The teaching of functions as the first step to learn imperative programming

Carla A. D. M. Delgado[1], **João C. P. da Silva**[1],
Fabio Mascarenhas[1], **Ana Luisa Duboc**[2]

[1]Departamento de Ciência da Computação – Universidade Federal do Rio de Janeiro (UFRJ)
Rio de Janeiro – RJ – Brasil

[2]Departamento de Ciência da Computação – Colégio Pedro II
Rio de Janeiro – RJ – Brasil

{carla,jcps,fabio}@dcc.ufrj.br, alduboc@gmail.com

Abstract. *The literature on teaching programming covers a wide discussion regarding the approach to be adopted in an introductory course. While the object-oriented paradigm requires a high level of abstraction and can overshadow basic concepts, the structured imperative paradigm lacks a guideline to motivate and drive the process of building programs. In this paper we present a modularization based approach for teaching programming to the novices using the imperative paradigm. The construction of self contained modules of code - programming functions in Python - is worked with the students from the very first class until the end of the course. We describe our experience and report the results obtained from the implementation of this approach in a scenario of more than a dozen courses offered each semester over three years.*

1. Introduction

The teaching of algorithms and programming is an area of study and research that emerged in the 1970s and gained momentum in the 1980s [Robins et al. 2003]. Currently, many careers such as engineering, mathematics, meteorology and biology, require computing knowledge and programming skills. In higher education, an introductory programming course is of fundamental importance for any student in the area of exact sciences [McKeown and Farrell 1999], and in some countries, like Brazil, it is in the university that many students have their first contact with the computer, not only as users but also as developers.

The demand for professionals qualified in programming in several areas beyond computer science is growing, and should continue to grow in the coming years [ACM and IEEE 2013]. However, introductory programming courses (CS1 and CS2[1]),have high failure rates [Nikula et al. 2011, Mcgettrick et al. 2005, Delgado et al. 2004]. Given the importance of the topic and the major challenges involved, the literature on teaching programming often addresses the restructuring of CS1 and CS2, mainly due to the choice of the programming paradigm and programming language to be adopted [Koulouri et al. 2014]. The most considered options of programming

[1]CS1 and CS2 are the usual names for the Computer Science 1 and 2 courses, that cover introduction to programming in universities

paradigm are the Structured Imperative (SI) and the Object-Oriented (OO). The Functional paradigm appears less frequently.

The classical approach of programming teaching is based on the SI paradigm. The focus is on the teaching of the programming language structures and the mechanisms of storage and retrieval of information (reading and writing data to memory using variables). This approach allows an exhaustive exploration of these concepts, which are considered fundamental concepts for both SI programming as for OO programming (OOP). This is its greatest advantage. The OO approach, despite having the advantage of being very versatile and adopted in many software development projects, requires a higher level of abstraction and also a more extensive and elaborate syntax [Koulouri et al. 2014]. This approach, however, has the advantage of directly providing a program design technique.

For over more than 10 years teaching this course in different institutions, we accumulated experiences and intuitions about teaching programming for different profiles of novice students. During this time, we had the opportunity to evaluate students who took CS1 based both on SI and OO approaches[2], and with different lecturers. We observed that, in general, students who had good grades in introductory courses based on the SI paradigm progressed smoothly when faced with more advanced programming challenges, and even with OOP in subsequent courses. However, the percentage of failure and dropout in courses that adopt the SI approach is historically high.

On the other hand, we noticed high incidence of students who, despite having obtained good grades in CS1 based in the OO approach, not only had difficulties in understanding simple algorithms and programs written in the SI paradigm, but had also difficulty in developing more complex programs in the OO paradigm itself. The problem seems to get worse with the use of a ”black-box” teaching approach, where the student uses pre-built packages and classes, with little opportunity to build his or her own code from the language primitives. Our hypothesis is that basic concepts are not well assimilated because they are overshadowed by the OO abstraction. Our perception from previous experience led us to conclude that a good understanding and mastery of the structures and mechanisms of SI programming is essential to build a foundation for the abstraction required by the OOP.

While driving a reformatting process for CS1 and CS2 with the aim to improve both learning and operationalization of these courses (about 20 classes each semester), we put up a didactic proposal based on the existing literature and the contributions of our colleagues from the Computer Science Department in Federal University of Rio de Janeiro (DCC/UFRJ). Our proposal was further refined using the feedback we received during the following three years of its implementation, when we had several lecturers working according to it in our institution. The SI paradigm is our choice for the novices. But our approach deviates from the classics: instead of starting the course with the writing of complete programs, we start with the construction of programming functions. Our goal is to provide the student a guideline for the more abstract cognitive tasks of design and construction of programs, from the very beginning of the learning process. OOP is left to CS2. Regarding the programming language, we choose Python, mainly because of its simplicity.

[2]Experiences with the Functional paradigm were fewer and are not worth documenting here.

In this paper we present our didactic proposal for introductory programming courses and the results obtained with its implementation over three years in several classes of different university courses of STEM[3] at UFRJ. We also report our experience as coordinators of an initiative for the standardization of these course classes. Our proposal, although simple, allowed to give the students a more modern, cohesive and apparently more interesting format to CS1, without losing focus on conscious learning of the basics concepts. The adoption of the OO paradigm in CS2 also made up for a cohesive and motivating course, providing an appropriate scenario for tackling the use of graphical user interfaces and other more elaborate programming apparatus.

2. Related Works

[ACM and IEEE 2013] presents the three most common approaches to introductory programming courses: ”programming first” - the classical approach, with the imperative paradigm; ”Objects-first” - the object orientation is introduced early in a programming course; ”functional-first ”- programming concepts are introduced with a functional language. Arguments for and against each of these approaches have been raised. For example, a strength of the approach ”objects-first” is that, from the beginning, students have contact with programming principles widely used in industry. In contrast, a point against it is that the object orientation requires an abstract design and syntactic overhead that can complicate the programming task. The discussion does not reach a final verdict in favor of any particular option.

[Bruce 2005] summarizes a discussion that took place on the SIGCSE mailing list in March 2004 on how to teach introductory programming using Java. The article mainly focuses on one question: when objects should be introduced in CS1. The options discussed were (1) OO concepts should be introduced at the beginning of the programming learning; (2) OO concepts should be introduced in the ending of an introductory course; and (3) OO concepts should be left to a subsequent programming course (CS2).

An argument in favor of option (1) is that the OO contributes to learn design techniques of programs and modularization. Another aspect concerning (1) is that people that are used to the SI paradigm need to change the way they think (and teach). Thus, an argument against option (1) would be the unpreparedness of some lecturers due to lack of experience with OOP. Another disadvantage pointed out is that the concepts of SI programming such as loops and conditional statements receive little attention. These are the strengths of approaches (2) and (3), and the main argument in favor of the approach (3) is that each paradigm receives adequate attention. A reported disadvantage of approaches (2) and (3) is that once another paradigm has been learned, learning the ”way of thinking” required by OO becomes more difficult.

[Bruce 2005] concludes that approach (1) should preferably be adopted but in association with pedagogical tools for OOP teaching in order to allowing the reduction of the complexity of mental abstractions as well as easing the learning of the OOP language syntax. If this is not possible, option (3) would be preferable. We agree that option (3) is preferable with respect to option (2), but we can not assess whether option (1) would be our choice if a pedagogical platform for OOP was available. Never a platform of this

[3] An acronym for Science, Technology, Engineering and Mathematics.

kind was available at the institutions where we worked[4].

Several authors have contributed to this debate (such as [Ehlert and Schulte 2009, Reges 2006, Vilner et al. 2007]). A consensus, however, was never reached. We believe that hardly one of the alternatives will show to be superior to others in all scenarios, considering courses-students-lecturers-institution-learning resources.

Our proposal is well aligned with [Reges 2006] where the central idea is going "back to basics" in introductory programming courses. [Reges 2006] emphasizes problem solving (although he admits this is a fuzzy concept), procedural decomposition and the mastery of basic skills. The differential of our proposal is to emphasize (even more) on building concise code modules, letting the mechanisms of interaction with the user to the end of the course. We believe that in this way we are working more properly cognitive tasks related to the development of modular programs, without losing focus on the basics.

Although our proposal initiates the learning of programming with the definition and use of functions, it is distinguished from approaches that emphasize the learning of functional programming paradigm, such as the approach called "*Program by Design*" [Felleisen et al. 2001, Felleisen et al. 2004, Bieniusa et al. 2008, Sperber and Crestani 2012] and the previous approach of [Abelson and Sussman 1996], that had already been adopted in Brazil in the introductory programming course at PUC-Rio [Ierusalimschy 1997], and was later replaced by a more traditional approach [Celes and Ierusalimschy 2012]. From here on, every time we use the term *function* regarding programming we are referring to imperative functions or procedures, not the more mathematically pure functions of functional programming.

The use of functions from the beginning of the course as a unit of decomposition and abstraction is the only point in common between our proposal and functional approaches. Functional approaches exchange the use of the building blocks of imperative programming, such as variable assignment, dynamic data structure, sequence of commands, repetition structures and input and output commands, for pure functional programming, such as immutable data structures, recursion, high order functions and combining functions. Our proposal builds on the traditional teaching of SI programming, but increases emphasis on the use of functions and procedures as modularization tool, restricting the use of input and output commands to the main function of the program.

Regarding the programming language to be adopted in an introductory course, a consensus seems to exist. According to [Mannila and de Raadt 2006] one of the most important criteria for the choice of the language is the simplicity of the syntax and the structure of the language. The Python language appears as a choice that meets these criteria very well [Downey 2007], as well as offers the possibility of adopting both the SI and OO paradigms. Several authors have reported positive experiences concerning enhancing motivation and student satisfaction, reduction in failure and dropout rates, and increase of the grades with the adoption of Python instead of languages like Java or C / C ++ [Grandell et al. , Agarwal et al. 2008, Goldwasser and Letscher 2008]. The use of Python in CS1 and CS2 in UFRJ preceded the proposal we present in this article. Python language was adopted in 2007, and our observations confirmed the reports of the

[4]Main reasons for this are the lack of resources invested in acquiring and maintaining both hardware and software for undergrad labs

aforementioned authors.

3. Our Proposal

This proposal stems from an initiative for dealing with the systematic problems of CS1 and CS2 in UFRJ. The high dropout and failure rates, together with the visible frustration of lecturers and students, lead a group of 8 teachers from DCC/UFRJ to meet weekly during 2012 in order to study, exchange experiences, discuss and propose strategies for improving introductory programming courses offered to students of several STEM areas. A coordinator for this initiative was officially established in 2014, with a mandate to align the current practices with what has been discussed by the group, and to manage and organize the tasks for that end.

3.1. Scenario

DCC/UFRJ offers in average 25 CS1+CS2 classes in the first semester, and 24 in the second, a total of 49 classes per year. These are service classes: the students are not computer science students, but students of STEM disciplines in other departments. There is considerable rotation among the lecturers of these classes, with most of them being temporary adjuncts. There was a lot of dissatisfaction with these courses among students, course coordinators and lecturers. The main sources of dissatisfaction were: high dropout rates, poor grades, and insecurity by the part of coordinators that the courses were being taught properly, because there was great variation among both the subject matter and teaching styles.

The objective of our initiative was to standardize these courses, in order to (1) guarantee a minimum quality for the courses and (2) make them more efficient, by reducing individual labor and by evolving the course through consideration of previous experiences in both managing and teaching those courses. We bet on the effectiveness of this standardization to address the dissatisfaction that we have identified. To reach this objective, the following goals were set:

1. Review the syllabus of both courses (CS1 and CS2)
2. Setting strategies for teaching programming in courses with this profile
3. Dissemination of the new syllabus among teachers, students and coordinators
4. Uniformization of teaching plans and course material
5. Unification of exams and grading criteria
6. Supervision of how the courses are being taught

3.2. Course objective and syllabus

We did an exercise among the teachers to encourage them to think about what skills they expected the students to learn on an introductory programming course. The first version of this list of skills did not mention modularization, and we ended up noticing a lack of program design techniques, or essentially a guideline for leading the students in the constructions of algorithmic solutions to problems. After several iterations of refining this list of fundamental skills, we ended up with the following list:

- Identifying the relevant information in a problem and its respective representation and manipulation by the program.

- Comprehending the syntactical and semantic aspects of the programming language.
- Articulating commands, data structures and control structures for the construction of solutions for simple problems.
- Modeling and implementing modularized code for non-trivial problems.
- Constructing organized, reusable and legible code, following good programming practices.

After establishing this list of skills we reviewed the syllabus. Just the basic material relating to imperative programming has been kept in CS1. Accessory concepts like programming graphical interfaces have been moved to CS2.

3.3. Teaching strategies

As specified in the previous section, the biggest objective of the course is to develop skills for constructing correct, legible and organized programs. Once we have chosen modularization as the key program design technique that we were going to focus on, it was necessary to map this technique to specific programming language structures. For the CS1, our units of modularization are Python functions.

Understanding the concept of functions, as well as their construction and use, is vital to reaching the objective of the course. In the classical approach for teaching SI programming, this concept is often only taught almost in the end of the course, after all the commands and control structures of imperative programming, as well as simple imperative data structures, have already been taught. But the lecturers involved in reviewing our course have been unanimous in noting that, by only teaching functions near the end of the course, their learning is compromised because there is not enough time to properly practice the concept.

To deal with this problem, we had the idea of bringing the teaching of functions to the very beginning of the course. Besides solving the problem of lack of time for practicing the concept, we noticed other advantages of this strategy: the concept of function is the most concrete, in the universe of the students, to something that relates information (data) with the sequences of operations that transform this input information in the desired (output) information. The students already have prior exposure to mathematical functions, and can bring their mathematical intuitions to the study of functions in imperative programming. The connection with existing skills and knowledge facilitates learning and decreases sources of fear and frustration.

The students are familiarized from the start with writing simple functions and using the functions that they already wrote as building blocks in the construction of more complex functions. During the course, we gradually leave behind the intuition brought from mathematics, where functions are just an expression involving inputs and constants, to imperative functions implemented by blocks of sequential commands that can affect local and global variables. The students are encouraged to begin viewing their functions as a sequence of steps that are executed to solve some problem.

By working from the start with the identification and building of small cohesive functions, important abstract concepts like modularization, code organization, and code reuse are brought to practice and consideration of the students. Other advantages of beginning the course with functions include:

1	What is programming; Functions	7	Loop statement - while
2	Functions	8	Loop statement - for
3	Data types, strings. Decision Statement - if	9	Nested loops and matrices
4	Variables and assignment. Strings	10	Dictionaries
5	String manipulation, tuples, lists.	11	Basic input and output
6	Lists	12	More complex programs

Table 1. Material covered in each week of the course. A midterm exam happens between weeks 6 and 7, and final exams are after week 12.

- Notions of input and output of an algorithm are treated conceptually and not confused with console input and output (during most of the course the students interact directly with the functions they define via the Python REPL[5], and only in the final part of the course they are taught standard console input and output).
- The concept of parameters and return values precede the concept of variables. We noticed that students that were first taught to use variables struggle with the use of parameters as input values as well as the use of the return value from a function.
- We can practice developing and calling functions incrementally, without having to write a complete program. Short functions are also easier to test, and to debug.

The material covered in each week of the course can be seen in table 1. After covering functions and their use, we move on to simple data types (numbers, booleans and basic use of strings), and only after that we begin teaching conditional statements. Considerable time is spent on techniques for using conditional statements and testing code that uses them. Our view is that boolean logic is something new and non-intuitive (and thus error-prone) for students, and it takes time to properly build the formal knowledge to use these concepts correctly. Only after the student has mastered conditional statements we introduce variables and assignment. Local variables are shown as a way to organize and name intermediate results, and global variables as a way of keeping state.

Basic use of Python lists and tuples is seen after variables and assignment. Tuples and lists are the most simple structured data types in Python, having syntactic support in the Python language for their creation[6]. As the students have not yet been taught how to build loops, the focus is on basic list operations such as indexing, slicing, copying, mostly as a way of teaching how transforming and organizing data is the basis of algorithms.

Loops are introduced in the second half of the course, when we expect that the student has already internalized that a programming language is a formal language, unambiguous and deterministic. At this point, the student should be able to analyze and debug his or her own code. In order to develop this skill, besides assigning several laboratory exercises where the students are expected to build increasingly more complex programs, we also feature several exercises where the students ”bench test” their programs[7]. Traversing and manipulating lists and matrices (encoded as lists of lists) are the

[5]Read-Eval-Print-Loop, a special console where the programmer can evaluate Python expressions and commands.

[6]Vectors at the C language style do exist in Python, but are not the conventional structured data types in Python, and its creation and manipulation are not as straightforward.

[7]A bench test is a step by step simulation of the program, either using pen and paper or a tool such as Python Tutor [Guo 2013].

main motivators for building functions that use loops.

In the end of the course we cover basic input and output or "interacting with the user", consisting in the construction of a main program that uses console input and output commands to gather information from the user that is supplied to the program's functions, and to show the results to the user. We stress that "algorithmic" functions should never directly interact with the user, allowing the same functions to be reused with other methods of interaction. The interaction should only be done either in the main program or in functions specifically built for that.

4. Results and Conclusion

The main sources of dissatisfaction identified in our scenario were the high dropout rates, failure, and uncertainty about the consistency and quality of the courses. When thinking about these issues, we realize that they are in fact not local issues, but issues related to changes in the higher education scenario. The amount of classes, courses and teachers involved grew an order of magnitude or more, making inappropriate the treatment given in the past. To operationalize courses with these new orders of magnitude, strategies that do not take into account the continuity of the teaching-learning process beyond an isolated discipline are not appropriate. Class changes, reproofs and the preservation of continuity of learning for a next discipline are important demands. Achieving quality in this scenario is far from what a lecturer can do alone focusing only on the class he teaches, regardless of the effort to do the best. To tackle the problems, we had to think back the core of the courses: objectives, course content, teaching strategies and syllabus.

Our main contribution is the proposed teaching methodology: start with structured imperative programming, focused on modularity in CS1, and then focus on teaching OOP in CS2. In this paper, we underlined the methodology used in CS1, addressing the basic contents of SI programming in an unconventional order as summarized in table 1. To the best of our knowledge, a SI function-centered teaching approach for CS1 is not documented in the literature. The benefits of this new approach were summarized in section 3.3. As the lack of a design strategy was pointed as a big disadvantage of classical SI-based CS1 courses, it is worth mentioning that this approach provides an underlying program design strategy - the modularization.

Besides reformatting CS1 and CS2, systematizing the provision of these courses was also a useful strategy to ensure the quality and increase efficiency. When you have a high rotation of lecturers and a considerable number of temporary adjuncts, every individual evolution is easily lost. The establishment of a coordinator to keep track of the classes and maintain contact with the lecturers was very important. Tasks to achieve the desired standardization of the classes, like the development of the course material and its continuous improvement could be centralized by someone who had a global picture in mind. The course material consists of slide presentations for theoretical lectures and lists of exercises for practical lectures, and is freely available at `http://ladybugcodingschool.com/` for viewing and download.

We consider that, after four years of work, from the goals presented in section 3.2 were satisfactorily met the review and wide dissemination of the syllabus(1 and 3), the establishment of strategies for teaching programming (2), and the uniformization of teaching plans and course material (4). The unification of exams and grading criteria (5)

and the supervision of the courses (6) were partially achieved.

Wide dissemination of the syllabus, despite looking like a detail, brought several benefits. This goal was achieved with the creation of a website, and the dissemination of this, every semester, to everyone involved (teachers, students and coordinators). The syllabus works as a contract, in a high level, establishing what to expect from the course, leaving those involved more confident that what is expected will actually be delivered.

The unification of the exams and grading criteria occurred during one year, but went no further because it requires the involvement of many people in different roles (secretary, heads, lecturers, and even higher councils). Perhaps, with more time, it could be possible to get engagement by all persons involved, and thus make the unified exam a reality in our institution. We are now experiencing the first year in which the unification of the exam did not happen, and we realized that it begins to affect the work: as each lecturer now has complete autonomy on the evaluation of their own class, the commitment to stick to the contents of the syllabus was weakened.

Our perception is that the emphasis on the construction and use of functions from the beginning of an introductory course reduces the time and effort required to develop the skills to build programs, in addition to providing the basis for the further development of this skills in future stages. Even teachers who had resistance to adopt this new approach, commented they were satisfied with the outcome. Students who have failed before in the course and who experienced both approaches, have reported the preference for the new one. As future work, we intend to conduct a detailed analysis of the grades of all the classes where this approach has been implemented, and to compare data collected before and after this implementation. Our preliminary analysis has revealed an interesting phenomenon: the polarization of the grades was attenuated, suggesting that the learning curve was smoothed. Our next goal is to consolidate these assumptions from the historical data.

References

Abelson, H. and Sussman, G. J. (1996). *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition.

ACM and IEEE (2013). Computer science curricula 2013: Curriculum guidelines for undergraduate degree programs in computer science. Technical report, ACM Joint Task Force on Comp. Curricula ; IEEE Comp. Soc.

Agarwal, K. K., Agarwal, A., and Celebi, M. E. (2008). Python puts a squeeze on java for cs0 and beyond. *J. Comput. Sci. Coll.*, 23(6):49–57.

Bieniusa, A., Degen, M., Heidegger, P., Thiemann, P., Wehr, S., Gasbichler, M., Sperber, M., Crestani, M., Klaeren, H., and Knauel, E. (2008). Htdp and dmda in the battlefield: A case study in first-year programming instruction. In *Proc. of the 2008 Int. Workshop on Func. and Decl. Prog. in Education*, FDPE '08, pages 1–12, NY, USA. ACM.

Bruce, K. B. (2005). Controversy on how to teach cs 1: A discussion on the sigcse-members mailing list. *SIGCSE Bull.*, 37(2):111–117.

Celes, W. and Ierusalimschy, R. (2012). Apostila de programação i.

Delgado, C., Xexéo, J. A. M., Souza, I. F., Campos, M., and Rapkiewicz, C. E. (2004). Uma abordagem pedagógica para a iniciação ao estudo de algoritmos. In *Anais do XII Workshop de Educação em Comp. (WEI)*.

Downey, A. B. (2007). Python as a first language: Pre-conference workshop. *J. Comput. Sci. Coll.*, 22(6):3–4.

Ehlert, A. and Schulte, C. (2009). Empirical comparison of objects-first and objects-later. In *Proc. of the Fifth Int. Workshop on Comp. Education Res.*, ICER '09, pages 15–26, NY, USA. ACM.

Felleisen, M., Findler, R. B., Flatt, M., and Krishnamurthi, S. (2001). *How to Design Programs: An Introduction to Programming and Computing*. MIT Press, Cambridge, MA, USA.

Felleisen, M., Findler, R. B., Flatt, M., and Krishnamurthi, S. (2004). The structure and interpretation of the computer science curriculum. *J. Funct. Program.*, 14(4):365–378.

Goldwasser, M. H. and Letscher, D. (2008). Teaching an object-oriented cs1 -: With python. *SIGCSE Bull.*, 40(3):42–46.

Grandell, L., Peltomäki, M., Back, R.-J., and Salakoski, T. Why complicate things?: Introducing programming in high school using python. In *Proc. of the 8th Australasian Conf. on Comp. Education - Vol. 52*.

Guo, P. J. (2013). Online python tutor: Embeddable web-based program visualization for cs education. In *Proc. 44th ACM Technical Symposium on Comp. Science Education*, SIGCSE '13, pages 579–584, NY, USA. ACM.

Ierusalimschy, R. (1997). Apostila de introdução à ciência da computação.

Koulouri, T., Lauria, S., and Macredie, R. (2014). Teaching introductory programming: A quantitative evaluation of different approaches. *Trans. Comput. Educ.*, 14(4):1:28.

Mannila, L. and de Raadt, M. (2006). An objective comparison of languages for teaching introductory programming. In *Proc. of the 6th Baltic Sea Conf. on Comp. Education Research*, Baltic Sea '06, pages 32–37, NY, USA. ACM.

Mcgettrick, A., Boyle, R., Ibbett, R., Lloyd, J., Lovegrove, G., and Mander, K. (2005). Grand challenges in computing: Education—a summary. *Comput. J.*, 48(1):42–48.

McKeown, J. and Farrell, T. (1999). Why we need to develop success in introductory programming courses. *The Journal of Computing in Small Colleges*, 14(3):241–250.

Nikula, U., Gotel, O., and Kasurinen, J. (2011). A motivation guided holistic rehabilitation of the first programming course. *Trans. Comput. Educ.*, 11(4):24:1–24:38.

Reges, S. (2006). Back to basics in cs1 and cs2. *SIGCSE Bull.*, 38(1):293–297.

Robins, A., Rountree, J., and Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Comp. Sci. Education*, 13(2):137–172.

Sperber, M. and Crestani, M. (2012). Form over function: Teaching beginners how to construct programs. In *Proc. of the 2012 Annual Workshop on Scheme and Functional Programming*, Scheme '12, pages 81–89, NY, USA. ACM.

Vilner, T., Zur, E., and Gal-Ezer, J. (2007). Fundamental concepts of cs1: Procedural vs. object oriented paradigm - a case study. *SIGCSE Bull.*, 39(3):171–175.