Teaching of Programming with a Programmer's Theory of Programming

Juris Reinfelds
Klipsch School of EE & C,
New Mexico State University
Las Cruces, NM, USA
juris@nmsu.edu

Abstract: We review the introductory programming courses of the widely accepted Curricula '68, '78, '1991 and '2001. We note that a one-language, imperative-paradigm approach still prevails, although multi-language programming systems are already available. We discuss the Kernel Language Approach, which provides a programmer's theory of programming that permits a widening of introductory courses to multi-language, multi-thread programming without loss of depth. We suggest two broad outlines for the removal of the one-language constriction from introductory programming courses. We observe that because of the introduction of dotNET and because of student exposure to net-centric multimedia applications, text-based "Hello World !" examples disappoint the expectations of today's students.

Key words: programming methodology, programming courses, curriculum, computer science, software engineering, introductory courses,

1. INTRODUCTION

In order to make progress, we first divide and conquer and then we unify [1]. In the sixties scientific programming was done in FORTRAN, commercial data processing was done in COBOL and algorithms were supposed to be programmed in an "algorithmic language" called ALGOL. These three language groups did not speak to each other much, went to separate conferences, did not read each other's publications. Soon each

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35619-8_15
L. Cassel et al. (eds.), *Informatics Curricula and Teaching Methods*

developed its own style of programming and the notion of *programming paradigms* was born.

Excessive originality was highly praised, especially in academia, which adopted the algorithmic language branch, and created the important functional, logical and object-oriented paradigms as well as an endless stream of "new" programming languages. In 1969 [3] Jean Sammet described 120 reasonably widely distributed programming languages in her book. She maintained a yearly register of programming languages for the first half of the seventies, where we can see how casually new programming languages were proclaimed and how quickly they were forgotten.

As a new science trying to gain respect among older, well-established sciences, computing needed a theory in a hurry. Mathematicians had elegant theories of computability, developed when hand computations by mathematicians were the only way to compute. Computer science adopted them, gained respect and "forgot" that computer programmers work with a very different set of concepts than hand-calculating mathematicians or mathematicians who are interested in the foundations of computability.

2. PROGRAMMING IN THE CURRICULUM

ACM Curriculum'68 [2] was the first widely accepted curriculum for undergraduate Computer Science. It combined programming with problem solving in a 3-credit-hour lecture-plus-laboratory course:

2.1 B1 – Introduction to Computing

> "... a **single** algorithmic language should be used for most of the course... It may be desirable to use a **simple second** language of quite a different character for a problem or two to demonstrate the wide diversity of computer languages available..."

The suggested course material for B1 was very extensive, so that the next ACM Curriculum'78 [4] split it into two 3 credit-hour lecture-plus-laboratory courses that "forgot" about the second language suggestion of B-1

2.2 CS-1 Computer Programming

> "... introduce problem solving methods and algorithm development...
> ...teach **one** high level programming language that is widely used...
> ...teach how to design, code, debug and document programs..."

2.3 CS-2 Computer Programming II

"...continue development of discipline in program design... ...introduce basic aspects of string processing, recursion and simple data structures..."

Curriculum'78 also had a course on programming languages and their implementation that was based on I-2 of Curriculum'68:

2.4 CS-8 Organization of Programming Languages

"...develop understanding of organization of programming languages and their run-time behaviour..."

The focus on one programming language created by Curriculum'68 and reinforced by Curriculum'78 started a heated debate as to which language should be *the chosen one*, a debate that continues today. A glance at the program of the most recent ACM SIGCSE'2002 conference [5] shows two sessions on CS-1 and one session on CS-2 where the question *"Which language will give us the quickest and best understanding of programming?"* still dominates.

3. RECENT CURRICULA

IEEE-CS joined with ACM to create Computing Curricula 1991 [6] which did not question the one-language approach and retained CS-1 and CS-2 as *"Introduction to Computing I and II"*, while CS-8 became *"Programming Languages"* with emphasis on a feature by feature comparison of major programming paradigms. The language implementation part of CS-8 was dropped into a course on compilers.

The most recent Computing Curricula'2001 [7] acknowledged that a consensus on "the chosen language" or even "the chosen paradigm" has not been reached and is not likely. Instead, Curricula'2001 introduce a gaggle of introductory courses entitled Imperative first, Object first, Functional first, Breadth first, Algorithms first and Hardware first, that are otherwise very similar to CS-1 and CS-2. The Programming Language course suggested by Curricula'2001 retains the "comparison of language features" character of CS-8.

Although we now have to solve problems with scope and size that were unthinkable twenty years ago, nothing much has changed in the programming curricula since 1968 and 1978. As Peter Van Roy remarks in his paper in this conference [8]

"...programming is taught as a craft in the context of a single paradigm ... often explicitly limited to a particular language and toolset. Almost no attempt is made to put these tools into a uniform framework."

4. A PROGRAMMER'S THEORY OF PROGRAMMING

A *programmer's theory of programming* should explain programming in terms of concepts that are already familiar to programmers. A *theory of programming* should capture the essence of programming and place a unifying foundation under programming languages.

The first programmer's theory of programming was developed by Edsger W. Dijkstra [9]. Dijkstra introduced the notion of "guarded statements" and captured the essence of imperative programming with a very small set of well-chosen and precisely defined statements. Precise semantics were defined using Boolean expressions as pre-conditions, post-conditions and loop invariants. Dijkstra's theory was very useful for the creation of easily understood programs for difficult algorithms [9], [10], [11].

5. THE MULTI-PARADIGM PROGRAMMING LANGUAGE OZ

Is it possible to design a programming language that is equally expressive, but much smaller and more elegant than the union of all programming language features from all paradigms? The Oz programming language [12] answers the question with a resounding yes.

Can such a language be implemented reliably and efficiently on multiple platforms? The Mozart [12] system gives us an implementation of Oz, a compiler, a runtime environment and program development tools that are outstanding in every respect.

Without direct knowledge of how it actually happened, we can guess that the designers of Oz first extracted the essence of each paradigm. Then they devised a syntax and semantics that combine these essentials into one precisely defined programming language. Finally they added some "syntactic sugar" which are statements that increase the expressivity of the programmer and the readability of the program.

One way to keep a combination of several things small is to look for commonalities that were overlooked before. For example, in the heyday of esoteric mainframes, each operating system was an independent creation

influenced by the underlying hardware, but even more by the exuberant creativity of its designers.

Then came UNIX that captured the essence of an operating system by stating that everything that stores information is a *file* and everything that manipulates information is a *process*. The hardware and software implementation of files and processes were left to hardware and software designers. Programmers could reason about operating system behaviour without resorting to implementation arguments and the first multi-platform system was born.

Oz bases the essence of programming languages on values and threads. A *value* is either simple (e.g. integer, float, procedure ...) or structured (e.g. array, record, list ...). A *thread* is an executing statement-sequence. A new, concurrent thread, which runs in parallel with the thread that creates it, may be created with the simple statement

thread <any statement sequence> **end**

Moz maintains a global name-space of unique internal names of variables that permits a simple variable-scope-rule:

> For all threads, remote as well as local, all variables that are in scope when a thread-statement executes, remain available during execution of the thread body, regardless of termination of the parent thread execution.

In other words, variable scopes do not change when a statement sequence is executed in its own thread. The run-time system makes values available to processes that need them in a transparent and efficient way.

By contrast, Java language designers placed the burden of remote-thread variable resolution on Java programmers and Java run-time implementers added to the programmer's burden the proper placement of "synchronized" tags on selected methods.

To accommodate declarative programming and to facilitate reasoning about multi-thread programs, Oz uses assign-once-only (like final variables of Java) variables as the primary variables. For programming with state, Oz provides cells, which are assign-many–times variables.

6. THE KERNEL LANGUAGE APPROACH

Compared to its multi-paradigm, distributed and concurrent programming scope, Oz is a surprisingly compact language. Nevertheless, Van Roy and Haridi [13] went a step farther and defined a subset of Oz that captures the essence of Oz in a remarkably small and elegant syntax and operational semantics. Every statement of Oz can be reduced to statements of this subset. Hence they named the subset *"A Kernel Language of Oz"*. Since Oz

spans the major paradigms as well as multi-thread computing, we expect that this Kernel Language will provide a basis for most programming languages as discussed by Van Roy and Haridi [13], where they extend the Kernel Language approach to the programming languages Erlang, Haskell, Java and Prolog.

There is one more level of structure. The Kernel Language has a very small declarative core that we may regard as the basic foundation of programming. With small extensions of the core Van Roy and Haridi [13] create Kernel Languages for the bases of other paradigms and for concurrent and distributed computing.

7. IMPACT OF KERNEL LANGUAGE ON INTRODUCTORY COMPUTER SCIENCE TEACHING

From Curriculum'68 to Curricula'2002, three courses, CS-1, CS-2 and ProgLangs have been allocated to the teaching of programming. CS-1 and CS-2 were tied to one programming language. This has not changed to this day.

Artificial boundaries of programming paradigms prevent students from seeing programming as a unified whole. The first "chosen language" FORTRAN was replaced by BASIC, which was replaced by Pascal, which was replaced by C, which C++ tried to replace and which is being replaced by Java. Curriculum'68 [2] had great foresight when it tried to tie its Programming Languages course to

> "... a survey of the significant features of existing programming languages with particular *emphasis* on the *underlying concepts* abstracted from these languages..."

but without a programmer's theory of programming there was no coherent set of underlying concepts and the course degenerated into a descriptive comparison of programming language features [14].

There have been some attempts to overcome the one-language syndrome of CS-1 and CS-2. None have survived mainly due to faculty reluctance to master more than one programming language, program development environment and compiler.

With National Science Foundation support in 1992 and 1993, we developed a laboratory-based three-paradigm CS-1, CS-2 course sequence [15], [16], with Prolog, Miranda and C representing the logical, functional and imperative paradigms. Students enjoyed the course, but faculty, support staff and teaching assistants did not. The mastering of three unrelated

program development environments and compilers took too much time and energy away from the programming focus of the courses.

With the multi-paradigm, multi-thread language Oz and a Kernel Language with which we can define and discuss the essence of each paradigm as well as concurrent and distributed computing in simple, yet precise terms, we finally have the missing theory of programming with which we can reorganize CS-1, CS-2 and ProgLangs to teach more material, achieve better depth of understanding and reach more students than with the current suggestions of Curricula'2002 which simply continue the approach of Curriculum'68 and '78.

Included below are two suggestions at the two ends of the student mind-set spectrum of how we can improve CS-1, CS-2 and ProgLangs with the help of the Kernel Language Approach.

8. INTRODUCTORY CURRICULUM FOR SERIOUS STUDENTS

Serious students want a deep understanding of programming concepts as soon as possible. Serious students are not comfortable with the use of black-box components and large, opaque subprogram libraries. They are not happy with having to click-select menu-items more or less at random to see "by experiment" whether the item fits into the program. Instead they like to reason about and clearly understand program statements and components before they use them in their programs. This suggests the following three courses

8.2 CS-1

Introduce the essence of imperative, object-oriented, functional, logical, concurrent and distributed programming through a study of the Kernel Language together with well chosen laboratory programming problems and pre-programmed examples in Oz.

8.3 CS-2

Update the usual set of algorithms and data structures covered in conventional CS-1 and CS-2 and illustrate them with problems and pre-programmed examples. Show how to use Kernel Language programming and reasoning skills to rapidly acquire conventional language (e.g. Java) programming skills.

8.4 ProgLangs

Introduce students to different programming languages and paradigms via net-centric multi-language multi-platform programming using dotNET or a similar multi-language, multi-platform programming system.

9. INTRODUCTORY CURRICULUM FOR NINTENDO GENERATION STUDENTS

In the April 2002 issue of CACM, Mark Guzdial and Elliot Soloway [17] recognize that even in a down-turned economy many programming jobs go unfilled while the dropout/failure rate in CS-1 courses is in the 15% - 30% range. They quote a report [18] that found shockingly low performance on simple problems even among 2nd year college students, surveying 4 schools in 3 different countries.

Soloway & Guzdial note that engaging the students is critical to deep learning. They observe that "Hello World!" and other text-based programming problems do not engage today's students. They suggest that multimedia programming will most likely engage the full attention and energy of today's Nintendo, MTV generation of students.

9.1 CS-1

Introduce programming concepts through practical web-page construction in the laboratory. Explain each concept with the help of the Kernel Language in a "just-in-time" fashion.

9.2 CS-2

Introduce net-centric, multi-thread, multi-platform programming skills. Explain concepts with Kernel Language. Motivate the use of CS-1, CS-2 covered data structures and algorithms by suitably chosen laboratory excercises. Provide pre-programmed examples in Oz. Extend to multi-language programming along the lines of Bertrand Meyer's two-language Ticket Reservation System example [19].

9.3 ProgLangs

Use Kernel Language to teach what is common to all paradigms and how the paradigms differ. Teach how to use Kernel Language Knowledge to

rapidly acquire programming skills in a widely used programming language e.g. Java.

10. THE DOTNET EFFECT

Mozart-Oz is a well-designed, well-implemented multi-paradigm, multi-thread and multi-platform programming system. At New Mexico State University, our experience with teaching programming with Mozart-Oz and the Kernel Language echoes the results reported by Van Roy [8]. Our students have also commented that they finally understood what the Java or C++ course had tried to teach them only after they were able to discuss these programming concepts in terms of the Kernel Language in our Mozart-Oz based programming course.

Before dotNET [20], one could ask whether our view of multi-paradigm programming might be too rosy, too optimistic. One could ask if multi-paradigm, multi-thread, multi-platform programming has a practical future.

Now that dotNET, initiated by Microsoft and supported by a consortium of major companies including IBM, Intel and HP, has shown that multi-language, net-centric, multi-platform programming is indeed possible and available, we urgently need a programmer's theory of programming with which to manage all this new knowledge that the next generation of programmers will need. One-language programming will be made obsoleteby dotNET and its successors. Therefore the sooner we can find new, more appropriate and effective ways to teach programming the better.

11. CONCLUSIONS

The Kernel Language approach provides a programmer's theory of programming, which programmers can use to reason about programs using terms and concepts that are already familiar to programmers.

The trend to multi-language, multi-paradigm, multi-thread, multi-platform programming that was initiated by dotNET, requires urgent redefinition of our one-language based CS-1, CS-2 introductory programming course sequence.

Another reason for urgent revision of the introductory programming course sequence is the student disenchantment with the text-based, "Hello World" type programming problems and program examples of current courses.

Therefore we should look very seriously and with some urgency at a redesign of the CS and Software Engineering introductory course sequence

to make the courses more relevant for the programming requirements as well as the student expectations of tomorrow.

12. ACKNOWLEDGEMENTS

The author is grateful to Peter Van Roy for two immensely intensive days in December 1999 when Peter taught me the basics of Oz programming. The author is grateful to Seif Haridi for the opportunity to spend a week at the Swedish Institute for Computer Science to meet with experienced Mozart-Oz programmers, especially Fredrick Holmgren who showed me how to do agent based programming in Oz. The author acknowledges the Mozart Consortium for the design of the programming language Oz that makes programming a pleasure and for the Mozart system that implements Oz so effectively and reliably.

13. REFERENCES

[1] Meyer, B., "The Power of a Unifying View", Software Development, June 2001.
[2] ACM Curriculum Committee, "Curriculum 68", CACM Vol. 11, #3, p.151-197, (1968)
[3] Sammet, J. E., "Programming Languages: History & Fundamentals", Prentice Hall (1969)
[4] ACM Curriculum Committee, "Curriculum'78", CACM Vol. 22, #3, p.147-167, (1979)
[5] Knox, D., (Ed.), "Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education, Feb. 27 – March 3, 2002, North Kentucky, ACM Press (2002)
[6] Tucker, A. B., Ed., "Computing Curricula 1991", http://www.acm.org/education/curr91/homepage.html, IEEE Computer Society Press (1991)
[7] Computing Curricula 2001, CS Volume 1, http://www.acm.org/sigcse/cc2001/
[8] Van Roy, P., Haridi, S., "Teaching Programming Broadly and Deeply: the Kernel Language Approach", (this volume), Kluwer Academic Publishers (2002)
[9] Dijkstra, E. W., "A Discipline of Programming", Prentice Hall (1976, 1997)
[10] Gries, D., "The Science of Programming", Springer (1981)
[11] van de Snepscheut, J. L. A., "What Computing is All About", Springer (1993)
[12] Mozart Consortium, http://www.mozart-oz.org
[13] Van Roy, P., Haridi, S., http://www.info.ucl.ac.be/people.PVR/book.html
[14] Wilson, L. B., Clark, R. G., "Comparative Programming Languages", Add. Wes. (1988)
[15] Reinfelds, J., "A Three Paradigm Course for CS Majors", Proceedings, 26th ACM SIGCSE Technical Symposium on Computer Science Education, p.223-227, (1995)
[16] Reinfelds, J., "1996 Lecture Notes for CS 272 (new)", 251 pages, Department of Computer Science, New Mexico State University (1996, 1997) .
[17] Guzdial, M., Soloway, E., "Teaching the Nintendo Generation to Program", CACM Vol. 45, #4, p.17-21, (2002).
[18] Mc Cracken, M., et al., "A multinational, multi-institutional study of assessment of programming skills of first-year CS students", ACM SIGCSE Bul. 33, 4, p.125-140 (2001)

[19] Meyer, B., "A multi-language example using Eiffel#, C# and ASP+ in dotNET", http://www.dotnet.eiffel.com, (2002).
[20] Meyer, B., "The Significance of dotNET", IEEE Computer, August 2001.