Computer Science in 2018

Editors: J. Voas, R. Kuhn, C. Paulsen, and K. Schaffer

We surveyed six of our profession's best senior computer science educators:

MICHAEL: Michael Lewis **(College of William and Mary)** rmlewi@wm.edu

KEITH: Keith Miller **(U. of Missouri – St. Louis)** millerkei@umsl.edu

SHIUHPYNG: Shiuhpyng Shieh **(National Chao Tung U.)** ssp@cw.nctu.edu.tw

PHIL: Phillip A Laplante **(Penn State U.)** plaplante@psu.edu

JON: Jon George Rokne **(U. of Calgary)** rokne@ucalgary.ca

JEFF: Jeff Offutt **(George Mason U.)** offutt@gmu.edu

We felt their opinions would be useful to IT Pro's readership to address 3 questions:

1. **What are today's core classes in computer science education and are they generally uniform in most universities and colleges? How do they compare with those in the early days in computer science education (1970s and 1980s)?**

MICHAEL: Looking around at various programs, I was a bit surprised to see the extent to which vestiges of the old core classes are still around: discrete math; data structures and algorithms; programming languages; computer organization or computer architecture; some sort of software development course. There is also a fairly ubiquitous math requirement of something like 2-3 semesters of calculus plus linear algebra.

It's a bit surprising how static the core of the curriculum seems to be, given how much computer science has changed over the years.

Of course, the content of the core CS classes has changed in varying degrees, and the electives are far more varied and numerous than what we had in the past. Indeed, many of the topics of electives today did not exist in the 1970s and 1980s.

JON: There has not been a general agreement on what exactly is the core of computer science neither today nor in the past. However, one can probably say that today's core classes in computer science are generally focused on procedural programming, data structures and algorithm analysis supported by courses in mathematics and statistics and that in the early days of computer science (1970's and 1980's) there was a greater emphasis on lower level

programming and courses dealing with computer hardware and fewer courses on data structures and software engineering.

SHIUHPYNG: Today's core classes include programming (e.g. algorithm, programming language, and data structure), mathematic (e.g. linear algebra and discrete mathematics), and system design (e.g. computer architectures and operating system). Although the objectives of the core classes remain the same, the content varies significantly in comparison with that in the 1970s and 1980s. It includes many new techniques we take for granted today, e.g., multi-tasking, just-in-time compilation, networking, and artificial intelligence. Moreover, the core classes need to cover new requirements.

KEITH: One way to answer this question is to reference the ACM/IEEE-CS Computer Science Curricula 2013 (https://www.acm.org/education/CS2013-final-report.pdf). It is not the only approach to computing curriculum available on the world stage (for example, the European Union is working on the Bologna Process, including computing). But CSC2013 approaches curriculum not on the basis of "classes," but on the basis of "hours." An hour is meant to be the amount of material covered in an hour of "lecture," although the CSC2013 document takes pains to not endorse lecture as the preferred method of pedagogy.

With that introduction, please look at Figure 1, which summarizes both CSC2013 and two previous curriculum guidelines from the same group.

Knowledge Area	CS2013 Tier1	CS2013 Tier2	CS2008 Core	CC2001 Core
AL-Algorithms and Complexity	19	9	31	31
AR-Architecture and Organization	0	16	36	36
CN-Computational Science	1	0	0	0
DS-Discrete Structures	37	4	43	43
GV-Graphics and Visualization	2	1	3	3
HCI-Human-Computer Interaction	4	4	8	8
IAS-Information Assurance and Security	3	6	--	--
IM-Information Management	1	9	11	10
IS-Intelligent Systems	0	10	10	10
NC-Networking and Communication	3	7	15	15
OS-Operating Systems	4	11	18	18
PBD-Platform-based Development	0	0	--	--
PD-Parallel and Distributed Computing	5	10	--	--
PL-Programming Languages	8	20	21	21
SDF-Software Development Fundamentals	43	0	47	38
SE-Software Engineering	6	22	31	31
SF-Systems Fundamentals	18	9	--	--
SP-Social Issues and Professional Practice	11	5	16	16
Total Core Hours	**165**	**143**	**290**	**280**

All Tier1 + All Tier2 Total	**308**
All Tier1 + 90% of Tier2 Total	**293.7**
All Tier1 + 80% of Tier2 Total	**279.4**

Figure 1. Tier 1 (required) and Tier 2 (roughly, recommended) are the "core hours" in CSC2013.

Although they aren't defined as such, it seems clear that several classes familiar to most CS graduates can be identified by locating large numbers of hours in the table: algorithms (row AL), computer architecture (row AR), discrete structures (DS), programming languages (PL), and software development (SDF) and software engineering (SE) stand out to me. Personally, I agree that these 6 areas, covered by some collection of courses, would be fundamental for any BS in Computer Science. I wouldn't think they would be sufficient, but certainly would be a base upon which to build.

We can go back to 1968 to compare early computer science curricula to the 2013 recommendations. An ACM task force in 1968 (William F. Atchison, Samuel D. Conte, John W. Hamblen, Thomas E. Hull, Thomas A. Keenan, William B. Kehl, Edward J. McCluskey, Silvio O. Navarro, Werner C. Rheinboldt, Earl J. Schweppe, William Viavant, and David M. Young, Jr.. 1968. Curriculum 68: Recommendations for academic programs in computer science: a report of the ACM curriculum committee on computer science. Commun. ACM 11, 3 (March 1968), 151-197.) recommends eight courses in the core curriculum, shown here:

Introduction to Computing

Computers and Programming

Introduction to Discrete Structures

Numerical Calculus

Data Structures

Programming Languages

Computer Organization

Systems Programming

We see strong similarities with the 2013 document, but several noteworthy differences. Calculus, though discussed in the 2013 document, is not explicitly mentioned in the 2013 core, though discrete structures are. Software engineering appears in 2013, but is not explicit in 1968. Perhaps most striking is how many hours in 2013 are given over to topics not explicitly covered in the 1968 core, including social issues and practice, parallel and distributed computing, and intelligent systems. In the decades between 1968 and 2013 it isn't surprising that new topics gained importance; perhaps it is surprising how similar many of the emphases are.

PHIL: I have been teaching in computer science and software engineering programs since the mid-1980s. Over the years, I have seen curricular changes to make room for breadth courses unrelated to Computer Science, and to make the programs more accessible to those who are not strong in mathematics. I believe that these changes come at the expense of a deeper understanding of computation.

ABET (the Accreditation board for Engineering and Technology) accredits CS programs through guidelines provided by the CSAB (Computer Science Accrediting Board) – a joint effort of IEEE and ACM. The current guidelines say that a CS curriculum must have

- "Coverage of the fundamentals of algorithms, data structures, software design, concepts of programming languages and computer organization and architecture.
- An exposure to a variety of programming languages and systems.
- Proficiency in at least one higher-level language.
- One year of science and mathematics"

Most CS programs comply with these guidelines with little variation.

Notice that the ABET recommendations omit operating systems, compiler theory, automata theory, database theory and more – courses that were traditionally taught in most CS programs through at least the late 90s and which remove the "magic" from how computers and software applications really work. ACM/IEEE 2016 curricular recommendations (http://www.acm.org/education/curricula-recommendations) provide for more depth beyond that ABET criteria, but I am not sure how widely adopted these are.

JEFF: ABET accredits most CS programs in the US, and as a result, the requirements, at least in North America, tend to be fairly uniform. Students from the 1980s (like me) still recognize many of the year one through three courses. Courses like introductory CS, introductory programming, data structures, computer organization and assembly, operating systems, computability, and algorithms are still standard. At the higher level, we see newer topics like security, web and mobile app development, game development, and big data analysis alongside standbys like database, graphics, networks, and AI. I just looked at the requirements at my university (George Mason) and the coursework we had in 1980 would come very close to satisfying current requirements. I find that surprising.

2. **Is the following statement true: "the brightest computer science graduates are often heavily self-taught due to their passion for this area?" How often do you experience cases where the students know more than their professors?**

   KEITH: In my experience, that statement is true, although I don't think it is always the brightest students who are heavily self-taught. I think it is often the most passionate students who are heavily self-taught. And this self-teaching is often in specific areas, and mostly concrete areas. Students may know quite a bit about how to program particular machines or systems, even though their understanding of algorithms in general (for example) may not be particularly sophisticated. I don't think I can quantify how often this happens to me, but surely in a class of 25 undergraduates it happens several times in a semester that at least one student knows at least some detail about a particular system or programming language that I don't know (or remember).

   MICHAEL: I agree that the brightest CS students generally learn a lot on their own or by working with equally bright students. These students have an active extra-curriculum of personal programming projects, hackathons, programming contests, and various jobs and internships. The extra-curriculum plays a critical role in the development of CS students.

   In my experience it's pretty common for students to know more about particular software tools or frameworks than their professors do. Students frequently return from internships with all manner of knowledge such as web programming frameworks they learned over the summer. Also, in their personal software projects they end up doing things that faculty might not typically do (e.g., hacking Bluetooth drivers on cellphones).

   On the other hand, CS faculty know more about the science part of computer science than do students, so we're probably worth keeping around.

   PHIL: My experience is that the best young computer scientists have a solid undergraduate education but supplemented with lots of hands on experience obtained through some combination of internships, part-time jobs and self-study. There are so many open source projects to work on, free tools, and low cost small platforms, such as Arduino and Raspberry Pi, to play with that there is no excuse for young computer scientists to not have lots of hand on experience by the time the graduate -- the best students take advantage of these opportunities.

As for students knowing more than their professors – in one area or another, my students know more than me all the time. I can't be expert in every programming language, every development environment, application domain or every piece of hardware. I constantly learn from my students.

SHIUHPYNG: In computer security discipline, both attackers and ethical hackers are often self-taught. More and more self-learning resources are now available on the Internet. The knowledge and implementation skill in computer science can be easily digitalized and distributed online, and therefore the passion can motivate the students to polish their skills through self-learning. As computer science domain knowledge expands and grow rapidly, it may be a challenge for professors to keep up with the fast growing areas and in particular to cover the cross-disciplinary hands-on experience. As an example, Zuckerberg, the Facebook founder, took one year to build up an intelligent house which can follow speech commands, control switches, and even tell a joke. Many cross-disciplinary know-hows are involved in this case. However, in my personal opinion, the academic way of thinking and problem solving still be the key to the success of new technologies.

JON: It is certainly true in that some of the very bright computer science students are heavily self-taught. However, their knowledge base tends to have gaps that need to be filled in through more formal education processes. This was exemplified by one of our very successful undergraduate students who was challenged in his graduate work elsewhere by the advanced theoretical computer science courses expected of a graduate student. This talented student had avoided most theoretical courses offered in his undergraduate studies. It is not difficult for a bright computer science student to know more than a professor if knowledge is measured by detailed knowledge of specific software or hardware products. For a deep understanding of computer science professors are seldom challenged by the students.

JEFF: I've only seen 2 or 3 students in my 30 year career who are primarily self-taught to be software engineers. Many are self-taught to be programmers, but most were bad programmers. Certainly not engineers. And many students have very high self-efficacy, believing that they already know everything because they've written a few Android apps. I see a few seniors surpass their good teachers, and always find that exhilarating. Unfortunately, I see more students surpass teachers because their teachers do not know much.

3. **IEEE's Computer Society has developed a Body of Knowledge (BOK) and an examination to be tested against the BOK to be licensed as a software engineer. Do you see any impact from this now or in the future?**

JEFF: Not personally. The last time I looked at the BoK it struck me as out of touch with modern software engineering. It looked like something that may have been appropriate in 1995, not 2017. That's just my opinion of course.

JON: It is difficult to assess the impact of the Body of Knowledge in a computer science department since is specifically aimed at software engineering (incidentally a term coined by F. L. Bauer). Computer science, as we think of it, encompasses software engineering, but includes

other disciplines too. Splitting off new departments of software engineering from computer science would mean less interaction cross disciplinary research in the overall computer science area.

MICHAEL: I have not seen any impact of this, even though my state (Virginia) was one of those that first asked for a licensure system.

It is interesting how many of the core topics in the IEEE software engineering Body of Knowledge are those core CS courses discussed in the first question, e.g., discrete math; data structures and algorithms; software development; computer architecture. This supports the view that the CS curriculum really has an identifiable core.

SHIUHPYNG: The BOK contains necessary knowledge in a summarized form. It will be very helpful to new graduates. In this case, the license may be a good way of ensuring the fundamental knowledge of developers.

PHIL: I led the effort to create the professional engineer (PE) licensing exam for software engineers in the US, so I can answer this question from a unique perspective. First, for several reasons a different body of knowledge, similar to SWEBOK, had to be created for the licensing exam. We put a tremendous amount of effort in surveying hundreds of professionals, creating the body of knowledge, and writing (and maintaining) the exam. The reasoning and process behind the exam are described in Phillip A. Laplante, Beth Kalinowski, Mitchell Thornton, "A Principles and Practices Exam Specification to Support Software Engineering Licensure in the United States of America," Software Quality Professional, vol. 15, no. 1, January, 2013, pp. 4-15.

Unfortunately, since its introduction in 2013 there have been few exam takers (less than 100). There are several reasons for this. First, (with some exceptions) deans, department chairs and faculty seem generally uninterested in promoting the exam. Secondly state departments of engineering have not been uniformly requiring software PEs for public works that contain software. Without this requirements, there is little demand for licensed software engineers. Finally, the path to licensure is difficult because candidates must pass the Fundamentals of Engineering exam. This examination is very broad based and expects the candidate to be knowledgeable in areas that most engineers would study, but not most software engineers.

KEITH: The issue of licensing software engineers has a long and contentious history. The issue of whether software engineers should, or will, be licensed has certainly not been settled, despite decades of hard work by several organizations. For example, see (Laplante, Phillip A. "Licensing professional software engineers: seize the opportunity." Communications of the ACM 57.7 (2014): 38-40.) For a contrary view in the popular press, see (Bogost, Ian. "Programmers: stop calling yourselves engineers." The Atlantic (2015)).

Because of the lack of consensus among professional organizations, I don't think that licensing will be important in the next few years. It may gain momentum if there are many spectacular disasters. Meanwhile, the BOK may improve our understanding of the principles important to our profession, and may have a more immediate and direct effect on curricula and practice.