Concurrent Programming – basic introduction to language constructs in Message Passing (MPI) programming and Shared-Memory (openMP) programming.

LABS

It is the lab part of the course that is used to convey the two-layer architecture prevalent in many programming languages, while actually programming a full rudimentary compiler (two layers) that implements arithmetic and Boolean expressions, function calling and returning using a run-time stack, a specific calling convention, and high-level language paradigms. The lab assignments were the series of "mini" programming projects found at the end of each of the chapters 7-11 of (Nisan & Schocken, 2005). For this purpose, we taught those chapters as part of the lectures of the course (see the ***Course Methodology*** section, above).

Hand-in procedures

The students are presented with a specific "mini" project once every two weeks, and they must hand in their working solution before the next one is presented. During the lab sessions, the instructor proceeds to explain specific hard-points that they might encounter, and how they are expected to overcome them. Needless to say, the actual solutions or algorithms are not disclosed.

Each pair of students is randomly assigned a programming language that is not part of the degree curriculum (specifically excluding C/CPP, Java, Lisp, Prolog, C#, VB.NET, Python, etc.), which they had to self-learn. They are then required to hand-in their assignments in that specific language. The purpose of this requirement is three-fold:

- to teach them what is a compiler from language A to target language B. That it is just a program that can be implemented in any third language,

- to have them self-learn a programming language. This is part due to the departmental initiative of self-study in junior level courses, and part so as to boost self-appreciation of their capabilities, and
- to avoid code plagiarism by students. It is hard to copy other's work when written in a different programming language.

Due to the fact that all programs generate functionally similar output for a given input, it is quite simple to test and see if their compilers work as specified. Also, the excellent testing tools provided by the Nissan and Schoken's (2005) book's website make those running tests very easy and accurate.

In addition to all the above, we developed a very effective method for testing and grading lab assignments:

- all students' pairs must submit their solutions according to a strict deadline schedule;
- we do not test all the submitted works, but a representative part of them, randomly chosen – on average, every pair showed and ran their submission for two randomly chosen labs;
- at the end of the semester, during the 20-minute presentation of their programming language, they are also required to show and run one of the submissions that they did not demonstrate to their lab instructor during the semester;
- if that specific submission has any compilation or running errors, they receive 0 points for it.

This lab assignment's grading and testing method ensures that the students must work during the semester on those "mini" programming projects; if not, they endanger their total course grade.

Wiki

In order to assist with learning quirks of specific programming languages, the students are also required to develop a wiki[1] that includes - for each programming language -

- where to find good reading material, sample programs, tutorials, etc.
- which IDEs should be used (if one exists) and which to avoid.
- what compilers or interpreters are available, where they can be found and, how to get them up and running (installation, proprietary IDEs, plug-in into existing IDEs such as Visual Studios, Eclipse, NetBeans, etc.).
- specific, language-oriented commands that would save them many search hours when solving specific assignments (e.g. how to list the contents of a folder from within the program). These helpers are intended to allow students to spend the majority of their time programming the problem set and not reading many blogs and knowledge centers for a specific language-oriented task (usually a function of some obscure syntax).

This Wiki was developed, over time, by the students, for the students. That is, as part of the course obligations, each student has to augment the wiki with things that he found difficult and how he solved it. They have to write the problem, the solution and its' source, so future students can easily solve the same problem and look up the source for further reading. This implies that some wiki entries are more elaborate than others, depending on the entries from previous years. It is a student's job to help the next student save time he already spent searching for a given answer.

CONCLUSIONS

There are many ways for teaching a PPL course. As already mentioned, classical textbooks (Sebesta, 2012; Sethi, 1989) chose to contrast language constructs and the many options that a language designer has. Although this strategy is encompassing and covers many of the design decisions made, it

[1] Written in Hebrew; can be found at wiki.moodle.jct.ac.il

is extremely difficult to read and even more difficult to instruct. It did not capture our student's attention and failed to convey some of the more important principles we wanted to pass-on.

A course on PPL is supposed to give students the opportunity of learning programming concepts with a higher level of abstraction than in courses in which specific languages are taught. In order to achieve this goal, we completely re-designed the way we used to teach this course. Using the integrative approach presented here, we believe that,

- relative to the previous, and commonplace, teaching approach, this is a very effective and successful way for conveying this important subject.
- our new teaching approach gave the students a professional maturity that they lacked before they took the course.

All our insights about the use of the teaching approach presented here should be thoroughly analyzed from teaching/learning theories points of view, and the results of that analysis be published elsewhere in the future.

We hope that all what we presented in this paper will encourage other CS educators all over the world to try our approach, and we will be happy to receive feedback about their teaching experiences.

ACKNOWLEDGEMENTS

We would like to thank the chair of our CS department, Dr. Moti Reif, for allowing us to try our new teaching approach for the PPL course. We would also like to thank also all our colleagues and lab instructors who made possible the actual implementation of what we presented here. Last, but not least, we appreciate very much all the comments and suggestions given to us by the anonymous reviewers.

REFERENCES

Ben-Ari, M. (2006). *Understanding programming languages* (2nd ed.). John Wiley & Sons, Inc.

Cheney, C. J. (1970). A nonrecursive list compacting algorithm. *Communications of the ACM*, *13*(11), 677–678. https://doi.org/10.1145/362790.362798

Dijkstra, E. W. (1968). Letters to the editor: Go to statement considered harmful. *Communications of the ACM*, *11*(3), 147–148. https://doi.org/10.1145/362929.362947

Dijkstra, E. W. (1987). *On a somewhat disappointing correspondence*. Retrieved from http://www.cs.utexas.edu/users/EWD/ewd10xx/EWD1009.PDF

Knuth, D. E. (1974). Structured programming with go to statements. *ACM Computer Surveys*, *6*(4), 261–301. https://doi.org/10.1145/356635.356640

Moore, D., Musciano, C., Liebhaber, M. J., Lott, S. F., & Starr, L. (1987). GOTO Considered Harmful Considered Harmfu Considered Harmful. *Communications of the ACM*, *30*(5), 351–355. https://doi.org/10.1145/22899.315729

Nisan, N., & Schocken, S. (2005). *The elements of computing systems: building a modern computer from first principles*. The MIT Press.

Python programming language. (n.d.). Retrieved from http://www.pyhton.org/

Sebesta, R. W. (2012). *Concepts of programming languages* (10th ed.). Pearson.

Sethi, R. (1989). *Programming languages: Concepts and constructs*. Boston, MA, USA: Addison-Wesley Longman Publishing Co.

Wilson, P. R. (1992). Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management* (pp. 1–42). London, UK, UK: Springer-Verlag. Retrieved from http://dl.acm.org/citation.cfm?id=645648.664824

BIOGRAPHIES

Moshe Goldstein received his BSc in Computer Science (CS) from the Universidad de la Republica, Montevideo, Uruguay. He received his MSc in CS from Ben-Gurion University of the Negev, Beersheva, Israel, in 1982. In 2010 he received a PhD in Computational Chemistry from the Hebrew University of Jerusalem, Jerusalem, Israel (HUJI). As of 1995 he holds a position at the CS Department of the Jerusalem College of Technology (JCT), where, since 2010 he has been a Lecturer and Researcher. Since 2012 he also has been holding a position at the Computer Programming Instruction Unit of HUJI. His research interests are in the field of Protein Structure Prediction (evolutionary) algorithms. He also is one of the founding members of JCT's Flexible Computation Lab (flexcomp.jct.ac.il) where he makes research and development in the field of Parallel Programming methodologies and tools (in software and hardware), based on the idea of Flexible Computation. Dr. Goldstein is a member of IEEE and ACM (Professional).

Ariel Stulman received his bachelor's degree in Technology and Applied Sciences from the Jerusalem College of Technology, Jerusalem, Israel, in 1998. He then went on to get his masters from Bar-Ilan University, Ramat-Gan, Israel, in 2002. In 2005 he achieved a Ph.D. from the University of Reims Chanpagne-Ardenne, Reims, France. As of 2006 he holds a position at computer department of the Jerusalem College of Technology. His research interests are in the field of network and communications security and technologies, mobile ad-hoc security, and IoT security. He also researches topics in software testing, formal methods and real-time systems.

Dr. Stulman is a member of IEEE and ACM (Senior), and is the founding director of the cyber research group at JCT.