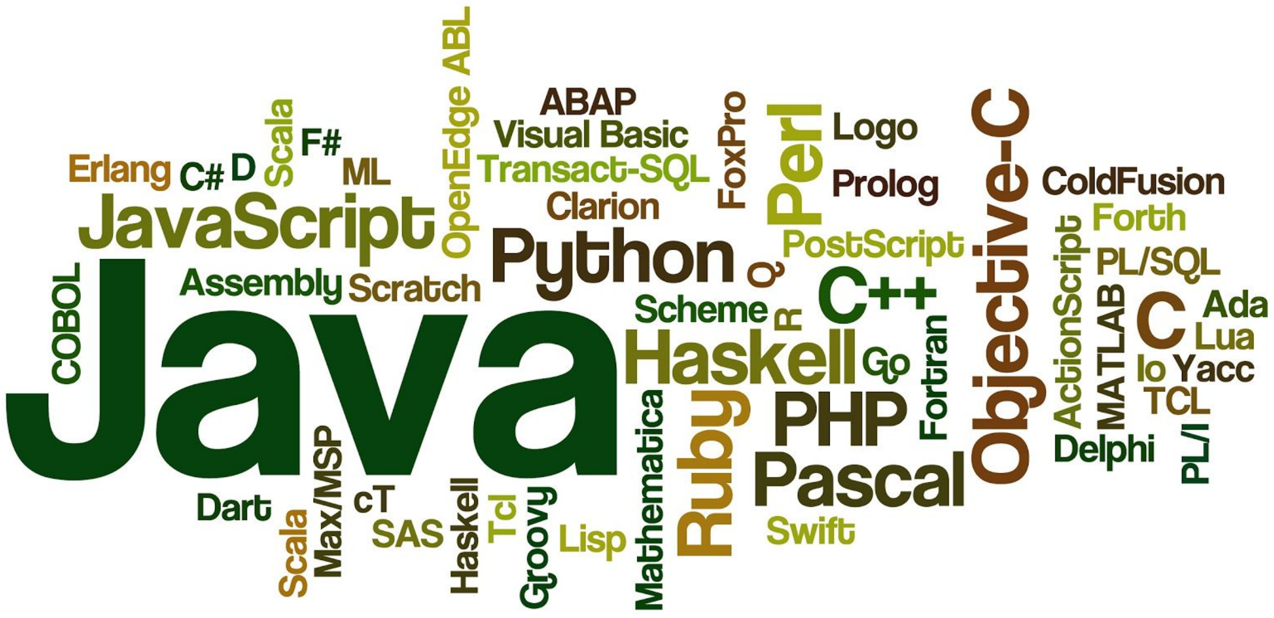
An analysis of educational strategies for introductory programming courses



Prepared by Vaughn Kottler for
Mike Shapiro, EPD 397: Technical Communication
Department of Electrical and Computer Engineering
University of Wisconsin-Madison
March 21, 2017

Screenshot source: https://github.com/vkottler/badgerloopControls/blob/master/Pic32/peripherals/C/pc_buffer.c

Memo of Adjustment: Draft & Ideation -> Final Submission

My initial proposal included the following as factors to be considered in the research:

- Programming Language Choice
- Mode of Instruction (Lecture vs. Exercise)
- Course Scope
- Measure of Success

The following research does cover these topics, focusing mainly on programming language choice and purposefully omitting explanations for different programming paradigms (object orientation, etc.) as no background information on such topics will be provided.

The paper is not organized according to these four topics, though the paper's organization is made clear in the executive summary and table of contents.

I examine real-world introductory programming courses and top internship positions which is not explicitly stated in the proposal. I find that having these two elements to cross-examine with the peer-reviewed research provides a strong foundation for supporting claims made in my conclusion.

The proposal contains evidence I include in the introduction, and I tried my best to incorporate compelling statistics into a warrant of sorts for the topic I am pursuing. I do not know if providing a warrant is a requirement, but it comes directly from a recommendation made by the instructor when he pointed out that my evidence doesn't prove a need for the education to be improved. Proving the education needs improvement as a student is admittedly a trecherous endeavor, and because of that I admit that my introduction contains arguably anecdotal information.

On that note, the proposal doesn't effectively define the scope of my research, which the executive summary makes up for.

Background on computer architecture was moved to the appendices. It's truly important information for understanding why language choice is important, but is anecdotal for purely following along with the research and arguments presented.

Memo of Adjustment: Final Submission -> Revision Submission

Changelog:

- Language popularity and summary moved to Appendix
- Analysis re-ordered so that job descriptions appear before course descriptions
    - this was how my presentation was structured, it occurred to me during the presentation process that this is more logical
- Figure 14 & 15 updated—coloring added as used in presentation
- Executive summary entirely re-written
    - This was a major critique of my original final submission
- Appendix III (advice from intern supply) removed
    - After reading it more thoroughly, I actually entirely disagree with their advice. I find doing exercises in practice interview questions to get a job an attack on my values. Does it not accomplish the same thing to use programming to solve real problems, a process that will allow you to learn the same exact things?
- Background section revised to explain purpose of appendix sections
- Conclusion entirely re-written
- Content revised where necessary because of organizational changes
- Table of contents updated

Out of Scope:

Introduction & Analysis content revision. A truly polished final product would require me to further cut down on verbose analysis and potentially even reduce the amount of analysis. I consider this beyond what is really necessary but acknowledge it as something that may have made this revision better.

Table of Contents

Executive Summary

Computer science (CS) and programming education carry the burden of preparing students for the responsibility of maintaining our growing digital infrastructure [6,11,15]. The goal of this research is to combine academic research with real-world job position descriptions to determine which current introductory computer science course does the best job of preparing students for the field in addition to making a general recommendation for how courses could be revised to better suit students needs.

The topic is introduced with information regarding the demand for computer scientists, engineers, and IT professionals to indicate a potential need to investigate the education being provided.

Background information that may be necessary to understand technical terms and topics discussed will be covered next. These topics are covered in depth in the appendices.

From here, academic research is consulted to begin constructing a model for an ideal introductory computer science course.

To complement what can be learned from researchers, internship position descriptions at top tech companies are examined to further narrow down the criteria for an ideal course.

Finally, real introductory courses will be surveyed to select the best course out of the sample size and concluding remarks will be provided based on all research that was conducted.

Introduction

Computers & Rapid Digital Integration

Computer programming is formally defined as a process that leads from an original formulation of a computing problem to executable computer programs. More generally, it is the means of controlling a computer to complete a desired task.

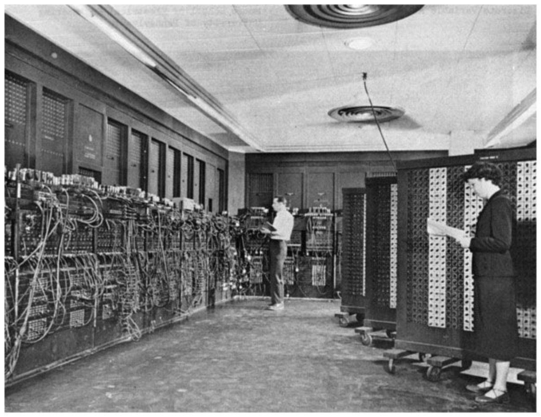


Figure 1: ENIAC at the University of Pennsylvania circa 1946, considered the first digital computer [7]

The range of tasks computers perform today, and even the definition of a computer itself has grown rapidly since the days of manually-fed, perforated tape strips and large machines that required rewiring of flipping of physical switches (Figure 1) to be controlled. A computer today can be considered anything containing a microcontroller or microprocessor that is making decisions for an electronic device or system. This covers a wide range of gadgets, and computer-occupation related jobs alone are predicted to make up 51% of all STEM related work by 2018 (Figure 2).

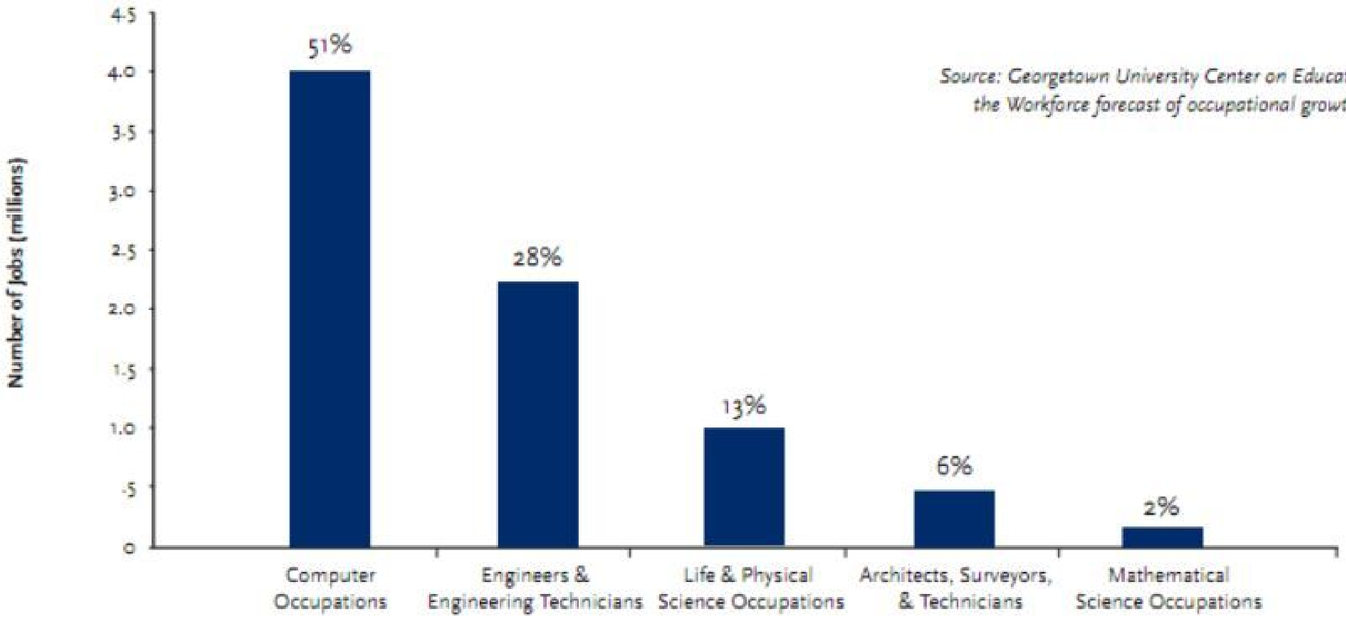


Figure 2: Predicted breakdown of available jobs in different STEM categories for 2018 [6]

It is also predicted that 65% of all jobs in the U.S. economy will require postsecondary education by 2020 [11]. Knowing roughly how many roles need to be filled in the near future and assuming a majority of these roles will require a degree of some kind, how prepared is the education system to fulfill its duty to prepare even the most immediate generation of graduating high school and college students?

Supply != Demand

At the high school level, CollegeBoard's Advanced Placement program offers a CS curriculum with an accompanying exam that over 6,000 of the world's leading educational institutions recognize as a legitimate measure of future college students' competence in various subjects [12]. In most cases, a high enough score on an AP exam grants a student college credit and potentially exemption from certain general graduation requirements.

In 2016, 4,810 high schools were certified to teach the AP Computer Sciences course (**11.42%** of U.S high schools) and **57,937** students took the exam (**out of 15 million** total high school students) [13, 14]. Although dismal, this is significant growth from the previous five years based on historical stagnation (Figure 3):

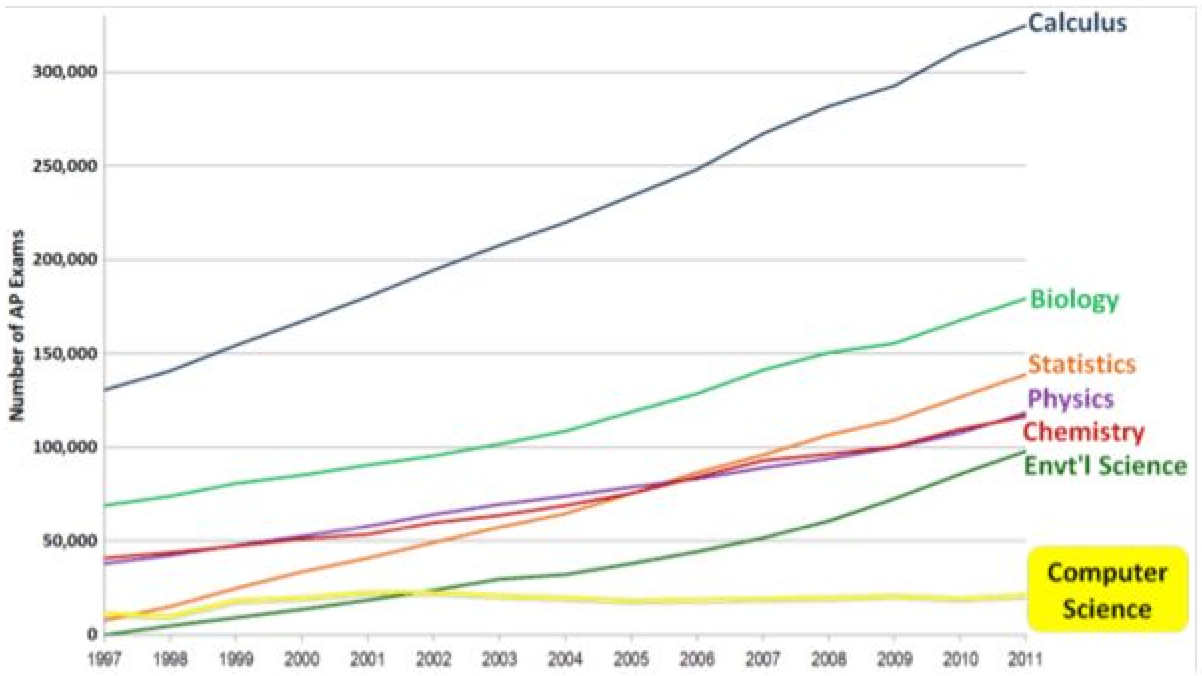
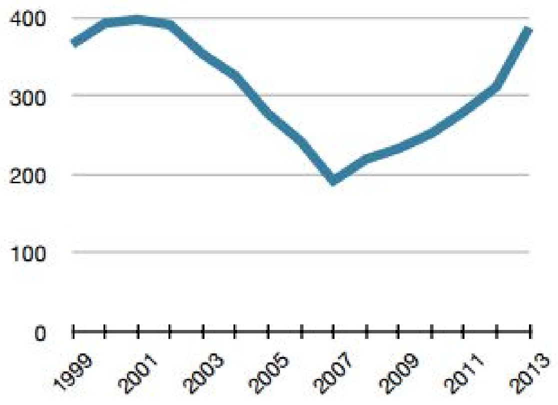


Figure 3: AP Exam Participation 1997-2011 [15]

An exact correlation between a student's future major choice and his/her AP exam portfolio has not been released since 2011, but at that time **less than 30%** of students who took the exam went on to study CS [16]. A similar trend of growth can be observed in the enrollment of computing-related majors at the college level over this same set of years (Figure 4):



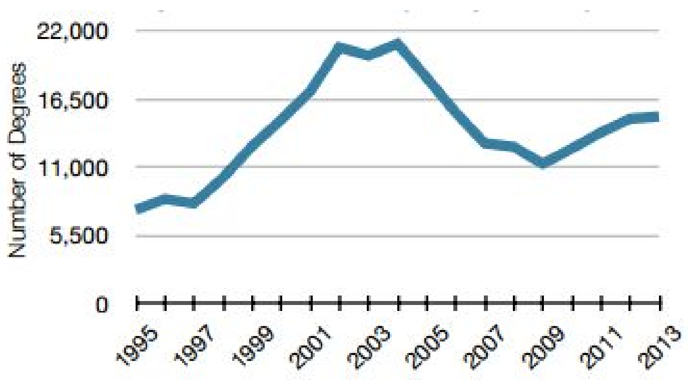


Figure 4: (Left) Average CS majors per U.S. CS Department. (Right) Total BS Production (All Departments) 2013 [15]

Implication

The education systems are having a difficult time adjusting to the societal demand for programmers. So much, that we experience a significant deficit in the number of graduating computer and information systems students relative to the number of job openings available annually (Figure 5):

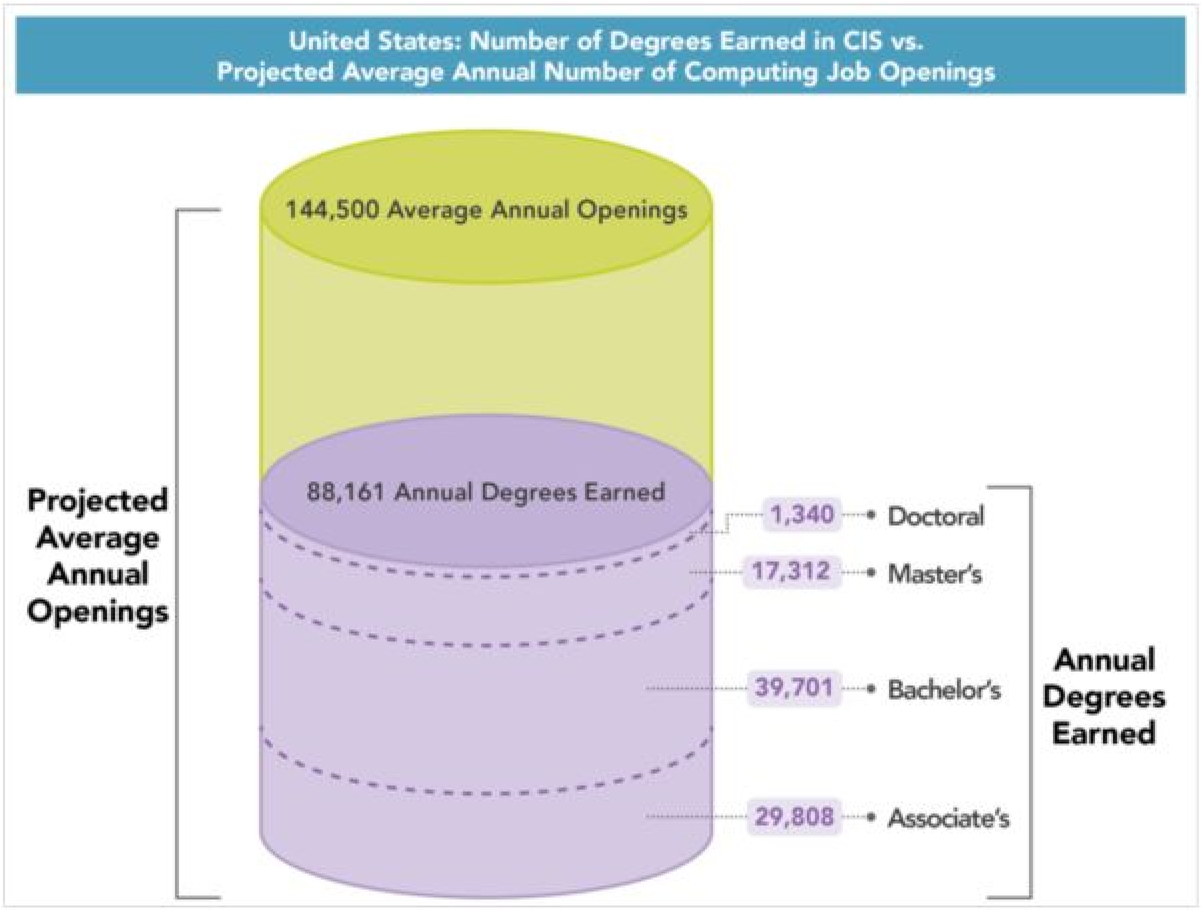


Figure 5: Ratio of jobs available in CIS fields vs. number of students graduating with relevant degrees, annually 2013 [15]

The primary implication of this provides a warrant that is two-fold to an investigation of introductory CS education:

1. It is the responsibility of an introductory CS course to retain students [2], which is necessary to reduce the annual deficit of jobs. The introductory course provides the opportunity for students to develop an identity as someone interested in CS and IT (Information Technology) in addition to serving as the first step towards actualization of such careers [2].
2. Lack of sufficient competition in the job market, and the variety of jobs available do not bode well for ease of effective curriculum implementation, academic research needs to be consulted to determine what a standardized curriculum may look like to prevent companies from making up for gaps in experience out of pocket.

One may argue that investigating introductory CS education on this basis is not warrant enough, and that the warrants are not based on arguments echoed by peer-reviewed academic research. This is a valid argument, but whether or not the educational institutions should be held accountable for the deficit of programmers is not the main focus of this research.

Background

There are a number of computing topics that are important to be aware of in order to fully understand the arguments to come.

This information has been organized into appendices:

Appendix I: High Level Language Summary

"Code" refers to human readable text written in a programming language. There are many programming languages, and the reasons for that are explored in the following appendix, but I strongly recommend this section to have context when trying to understand the discussion of language choice in an introductory course.

Appendix II: What Computers Are and How They Work

I recommend reading this section if computer architecture is a foreign concept. It is important to understand how a computer executes code in order to better internalize how languages are different and why it matters from the perspective of education.

The following section consists of the research that was conducted.

Research & Analysis

This section will be presented in the following order:

1. Peer-reviewed, academic research discussing methods of teaching introductory CS
2. Comparison of the above section with what companies require of student interns
3. Examination of a selection of real-world introductory CS courses

1. Theory of Education

In “Programming Languages and Teaching”, an independent report in 2000 authors had the following to say about the difficulty in teaching CS at the high school level:

In the experience of the author, the main problem with teaching programming languages to younger students has been that the results obtained from a lot of hard work (in most programming languages) bears little resemblance to the programs the students see and use everyday on their own computers. [1]

The authors studied the students’ results from a course they designed with **Java** as the chosen language for the following reasons:

- Object-orientation
- Relevant to industry and to the student
- Easily implemented with regard to cost and resources
- Provides a wide range of tools for the programmer and have good re-use

It is also mentioned that at this point in time, object-oriented programming was considered too difficult to be a suitable first programming style, and that many schools and institutions preferred procedural programming though the authors predicted this to be changing in favor of object-orientation [1].

Another study from the Uppsala Computing Education Research Group in 2014 concurs with the notion that it may be *unlikely* for a student to consider CS as a suitable field of study in the future without a way to relate to the material in a meaningful way [2]. Their study analyzed the ways that students “experience participation” in CS/IT by interviewing these students to find out what they believed life would be like as a computer scientist or IT engineer, how they think they became interested in the field, and what their first-year experience had been like.

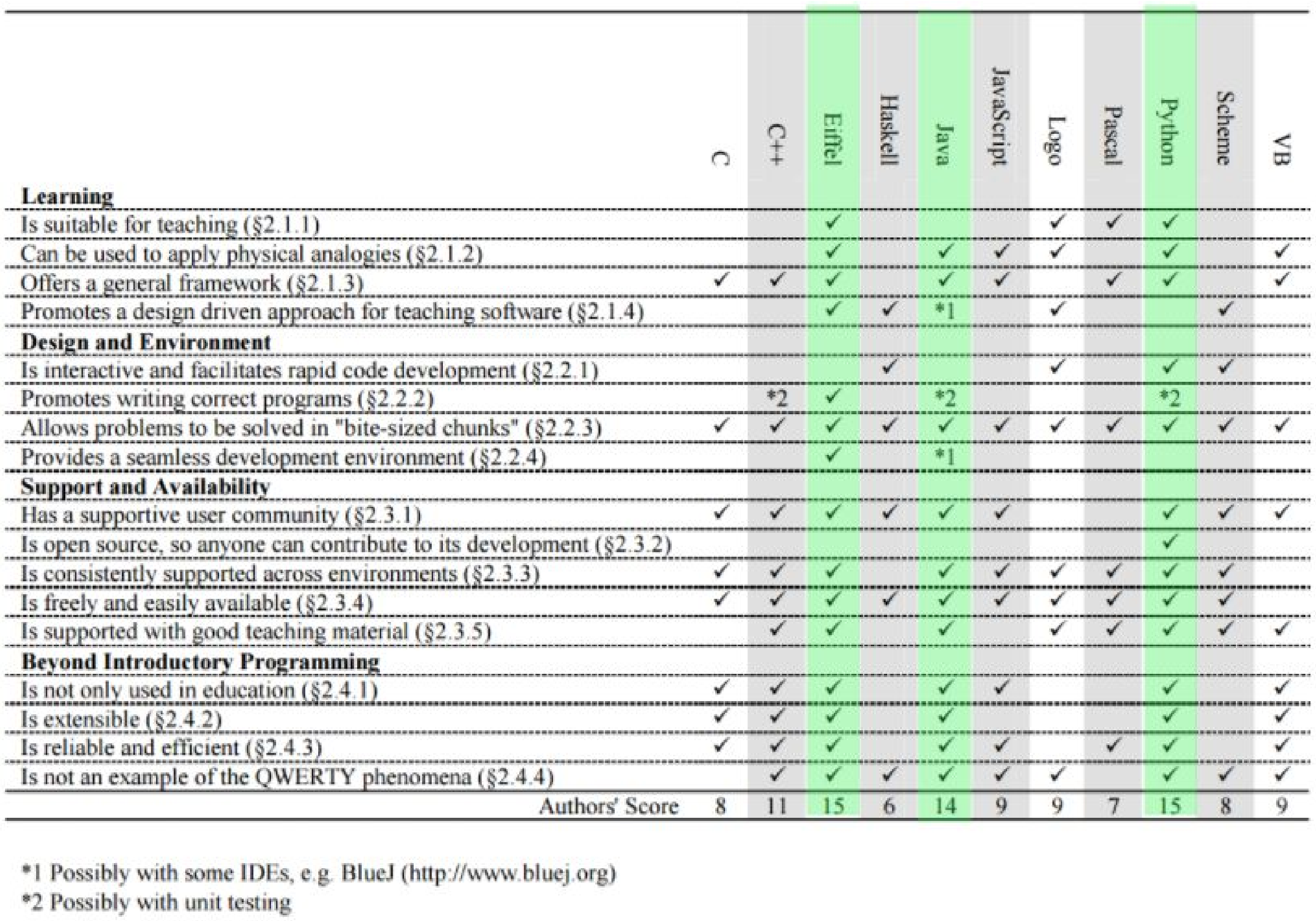
Notably, their own research on the topic before the study began indicated the following:

Studies in science, technology, and math (STM) education indicate that development of identity can be problematic for many students, and that they struggle to integrate their study experiences with their perception of who they are and want to become. Identity has also been found critical for retention, and is an important underexplored research area. [2]

Their results emphasize that all of the interviewees indicate some form of creation to be a primary motivation to continue studying CS/IT. It was important to the focus group to be able to feel that they had created something of their own [2].

This language-independent feature of a course [creation] is one that may be overlooked in even the small group of introductory programming courses we will examine later.

A 2006 study took many factors into consideration with respect to language choice for an introductory programming course. Their results are summarized graphically in Figure 14:



	C	C++	Eiffel	Haskell	Java	JavaScript	Logo	Pascal	Python	Scheme	VB
Learning											
Is suitable for teaching (§2.1.1)			✓				✓	✓	✓		
Can be used to apply physical analogies (§2.1.2)			✓		✓	✓	✓		✓		✓
Offers a general framework (§2.1.3)	✓	✓	✓		✓	✓		✓	✓		✓
Promotes a design driven approach for teaching software (§2.1.4)			✓	✓	*1		✓			✓	
Design and Environment											
Is interactive and facilitates rapid code development (§2.2.1)				✓			✓		✓	✓	
Promotes writing correct programs (§2.2.2)		*2	✓		*2				*2		
Allows problems to be solved in "bite-sized chunks" (§2.2.3)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Provides a seamless development environment (§2.2.4)			✓		*1						
Support and Availability											
Has a supportive user community (§2.3.1)	✓	✓	✓	✓	✓	✓			✓	✓	✓
Is open source, so anyone can contribute to its development (§2.3.2)									✓		
Is consistently supported across environments (§2.3.3)	✓	✓	✓		✓	✓	✓	✓	✓	✓	
Is freely and easily available (§2.3.4)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
Is supported with good teaching material (§2.3.5)		✓	✓		✓		✓	✓	✓	✓	✓
Beyond Introductory Programming											
Is not only used in education (§2.4.1)	✓	✓	✓		✓	✓			✓		✓
Is extensible (§2.4.2)	✓	✓	✓		✓				✓		✓
Is reliable and efficient (§2.4.3)	✓	✓	✓		✓	✓		✓	✓		✓
Is not an example of the QWERTY phenomena (§2.4.4)		✓	✓	✓	✓	✓	✓		✓	✓	✓
Authors' Score	8	11	15	6	14	9	9	7	15	8	9

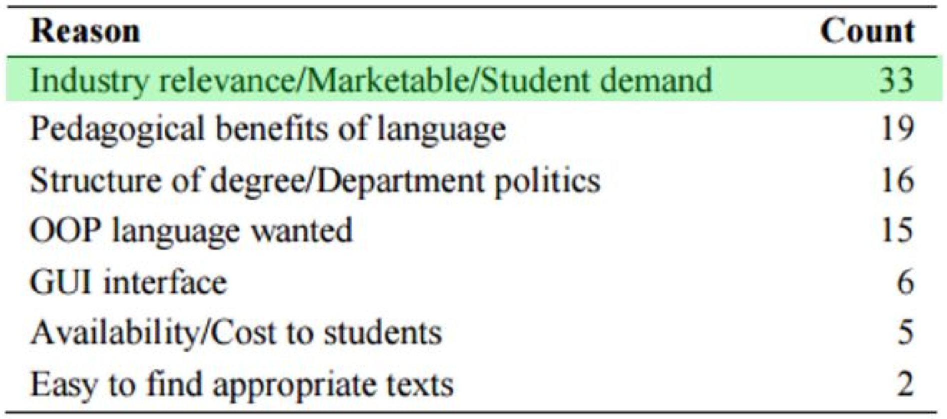
*1 Possibly with some IDEs, e.g. BlueJ (http://www.bluej.org)
*2 Possibly with unit testing

Figure 14: Programming Language evaluation by feature set [3]

By providing well founded criteria, this study has attempted to provide objectivity into what has been, up to now, frequently an emotive argument. [3]

There is palpable frustration in the researchers' above statement. Based on their study, Eiffel and Python are the best language candidates for an introductory course with Java following closely behind.

Before conducting this study, the researchers examined "a census of introductory programming courses within Australia and New Zealand" to understand what instructors look for when choosing a language to teach their courses. **Industry relevance** was the most common factor. A summary of the responses is available in Figure 15.



Reason	Count
Industry relevance/Marketable/Student demand	33
Pedagogical benefits of language	19
Structure of degree/Department politics	16
OOP language wanted	15
GUI interface	6
Availability/Cost to students	5
Easy to find appropriate texts	2

Figure 15: Instructors' most influential factors [3]

This study also cited the following as "the primary objective of introductory programming instruction":

to nurture novice programmers who can apply programming concepts equally well in any language.

Why, then, is so much effort placed on finding the perfect language? Why not **teach multiple languages** throughout a semester? The authors also wondered this and had the follow to say on the matter:

. . . many papers from literature argue that one language is superior for this task. Such research asserts a particular language is superior to another because, in isolation, it possesses desirable features or because changing to the new language seemed to encourage better results from students. What is shown in literature is surely only a reflection of the innumerable debates that have undoubtedly taken place within teaching institutions. [3]

A multi-language curriculum argument was not one I could find existing research on, but later on we will find out how important knowing multiple languages is for industry relevance and how many courses follow this paradigm.

A year later a study was published that attempted to summarize all existing research on the subject of introductory programming. Their goal was to provide a resource for instructors to reference when deciding how to construct their curriculum by examining existing curricula, pedagogy, language and teaching tool choices [4].

The study cited over one-hundred sources and included most notable source material in the appendix, including an excerpt for each cited study's purpose.

Surprisingly, they made the following remark in their conclusion:

We conclude that despite the large volume of literature in this area, there is little systematic evidence to support any particular approach. [4]

The title begins with "A Survey of Literature" and they later mention that the main purpose for making this assertion is so that their research provides guidance to educators partaking in curriculum design rather than "a canonical answer to the question of how to teach introductory programming." [4]

The study cites **C**, **Java**, and **C++** as consistent members of the "top four languages" club in both industry and education. Language popularity alone is not warrant enough to declare a winner, though, and the researchers had the following to say on this matter:

Despite the popularity of languages such as Java, C and C++, there has been much debate about the suitability of these languages for education, especially when introducing programming to novices (for example [66, 47, 12, 22, 23]). These languages have not been designed specifically for educational purposes, in contrast to others that have been designed with this specific purpose in mind (e.g., Python, Logo, Eiffel, Pascal). [4]

Knowing other languages were designed with **learnability as a feature** is highly compelling. Although only Python appears to belong to this category in addition to being one of the most popular languages, it remains to be seen whether the other languages are featured in the curricula of the courses under examination in the next section.

The final research article to be examined comes from early 2016 and incidentally supports this paper's warrant in its abstract:

The rapid integration of technology into our professional and personal lives has left many education systems ill-equipped to deal with the influx of people seeking computing education. [5]

This study sought to experimentally prove that incorporating subgoals—defined as "components of complex problem solutions that provide functional pieces of the final solution" [5]—allows students to perform better "while solving novel programming problems" than students carrying out the same tasks through conventional means. They found that across all assessments, the subgoal focus groups completed 36% more of the problem-solving tasks correctly. The participants were chosen so that no one had taken more than one computer programming course, or had any experience with the tool (Android App Inventor) they used to conduct the experiment.

Summary of Findings - Academic Research

One study examined a multitude of languages and scored them based on a set of criteria [3], but a later and more thorough study pointed out that making a recommendation on programming language choice may not be the ideal approach to creating the most effective introductory programming curriculum [4]. The same study revealed a discrepancy in language popularity and learnability— the most popular and most widely used languages in industry are not necessarily the most logical candidates for introductory education [4].

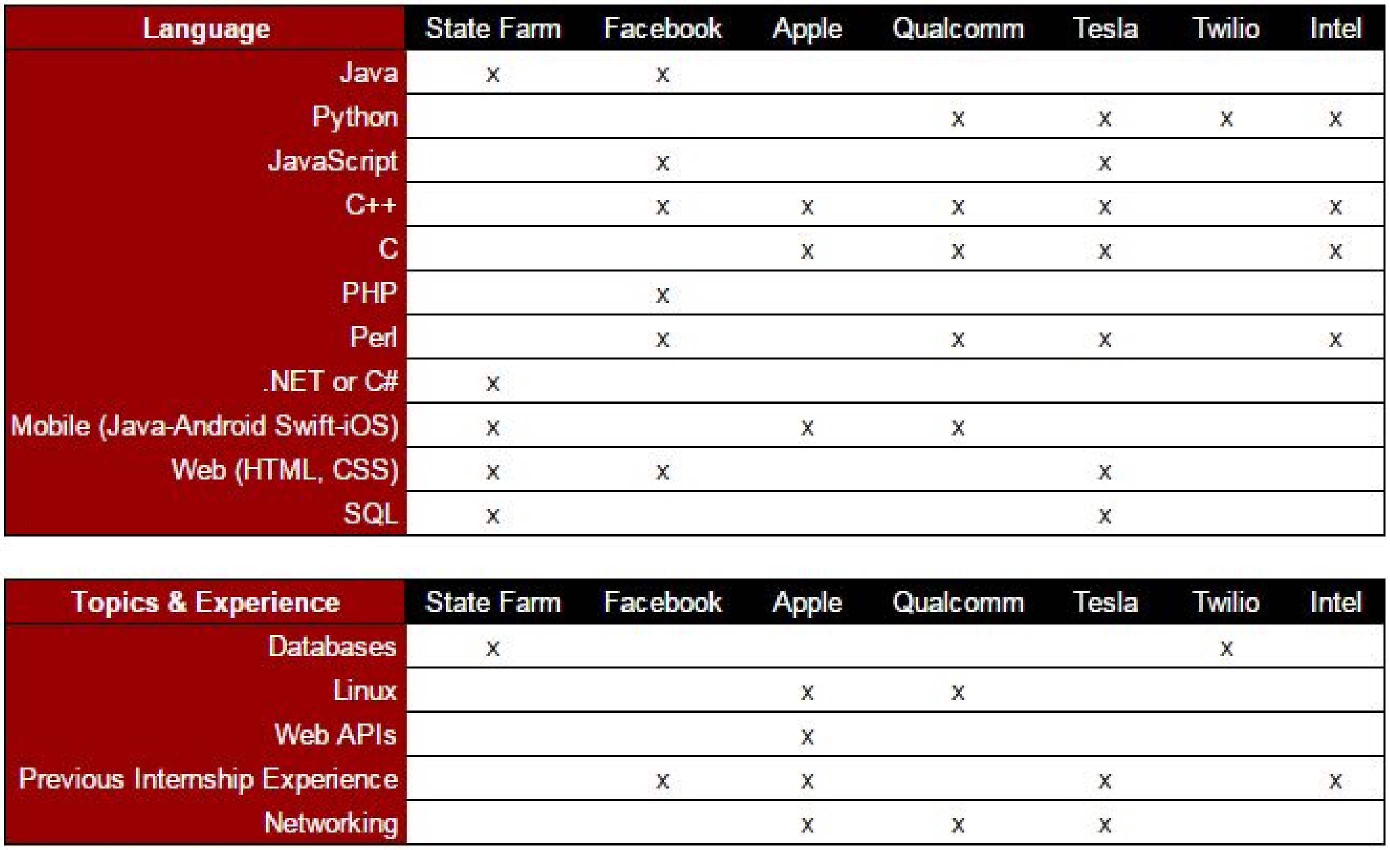
Most of the studies testified that the subjectivity and bias can be hard to eliminate at individual institutions but concur that **industry relevance** [1-5] and **student engagement that promotes a sense of accomplishment** [1-2,4-5] are crucial to a successful course.

Next, we will gather as much information as we can to define industry relevance explicitly. This is necessary to know if current courses satisfy this most important characteristic as well as for making a generic recommendation for an ideal course.

2. Real World Demand

Jobs found through intern.supply. A lot of applications are seasonal and therefore closed for the upcoming Summer, so information could not be found for: Google, Microsoft, SpaceX, GitHub, HP. LinkedIn and more.

Figure 15 is a summary of my findings when observing the job descriptions for Software Engineering internships at State Farm, Facebook, Apple, Qualcomm, Tesla, Twilio, and Intel:



Language	State Farm	Facebook	Apple	Qualcomm	Tesla	Twilio	Intel
Java	x	x					
Python				x	x	x	x
JavaScript		x			x		
C++		x	x	x	x		x
C			x	x	x		x
PHP		x					
Perl		x		x	x		x
.NET or C#	x						
Mobile (Java-Android Swift-iOS)	x		x	x			
Web (HTML, CSS)	x	x			x		
SQL	x				x		

Topics & Experience	State Farm	Facebook	Apple	Qualcomm	Tesla	Twilio	Intel
Databases	x					x	
Linux			x	x			
Web APIs			x				
Previous Internship Experience		x	x		x		x
Networking			x	x	x		

Figure 15: Aggregate Data [35-41]

In most cases, companies list **multiple languages** as **minimum requirements** to be considered as an applicant for their advertised position [35-41].

Each job listing also had requirements that no other listing had, so this representation of data is not exhaustive by any measure.

Figure 16 displays the number of times a language was included as a requirement out of the seven job listings analyzed.

Figure 17 shows how frequently some of the most common topics and experience requirements were listed in the sample set.

Next, real, collegiate, introductory programming courses will be examined to have concrete evidence as to how the educational infrastructure is employing the researched strategies to prepare students for the field.

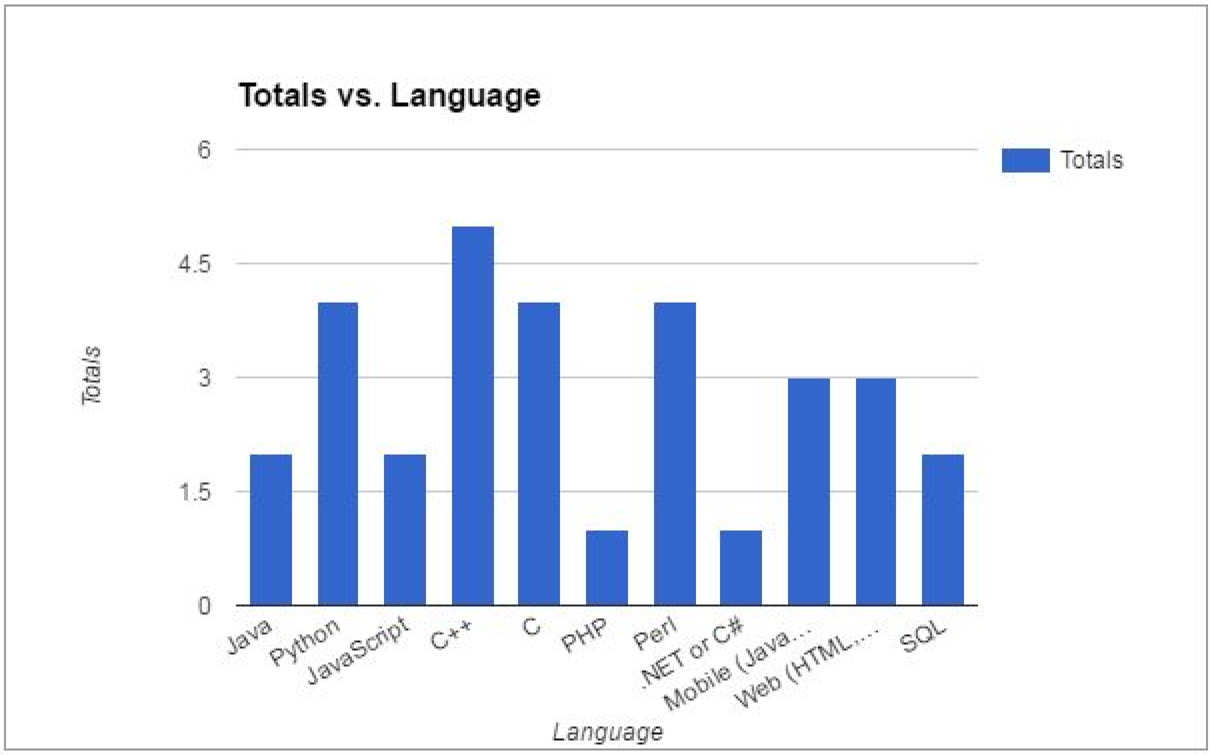


Figure 16: Language Totals [35-41]

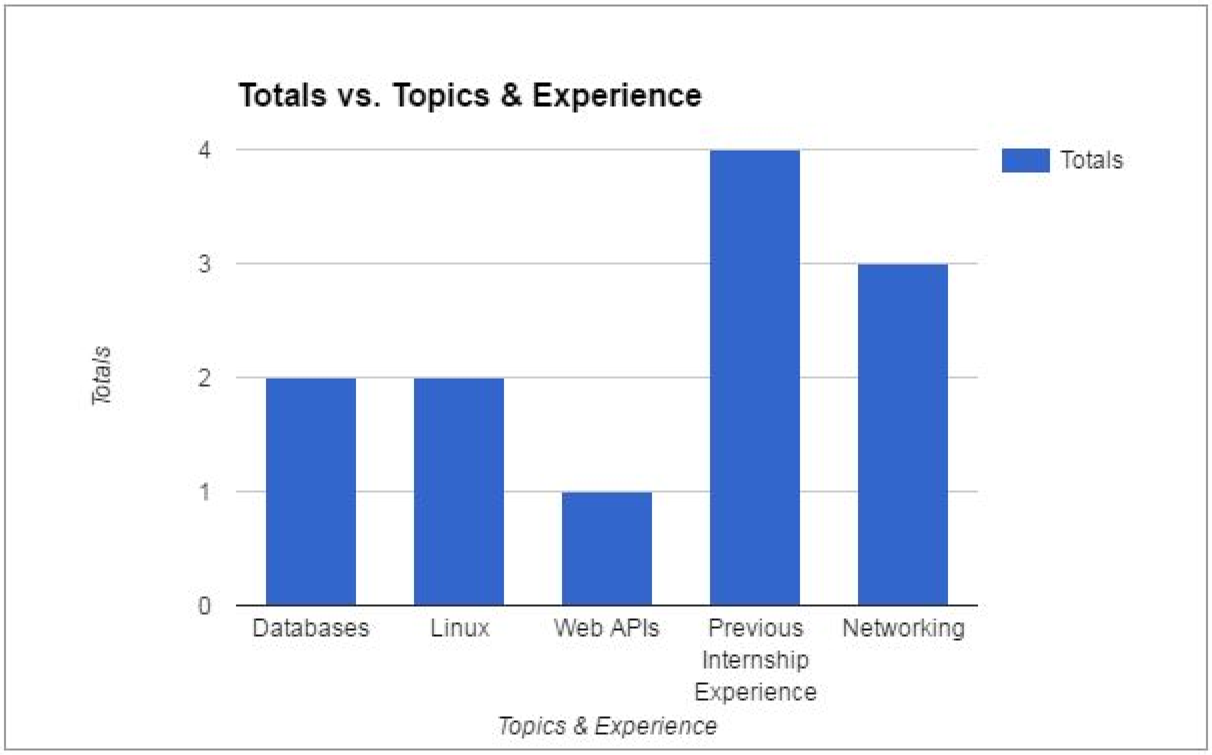


Figure 17: Topics & Experience Totals [35-41]

3. Modern Practices

How is introductory CS being taught, and what is the scope of a modern introductory CS course? Six examples have been reviewed to answer this question:

AP Computer Science A, Fall 2014 [26]

Language: Java

Justification:

> *Since any natural language (e.g., English) allows inconsistencies and ambiguities, solutions in computer science require a communication medium more formal than a natural language. For this reason, the AP Computer Science A course requires that potential solutions of problems be written in the Java programming language.*

Topics: Object-Oriented Program Design, Program Implementation, Program Analysis, Standard Data Structures, Standard Operations and Algorithms, Computing in Context.

Takeaway: This course is designed with preparing students for university in mind. They focus on enabling students to learn how to use programming to solve problems. Avoiding topics of computer architecture and hardware is appropriate here as part of their motivation is to get students excited about CS.

[UW-Madison] COMP SCI 301: Introduction to Data Programming [27]

Language: Python

Justification: Not found on course guide or in class syllabus.

Topics: Instruction and experience in the use of a programming language for beginners, program design, development of good programming style.

Takeaway: Course for students not pursuing CS as a major. Focuses on problem solving using programming and caters to the computer illiterate (not an insult, CS50 employs a similar strategy) by encouraging use of a web-based tool to execute code.

[UW-Madison] COMP SCI 302: Introduction to Programming [28]

Language: Java

Justification: Java is widely used to write programs for all kinds of computers to run.

Topics: analyze problems and formulate algorithms; create robust, user-friendly, well-structured and well-documented Java programs; read basic Java programs to determine their purpose; and have a basic understanding of how computers work.

Takeaway: Introductory course for CS majors. This class focuses on Java executed within Eclipse IDE and charges students $67 dollars out of pocket to complete homework from an online textbook.

[MIT] 6.00SC: Introduction to Computer Science and Programming [29]

Language: Python

Justification: language for expressing computations.

Topics: learning about the process of writing and debugging a program, learning about the process of moving from a problem statement to a computational formulation of a method for solving the problem, L learning a basic set of "recipes"—algorithms, learning how to use simulations to shed light on problems that don't easily succumb to closed form solutions, learning about how to use computational tools to help model and understand data.

Takeaway: emphasis on competence for seeking a job in any field as this course serves as an introduction for students who do and do not intend to major in CS.

[Harvard] CS50 [30]

Languages: Scratch, C, Python, JavaScript (all in one semester!)

Justification:

> *Insofar as CS50 is not only an introduction to the intellectual enterprises of computer science but also the art of programming, we introduce students along the way to a number of languages so that they exit the course not having only learned X, where X is some language, but having learned how to program (procedurally). Since 2007 have we first introduced students to Scratch, thereafter spending much of the semester in C, introducing students toward term's end to PHP, SQL, and JavaScript (plus HTML and CSS).*

Topics: Scratch, C, Arrays, Algorithms, Memory, Data Structures, HTTP, Machine Learning, Python, SQL and JavaScript.

Takeaway: Exposure to many topics instead of mastering one language is a seemingly unorthodox approach to introducing students to computing but shows the institution's willingness to innovate.

[Stanford] Computer Science 101 (Self-Paced) [34]

Language: JavaScript

Justification:

> *CS101 uses a variant of Javascript. However, the code used in CS101 is very stripped down, avoiding all sorts of boilerplate that would get in the way of learning. As a result, CS101 code does not look like full, professional Javascript code.*

Topics: The nature of computers and code, what they can and cannot do; how computer hardware works: chips, cpu, memory, disk; necessary jargon: bits, bytes, megabytes, gigabytes; how software works: what is a program, what is "running"; how digital images work; computer code: loops and logic; big ideas: abstraction, logic, bugs; how structured data works; how the internet works: ip address, routing, ethernet, wi-fi; computer security: viruses, trojans, and passwords; analog vs. digital; digital media, images, sounds, video, compression.

Takeaway: This free course available online acknowledges the breadth of computer related topics and caters to a zero-prior-experience audience.

Summary of Findings - Existing Courses

Stanford and Harvard follow a new paradigm: online and free to enroll or enrollment available to general public, emphasis on breadth of coverage versus depth of programming skill in one language.

MIT directly mentions awareness of preparing students for a job while UW-Madison fails to provide any warrant for their decisions in curriculum implementation and has by far the least amount of public information regarding the curriculum.

Notably, the variance in curriculum coverage is seemingly more volatile than one would expect in other STEM related courses. First, second and third semester calculus are much more likely to cover similar topics than introductory CS school-to-school, but the academic research provides a number of explanations for this.

Stanford's course focuses the least on programming tasks, and this approach aligns with the importance of a student's interpretation of what working in the field entails detailed in the Uppsala Computing Education Research Group's study.

This section concludes the analysis.

Companies also list many topics as minimum requirements that are not covered in introductory CS courses [26-30, 34], which leaves a lot of content to be covered in the next few courses.

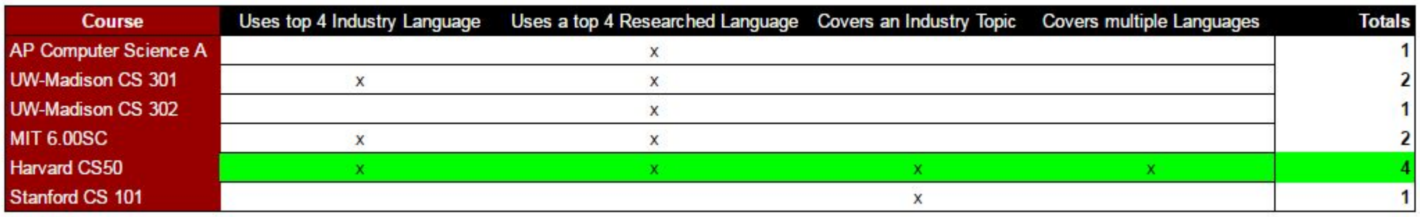
Conclusion

Computer programming is a skill that has a rapidly increasing demand that our current educational infrastructure is having difficulty meeting [6,15]. Every year, there are fewer people graduating with programming-related degrees of varying levels than there are job openings for people with these skills [15].

Academic research comments on the difficulty of selecting programming languages for introductory CS courses. There are exceptions to this, but some research undermines the very idea of evaluating languages at all because common languages aren't necessarily the best languages for education.

Because of this, I call upon my other two paths of analysis to recommend that a course should begin with a simpler, educationally-oriented language such as Python or JavaScript but should be followed up with C. If object-orientation is a desired programming paradigm, a course could begin in Java and eventually move to C++, though I find this to be less ideal as object orientation is not widely used (Java is not a top four industry language).

Even in a small sample size of courses and job descriptions, the mismatch of content being covered and requirements for candidates is surprising. Figure 18 represents my choice for best course and the criteria I used to draw that conclusion:



Course	Uses top 4 Industry Language	Uses a top 4 Researched Language	Covers an Industry Topic	Covers multiple Languages	Totals
AP Computer Science A		x			1
UW-Madison CS 301	x	x			2
UW-Madison CS 302		x			1
MIT 6.00SC	x	x			2
Harvard CS50	x	x	x	x	4
Stanford CS 101			x		1

Figure 18: Course Evaluation Results

Academic researchers would agree that Harvard's CS50 is among the nation's current best introductory computer sciences courses, and if research could be conducted that attempts to evaluate the effectiveness of each school's overall curricula, it may be possible to make less generic recommendations on how to design a course using inductive reasoning (in other words, I admit that my recommendation is a gross oversimplification).

Hopefully future research takes such an approach so the field of computing can continue to advance and innovate, giving new students more of a fighting chance to navigate the vast ocean of information and contribute to work being done at academic institutions and private tech companies alike.

References

[1] P. Higgins and N. Brophy, “Programming Languages and Teaching”, *DCU School of Computing*, p. 4, circa 2000 [Online], Available: http://computing.dcu.ie/. [Accessed Feb. 9, 2017].

[2] A. Peters, A. Berglund, A. Eckerdal, A. Pears,, “First Year Computer Science and IT Students’ Experience of Participation in the Discipline”, *2014 International Conference on Teaching and Learning in Computing and Engineering*, pp. 1-8, 11-13 April 2014. Available: IEEE Xplore, http://ieeexplore.ieee.org. [Accessed: Mar. 22, 2017].

[3] L. Mannila, M. de Raadt, “An Objective Comparison of Languages for Teaching Introductory Programming”, *Koli Calling 2006 Proceedings*, pp. 32-37, Feb. 2006. Available: ACM Digital Library, http://dl.acm.org. [Accessed: Feb. 9, 2017] .

[4] A. Pears, S. Seidman, L. Malmi, L. Mannila, E. Adams, J. Bennedsen, M. Devlin, J. Paterson, “A Survey of Literature on the Teaching of Introductory Programming”, *Innovation and Technology in Computer Science Education Working Group Reports Proceedings* also *ACM SIGCSE Bulletin*, vol. 39, no. 4, pp. 204-223, Dec. 2007. Available: ACM Digital Library, http://dl.acm.org. [Accessed: Mar. 5, 2017].

[5] L. Margulieux, R. Catrambone, M. Guzdial, “Employing subgoals in computer programming education”, *Computer Science Education*, vol. 26, no. 1, pp. 44-67, Jan. 2016. Available: Taylor Francis Online, http://www.tandfonline.com. [Accessed: Mar. 21, 2017].

[6] A. Carnevale, N. Smith and M. Melton, “Part 2: What is STEM?”, *STEM: Science Technology Engineering Mathematics*, pp. 18-26, Oct. 20, 2011. [Online]. Available: https://cew.georgetown.edu/cew-reports/stem/. [Accessed: Mar. 21, 2017].

[7] N. Emberton, "When was the first computer invented?", *computerhope.com*, no date. [Online]. Available: http://www.computerhope.com/issues/ch000984.htm. [Accessed: Feb. 9, 2017].

[8] J. Hruska, "Intel Core i7-4790K", *pcmag.com*, Jul. 31, 2014. [Online]. Available: http://www.pcmag.com/article2/0,2817,2461705,00.asp. [Accessed: Feb. 9, 2017].

[9] P. Zandbergen, "What is a Motherboard? - Definition, Function & Diagram", *study.com*, no date. [Online]. Available: http://study.com/academy/lesson/what-is-a-motherboard-definition-function-diagram.html. [Accessed: Feb. 9, 2017].

[10] G. Carrette, "Linkedin Profile Page - George Carrette", no date. [Online]. Available: https://www.linkedin.com/in/gjcarrette/. [Accessed: Mar. 21, 2017].

[11] A. Carnevale, N. Smith and J. Strohl, "Recovery: Job Growth and Education Requirements Through 2020", *Georgetown Public Policy Institute*, pp. 65-69, June 26, 2013. [Online]. Available: https://cew.georgetown.edu/cew-reports/recovery-job-growth-and-education-requirements-throug-2020/#full-report. [Accessed: Mar. 21, 2017].

[12] College Board, "About Us", *collegeboard.org*, no date. [Online]. Available: https://www.collegeboard.org/about. [Accessed: Mar. 21, 2017].

[13] College Board, “Program Summary Report”, *AP Program Participation and Performance Data 2016*, circa 2016. [Online]. Available: https://research.collegeboard.org/programs/ap/data/participation/ap-2016. [Accessed: Mar. 21, 2017].

[14] National Center for Education Statistics, "Back to school statistics", *nces.ed.gov*, no date. [Online]. Available: https://nces.ed.gov/fastfacts/display.asp?id=372. [Accessed: Mar. 21, 2017].

[15] Exploring Computer Science, "CS Education Statistics", *exploringcs.org*, no date. [Online]. Available: http://www.exploringcs.org/resources/cs-statistics. [Accessed: Mar. 21, 2017].

[16] K. Mattern, E. Shaw, M. Ewing, "Advanced Placement Exam Participation: Is AP Exam Participation and Performance Related to Choice of College Major?", College Board, 2011. [Online]. Available: https://research.collegeboard.org. [Accessed: Mar. 21, 2017].

[17] AMD64 Technology, *AMD64 Architecture Programmer's Manual Volume 1: Application Programming*, Advanced Micro Devices, 2013. [Online]. Available: https://support.amd.com/TechDocs/24592.pdf [Accessed: Mar. 21, 2017].

[18] C. Zapponi, "A Small Place to Discover Languages in GitHub", *githut.info*, 2014. [Online]. Available: http://githut.info/. [Accessed: Mar. 21, 2017].

[19] Wikipedia contributors, "JavaScript", *wikipedia.org*, 2017. [Online]. Available: https://en.wikipedia.org/wiki/JavaScript. [Accessed: Apr. 6, 2017].

[20] Wikipedia contributors, "Java (programming language)", *wikipedia.org*, 2017. [Online]. Available: https://en.wikipedia.org/wiki/Java_(programming_language). [Accessed: Apr. 6, 2017].

[21] Wikipedia contributors, "Python (programming language)", *wikipedia.org*, 2017. [Online]. Available: https://en.wikipedia.org/wiki/Python_(programming_language). [Accessed: Apr. 6, 2017].

[22] Wikipedia contributors, "PHP", *wikipedia.org*, 2017. [Online]. Available: https://en.wikipedia.org/wiki/PHP. [Accessed: Apr. 6, 2017].

[23] Wikipedia contributors, "Ruby (programming language)", *wikipedia.org*, 2017. [Online]. Available: https://en.wikipedia.org/wiki/Ruby_(programming_language). [Accessed: Apr. 6, 2017].

[24] Wikipedia contributors, "C++", *wikipedia.org*, 2017. [Online]. Available: https://en.wikipedia.org/wiki/C%2B%2B. [Accessed: Apr. 6, 2017].

[25] Wikipedia contributors, "C (programming language)", *wikipedia.org*, 2017. [Online]. Available: https://en.wikipedia.org/wiki/C_(programming_language). [Accessed: Apr. 6, 2017].

[26] College Board AP, "Computer Science A: Course Description Effective Fall 2014", *apstudent.collegeboard.org*, 2014. [Online]. Available: https://apstudent.collegeboard.org/apcourse/ap-computer-science-a/course-details. [Accessed: Mar. 21, 2017]

[27] University of Wisconsin-Madison Course Guide, "Comp Sci 301: Introduction to Data Programming", *cs.wisc.edu*, Apr. 6, 2017. [Online]. Available: https://pubs.wisc.edu/ug/ls_compsci.htm. [Accessed: Apr. 6, 2017].

[28] University of Wisconsin-Madison Course Guide, "Comp Sci 302: Introduction to Programming", *cs.wisc.edu*, Apr. 6, 2017. [Online]. Available: https://pubs.wisc.edu/ug/ls_compsci.htm. [Accessed: Apr. 6, 2017].

[29] Massachusetts Institute of Technology, "Introduction to Computer Science and Programming", *ocw.mit.edu*, 2011. [Online]. Available: https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-00sc-introduction-to-computer-science-and-programming-spring-2011/. [Accessed: Mar. 21, 2017].

[30] Harvard University, "CS50: Introduction to Computer Science", *cs50.harvard.edu*, 2017. [Online]. Available: http://docs.cs50.net/2016/fall/syllabus/cs50.html. [Accessed: Apr. 6, 2017].

[31] C. Hock-Chuan, "GCC and Make: Compiling, Linking and Building C/C++ Applications", *ntu.edu.sg*, Apr. 2013. [Online]. Available: https://www3.ntu.edu.sg/home/ehchua/programming/cpp/gcc_make.html. [Accessed: Apr. 6, 2017].

[32] R. Tiwari, "RISC Vs CISC, Harvard v/s Van Neumann", *LinkedIn SlideShare*, Jun. 19, 2015. Available: https://www.slideshare.net/RavikumarTiwari1/risc-vs-cisc-harvard-vs-van-neumann. [Accessed: Mar. 21, 2017].

[33] C. Traver, "C Language Programming", LinkedIn SlideShare, Sep. 8, 2014. Available: https://www.slideshare.net/29vaibhav/c-language-programming-38812176. [Accessed: Mar. 21, 2017].

[34] N. Parlante - Stanford University, "Computer Science 101 (Self-Paced)", *online.stanford.edu*, Jul. 15, 2014. Available: http://online.stanford.edu/course/computer-science-101-self-paced. [Accessed: Apr. 6, 2017].

[35] State Farm, "Become an Intern: IT/Systems – IT Software Developer Intern", *statefarm.com*, 2017. [Online]. Available: https://www.statefarm.com/careers/become-an-intern/IT-internships/software-developer-internships. [Accessed: Apr. 7, 2017].

[36] Facebook, "Internship - Engineering, Tech & Design: Software Engineer, Intern/Co-op", *facebook.com*, 2017. [Online]. Available: https://www.facebook.com/careers/jobs/a0I1200000IAGYKEA5/. [Accessed: Apr. 7, 2017].

[37] Apple, "Jobs at Apple: Software Engineering Intern: Core OS", *jobs.apple.com*, 2017. [Online]. Available: https://jobs.apple.com/us/search?jobFunction=CGINT#&t=0&sb=req_open_dt&so=1&j=CGINT&lo=0*USA&pN=0&openJobId=35799849. [Accessed: Apr. 7, 2017].

[38] Qualcomm, "Job Detail: Software Engineering Internships @ Qualcomm (Summer 2017)", *jobs.qualcomm.com*, 2017. [Online]. Available: https://jobs.qualcomm.com/public/jobDetails.xhtml?requisitionId=1946651. [Accessed: Apr. 7, 2017].

[39] Tesla, "Careers: Software Engineering Internship (Summer 2017)", *tesla.com/careers*, 2017. [Online]. Available: https://www.tesla.com/careers/job/software-engineeringinternshipsummer2017-45545. [Accessed: Apr. 7, 2017].

[40] twilio, "Software Engineering Intern (Summer 2017)", *boards.greenhouse.io/twilio/jobs*, 2017. [Online]. Available: https://boards.greenhouse.io/twilio/jobs/276983#.WOfpjGnyuUl. [Accessed: Apr. 7, 2017].

[41] Intel, "Software Intern", *jobs.intel.com*, 2017. [Online]. Available: http://jobs.intel.com/ShowJob/Id/1170951/Software-Intern/. [Accessed: Apr. 7, 2017].

[42] Intern Supply, "How to Know When You're Ready to Apply", *intern.supply*, no date. [Online]. Available: http://www.intern.supply/how-to-know-if-youre-ready-to-apply.html. [Accessed: Apr. 7, 2017].

Appendix I: High Level Language Summary

Languages: Usage

GitHut.info is a website that analyzes the code behind over *two million* projects hosted on **GitHub**—another website that provides free cloud storage for anyone seeking to store their code repositories in a central location with the ability to keep track of version and workflow using **git**, a widely used and free version control piece of software.

According to their dataset, the eight most commonly used languages on GitHub are (in order of most frequent to least frequently used) **JavaScript, Java, Python, CSS, PHP, Ruby, C++ and C** [18]. In this case the metric is total number of projects, not number of total lines of code (not a metric they track) or number of contributions per project. This information is echoed in Figure 13.

Looking at the highest average number of contributions per project, the top eight languages are (from most to least) C++, TeX, Rust, C, CSS, Scala, JavaScript, and Java.

Not all software is public on GitHub, and not all of these languages are “programming languages” where a programming language is defined as a language that you can use to create a file that can be executed by the operating system. For instance, CSS is short for cascading style-sheets and is used to style web pages and TeX is used to prepare documents such as a book or article submitted to a research journal (especially where mathematical formula will be used frequently).

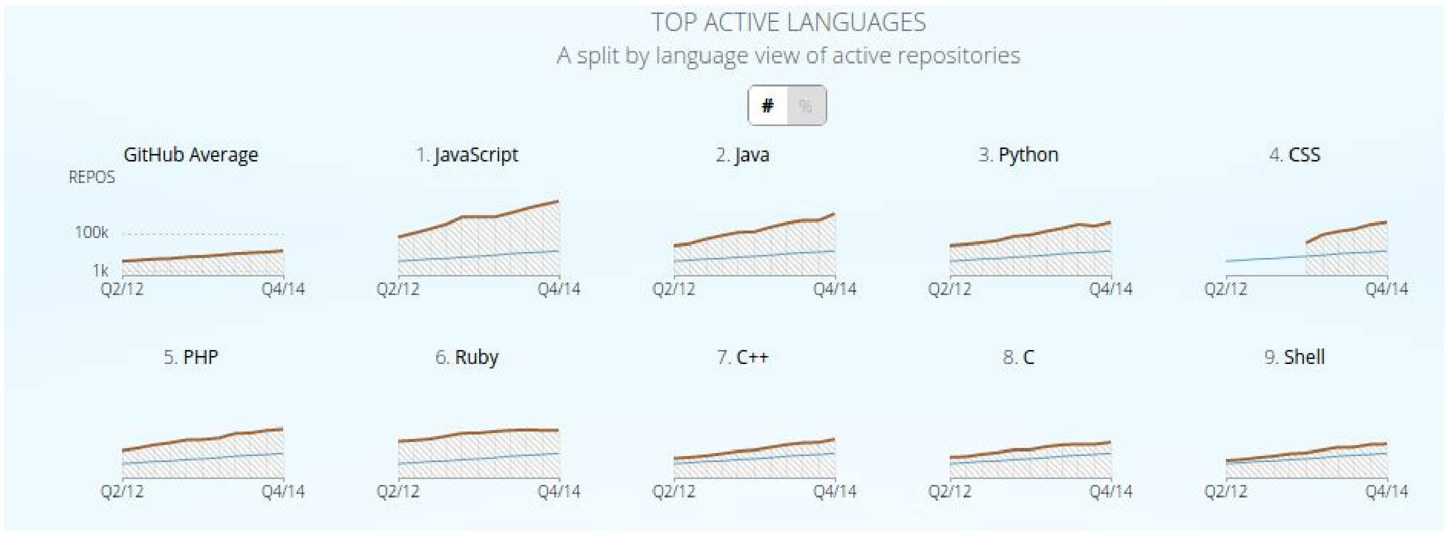


Figure 13: Top Active Langauges from GitHut.info

Languages: Under the Hood

Optionally review the following summary of the popular languages, this will serve as context in the following discussion of the introductory CS educational strategies:

JavaScript

First created in 1995, interpreted language, most interpreters are written in C/C++. One of the three core technologies in World Wide Web content production, not based on Java despite name. [19]

Java

First created in 1995, compiles to Java bytecode that is interpreted by a Java virtual machine (JVM) written in C (Sun JVM) or C++ (Oracle JVM). Created to provide a means for developers to "write once, run anywhere" because JVMs can run on any Java virtual machine regardless of CPU architecture. [20]

Python

First created in 1991, interpreted language, interpreter written in C. Design emphasizes code readability and concision. [21]

PHP

First created in 1995, interpreted language, interpreter written in C. Server-side scripting language designed primarily for web development, can still be used as a general purpose programming language. [22]

Ruby

First created in 1995, compiles to bytecode that is interpreted by an interpreter written in C. General purpose programming language. [23]

C++

First created in 1983, compiled language, standardized by the International Organization for Standardization. General purpose programming language. [24]

C

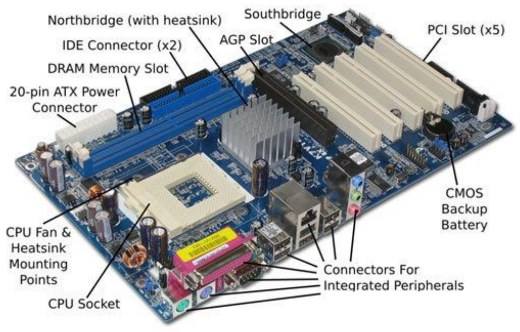
First created in 1972, compiled language standardized by both the American National Standards Institute and the International Organization for Standardization. General purpose programming language. [25]

Appendix II: What Computers Are and How They Work

Figure 6: Intel CPU front and back, 2013 [8]

Computer systems in the broadest sense provide infrastructure to our lives: commerce, transportation, energy and more. Before an aspiring computer scientist or programmer is prepared to assist in the development of large-scale or meaningful systems, training and education must take place in an environment with only a single device that is executing code.

Most commonly the device in question is a personal computer, where the physical hardware executing code is a CPU (central processing unit) that implements AMD64 (commonly x86-64 or just x64) instruction set architecture (Figure 6). The instruction set architecture is a standard that describes the expected behavior of the CPU when executing each different instruction, where an instruction is a binary number that may be 16, 32, or 64 bits long and where a bit is a single binary digit [17].


Northbridge (with heatsink)
Southbridge
IDE Connector (x2)
AGP Slot
PCI Slot (x5)
DRAM Memory Slot
20-pin ATX Power Connector
CMOS Backup Battery
CPU Fan & Heatsink Mounting Points
CPU Socket
Connectors For Integrated Peripherals
Figure 7: Motherboard with components labeled

But what about peripherals like mice, keyboards, monitors and audio equipment? With only the ability to work with numbers, how are we able to interact with and control these physical mediums? Digital electronics solve this problem by additionally defining specifications for physical interfaces that peripheral hardware must use to connect to and communicate with other hardware. USB is one example of such an interface, and the USB standard defines both the type of connector to use as well as the rules for communicating over the physical cable. Another requirement for achieving advanced functionality is the ability to store information both temporarily and permanently. We refer to the hardware performing these tasks as memory or storage. A printed-circuit board (PCB) is often the item that each of these components connects to—temporarily or secured firmly in place in the case of the CPU and USB device—and permanently in the case of passive components such as capacitors and resistors (Figure 7).

The CPU has hundreds of gold pads or pins on its underside that connect to the motherboard, and the motherboard contains a dozen layers of thin copper traces (wires) that connect the CPU to the rest of the devices on board.

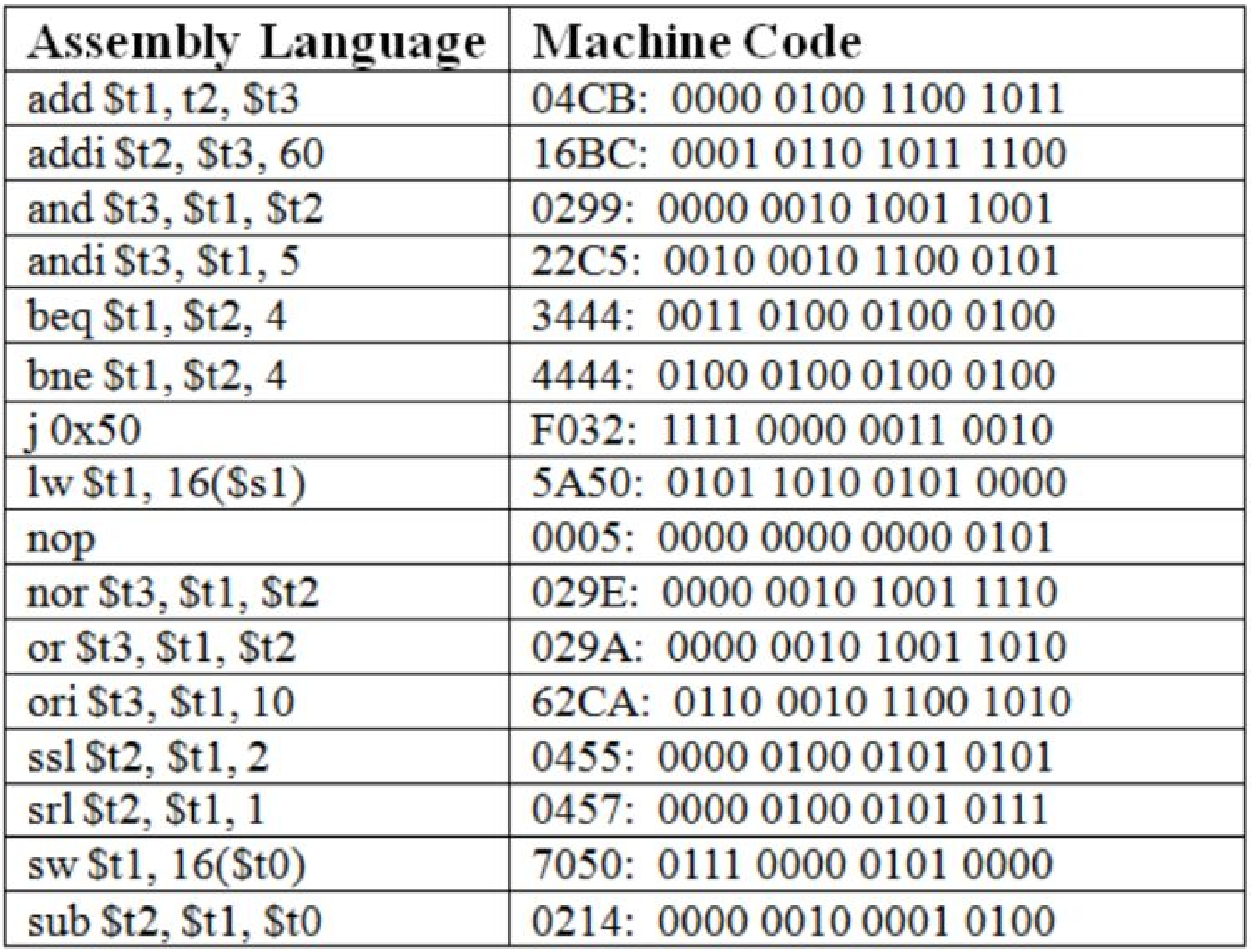
To better understand how a computer functions let's look at an example "program" or set of instructions organized to accomplish a specific task. These instructions would be stored in the CPU's internal memory after the CPU fetches it from another form of permanent storage or memory, and memory is organized so that groups of 8 bits, called a byte, are "individually addressable" where an address is 32 or more bits wide so that there are at least 2^{32} unique locations. For scale, a memory that has 32 bit memory addresses can hold a maximum of 4 gigabytes of information. Notice that 2^{32} = 4,294,967,296—the "giga" prefix does not follow standard SI conventions. Because computers use binary numbering instead of decimal, it makes more sense to group numbers by 2^{10} (1,024) than 10^3 (1,000), and the two are close enough to continue using the same prefixes.

Machine Code: The Language of Computers

Knowing a computer is limited to the operations described in its instruction set, and that these instructions are binary numbers, how are we able to use plain text and English to control what's happening?

The instructions typically cover behaviors such as addition, subtraction, multiplication, division, bit-shifting, logical operations and many different types of memory read and write operations.

Verbs such as “add” and “subtract” are much more meaningful than 10101010, 01010101 or whatever sequence of numbers happens to represent those instructions. The direction translation of machine code into human-readable representation is known as assembly language. Observe the example in Figure 8:



Assembly Language	Machine Code
add \$t1, t2, \$t3	04CB: 0000 0100 1100 1011
addi \$t2, \$t3, 60	16BC: 0001 0110 1011 1100
and \$t3, \$t1, \$t2	0299: 0000 0010 1001 1001
andi \$t3, \$t1, 5	22C5: 0010 0010 1100 0101
beq \$t1, \$t2, 4	3444: 0011 0100 0100 0100
bne \$t1, \$t2, 4	4444: 0100 0100 0100 0100
j 0x50	F032: 1111 0000 0011 0010
lw \$t1, 16(\$s1)	5A50: 0101 1010 0101 0000
nop	0005: 0000 0000 0000 0101
nor \$t3, \$t1, \$t2	029E: 0000 0010 1001 1110
or \$t3, \$t1, \$t2	029A: 0000 0010 1001 1010
ori \$t3, \$t1, 10	62CA: 0110 0010 1100 1010
ssl \$t2, \$t1, 2	0455: 0000 0100 0101 0101
srl \$t2, \$t1, 1	0457: 0000 0100 0101 0111
sw \$t1, 16(\$t0)	7050: 0111 0000 0101 0000
sub \$t2, \$t1, \$t0	0214: 0000 0010 0001 0100

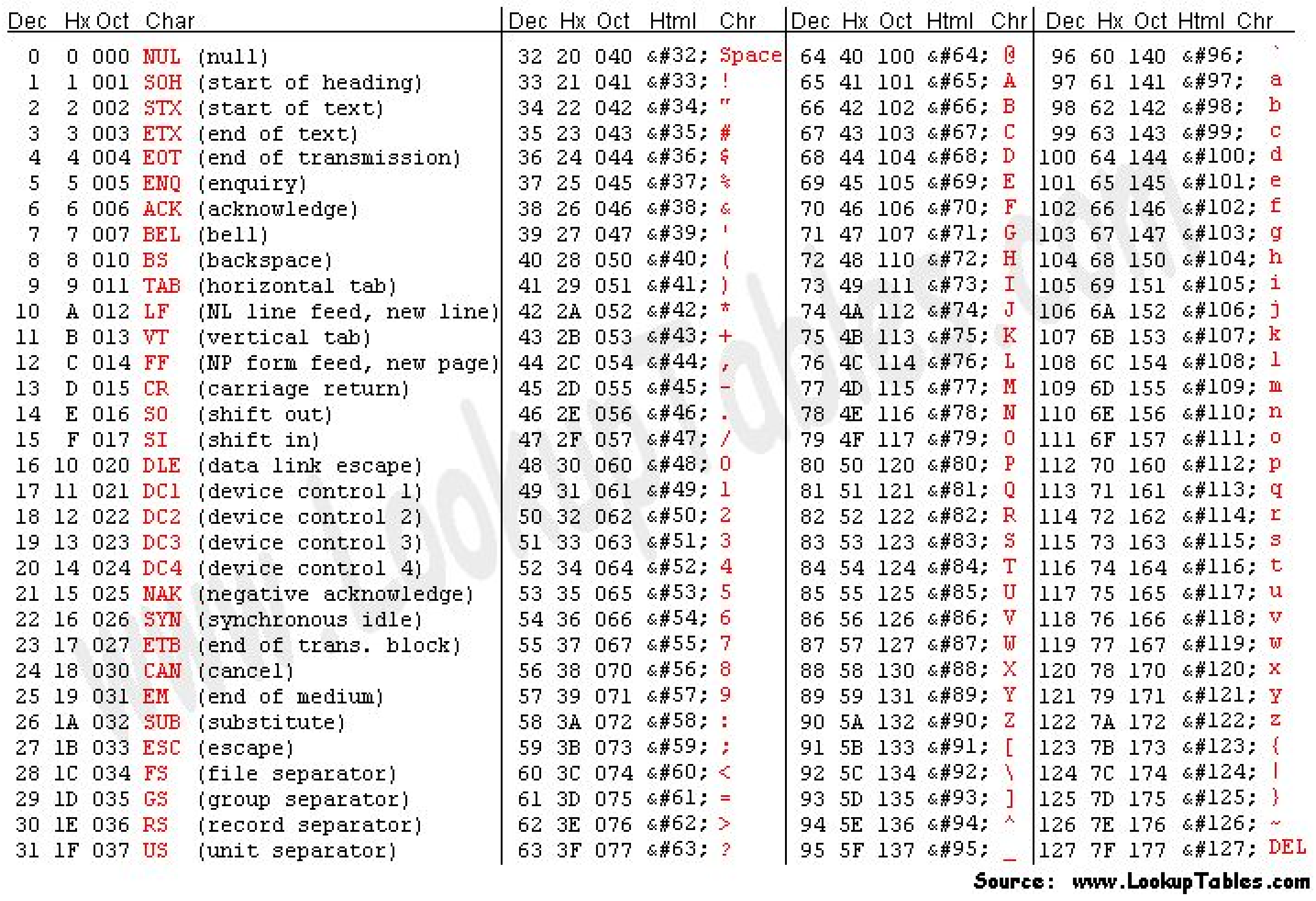
Figure 8: Assembly (left) with corresponding machine code (right). Specific assembly language and instruction set unknown

From left to right, an assembly instruction includes an abbreviation for the action being performed followed by the CPU registers (main, working memory) that denote the source and destination arguments.

The first line would cause register t1 to store the addition of t2 and t3: \$t1 = \$t2 + \$t3.

From Assembly to High Level Languages

In order to represent text with numbers, the American Standard Code for Information Interchange (ASCII) defines a mapping between decimal values 0-255 (0-[2^8 - 1] for 2^8 total symbols, one byte is required to represent an ASCII character or symbol). There are many variations of this standard, but Figure 9 shows the most common arrangement.



Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	SOH (start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	STX (start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B	98	62	142	&#98;	b
3	3	003	ETX (end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C	99	63	143	&#99;	c
4	4	004	EOT (end of transmission)	36	24	044	&#36;	$	68	44	104	&#68;	D	100	64	144	&#100;	d
5	5	005	ENQ (enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E	101	65	145	&#101;	e
6	6	006	ACK (acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F	102	66	146	&#102;	f
7	7	007	BEL (bell)	39	27	047	&#39;	'	71	47	107	&#71;	G	103	67	147	&#103;	g
8	8	010	BS (backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H	104	68	150	&#104;	h
9	9	011	TAB (horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I	105	69	151	&#105;	i
10	A	012	LF (NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J	106	6A	152	&#106;	j
11	B	013	VT (vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K	107	6B	153	&#107;	k
12	C	014	FF (NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L	108	6C	154	&#108;	l
13	D	015	CR (carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M	109	6D	155	&#109;	m
14	E	016	SO (shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N	110	6E	156	&#110;	n
15	F	017	SI (shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O	111	6F	157	&#111;	o
16	10	020	DLE (data link escape)	48	30	060	&#48;	0	80	50	120	&#80;	P	112	70	160	&#112;	p
17	11	021	DC1 (device control 1)	49	31	061	&#49;	1	81	51	121	&#81;	Q	113	71	161	&#113;	q
18	12	022	DC2 (device control 2)	50	32	062	&#50;	2	82	52	122	&#82;	R	114	72	162	&#114;	r
19	13	023	DC3 (device control 3)	51	33	063	&#51;	3	83	53	123	&#83;	S	115	73	163	&#115;	s
20	14	024	DC4 (device control 4)	52	34	064	&#52;	4	84	54	124	&#84;	T	116	74	164	&#116;	t
21	15	025	NAK (negative acknowledge)	53	35	065	&#53;	5	85	55	125	&#85;	U	117	75	165	&#117;	u
22	16	026	SYN (synchronous idle)	54	36	066	&#54;	6	86	56	126	&#86;	V	118	76	166	&#118;	v
23	17	027	ETB (end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W	119	77	167	&#119;	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	&#88;	X	120	78	170	&#120;	x
25	19	031	EM (end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y	121	79	171	&#121;	y
26	1A	032	SUB (substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z	122	7A	172	&#122;	z
27	1B	033	ESC (escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[	123	7B	173	&#123;	{
28	1C	034	FS (file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\	124	7C	174	&#124;	|
29	1D	035	GS (group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	]	125	7D	175	&#125;	}
30	1E	036	RS (record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	^	126	7E	176	&#126;	~
31	1F	037	US (unit separator)	63	3F	077	&#63;	?	95	5F	137	&#95;	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Figure 9: First 128 ASCII symbols, notice that not all symbols represent printable characters

This is how computers store text in memory, which is important because even with assembly language, there must be something to translate the assembly language text into machine code before it can execute.

This concept of translation can be applied again so that a new language can become assembly, which in turn becomes machine code. We can even do the same thing with our new languages.

This is where the term high level languages come from; the more translations required before code becomes machine readable, the higher level the language. For example, C is a language that compiles (term used to describe the translation from a programming language to machine language) into assembly language, where assembly language is then assembled into machine language by an assembler. If the code referenced code in other files or places in memory, a linker is responsible for piecing everything together for the final, executable machine code to be ready for use (Figure 10).

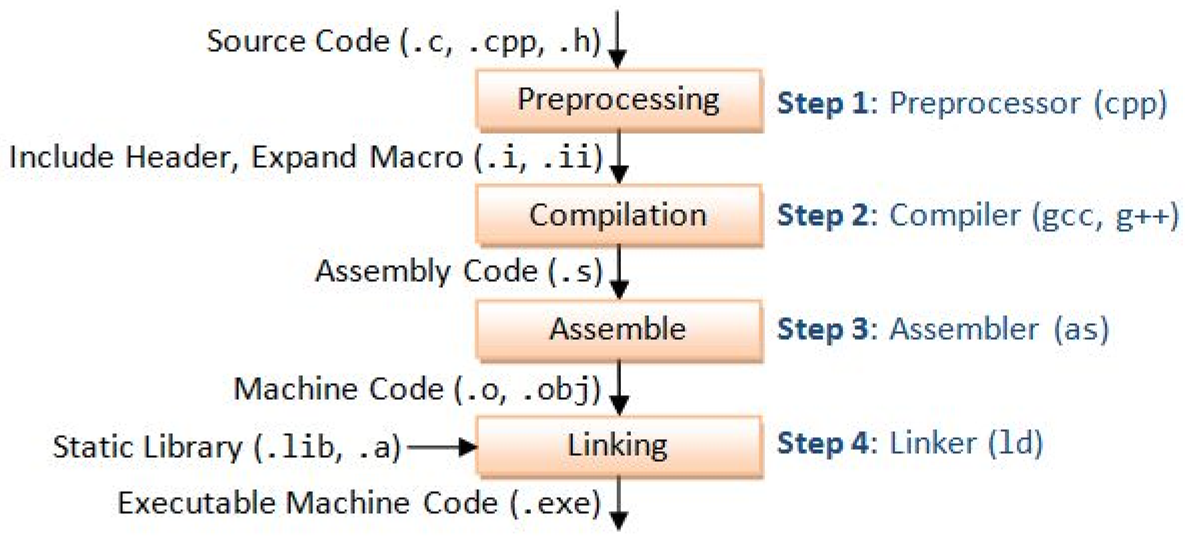
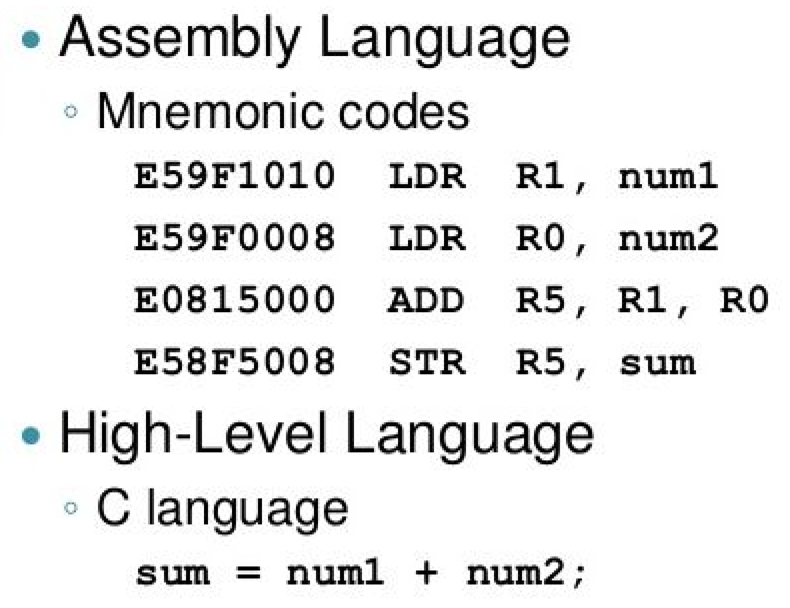

Source Code (.c, .cpp, .h)
Preprocessing
Step 1: Preprocessor (cpp)
Include Header, Expand Macro (.i, .ii)
Compilation
Step 2: Compiler (gcc, g++)
Assembly Code (.s)
Assemble
Step 3: Assembler (as)
Machine Code (.o, .obj)
Static Library (.lib, .a)
Linking
Step 4: Linker (ld)
Executable Machine Code (.exe)

Figure 10: The process for turning high-level code into executable instructions [31]

Let's look at how a single line of C translates into assembly in Figure 11 and then how a short C program represents a corresponding assembly program in Figure 12:



- Assembly Language
  - Mnemonic codes

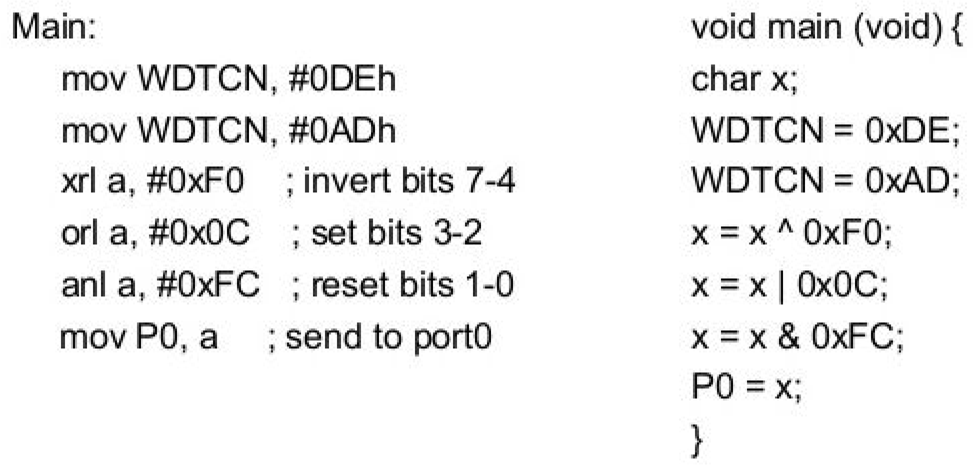
```
E59F1010   LDR   R1, num1
E59F0008   LDR   R0, num2
E0815000   ADD   R5, R1, R0
E58F5008   STR   R5, sum
```

- High-Level Language
  - C language

```
sum = num1 + num2;
```

Figure 11: C (below) and Assembly (above) Comparison I [32]

Notice that we can store the result of a sum of two numbers in one line, but in the assembly num1 and num2 are values being loaded into registers R1 and R2, and sum represents a specific position in memory that we write this result to. We are always aware of where things are from the perspective of the CPU, but in the C code we have abstracted num1, num2 and sum into variables, perhaps because the choice of register and memory location is arbitrary, and that we trust the preprocessor, compiler, assembler and linker to keep track of everything without us needing to know where it is.



```
Main:
    mov WDTCN, #0DEh
    mov WDTCN, #0ADh
    xrl a, #0xF0    ; invert bits 7-4
    orl a, #0x0C    ; set bits 3-2
    anl a, #0xFC    ; reset bits 1-0
    mov P0, a       ; send to port0
```

```
void main (void) {
char x;
WDTCN = 0xDE;
WDTCN = 0xAD;
x = x ^ 0xF0;
x = x | 0x0C;
x = x & 0xFC;
P0 = x;
}
```

Figure 12: C (right) and Assembly (left) Comparison II [33]

Seeing a more complete program provides better insight into what problems higher level languages solve and why we use them. x is a variable that we would like to represent a character (hence *char x*), typically meaning that we want to keep track of a meaningful ASCII value. With assembly, we would just have to remember that, for the register we choose to represent and keep track of x, we want it to always be 8 bits so we need to make sure to use instructions that perform operations with only 8 bits.

We also notice that C leverages the conventions of mathematical expressions: the left side of an equation represents the element that will hold the result of the operation taking place on the right side of the equation. The ampersand represents a logical *and* operation, the up-facing caret represents a logical *exclusive-or* and the pipe represents a logical *or*.

The main takeaway is that **a high level language can not ever replace assembly language completely**. It provides convenience for factors that a programmer does not need to be in control of for the task at hand, but in cases where full control is required, assembly language may need to be used. That, or rules may need to be given to the preprocessor, compiler, assembler or linker to ensure that those tasks are carried out according to the programmer's desired output.

A computing problem can be solved with programming in many different ways. The final noteworthy topic relevant to fully understanding this notion is that many common high-level languages are based not in assembly, but in C.