Programming Proficiency in One Semester: Lessons Learned

Don Colton
doncolton2@gmail.com

Aaron Curtis
aaron.mosiah.curtis@gmail.com

Computer and Information Sciences
Brigham Young University Hawaii
Laie, Hawaii 96762 USA

Abstract

Programming is a fundamental skill for Information Systems and Information Technology students. It is also a subject that some students fear, avoid, fail, retake, and fail again. An effective, inexpensive, one-semester approach is presented. Early indications suggest dramatically improved student interest and performance compared to our previous two-semester approach. Key features include heavy use of web-based online programming, use of a scripting language (Perl), development of general-purpose programming skills, and a free textbook (PDF).

Keywords: programming, curriculum, Perl, online programming, free textbook

1. INTRODUCTION

Programming is a fundamental skill for information systems and information technology students and professionals. Although some professionals seldom write a program, the skills can come into play in understanding what subordinates do, in writing spreadsheets, and in automating processes.

Accreditation standards (ABET, 2008) and model computing curriculum recommendations emphasize the importance of programming proficiency, (Shackelford, 2005; Gorgone, 2002). However, many of our students in IS and IT seem to consider programming to be a CS activity, and one they would rather avoid. Programming is not something these students visualize themselves as doing in their future careers. Students within the IS and IT programs, therefore, have difficulty maintaining engagement in computer programming courses.

Introductory students in these programs often find programming to be boring and difficult (Jenkins 2002) and experience high rates of failure (Bennedson and Casperson, 2007). Many students respond to these challenges by concluding that they are simply incapable as programmers (Jenkins 2001). These perceptions of incompetence result in significant dropout / failure rates in introductory programming courses and poor performance in subsequent programming courses (Guzdial and Soloway, 2002).

Educator responses to these failings include believing some students really cannot program, thereby lowering expectations of student performance (Evans and Simkin 1989), attempting to innovate in their teaching techniques to promote greater engagement (e.g., Leutenneger and Eddington 2007), and blaming poor performance on "unmotivated" students (Gill and Holton, 2006).

In this paper, we outline our experience in developing a one-semester approach that meets the standards and curriculum recommendations of teaching programming fundamentals while creating a learning environment in which students can develop competence and confidence as emerging programmers. The preliminary results of this approach suggest that student performance and perceptions of computer programming have improved significantly. We anticipate that the lessons learned in our experience will be helpful to educators attempting to address these issues in their local institutions.

2. THE WAY WE WERE

Many of our students in IS and IT seem to consider programming to be a CS activity, and one they would rather avoid (much like Calculus). Despite accreditation standards and model curriculum recommendations, it is not something they visualize themselves doing as part of their job. We have to sell it well for students to decide to really engage in learning.

Some years ago we set the bar fairly low. Students read and discussed simple programs but did not actually write them. Some teachers were afraid that students would fail, become discouraged, and change majors. Such an overview did not give students adequate programming skills to actually do even small-scale projects.

We tried setting the bar higher, with a long, shallow learning curve, using a two-semester approach and numerous micro-projects (Colton et al, 2005; Colton et al 2006). It worked much better. Most students developed skills but were not eager to use them. For them programming was tedious instead of fun.

We looked at other approaches but were put off by the high proportion of “magic” that seemed to be involved. By magic we mean that students developed skills that work marvelously well in a small number of settings but did not transfer to more general settings. This felt like “training” instead of “education.”

Finally we abandoned a key element of our two-semester sequence, creating in its place a new two-semester sequence. As the change worked its way through the system, we discovered some useful economies that led eventually to the development of a single, one-semester programming course.

2.1 The Two-Semester Sequence

Our department hosts three majors: Computer Science, Information Systems, and Information Technology. Where possible, courses are made to serve more than one major. The introductory programming class is taken by students in all three majors.

(One goal of the introductory programming class is to help computing students select the major that will best suit them. As freshmen, students often do not understand the differences between CS, IS, and IT. Seeing all three types of students in the same class helps students self-identify more accurately.)

We used C in CIS 101 as our foundational language. This choice was motivated by several factors. First, C is well known and highly respected. Second, C is small enough to be well understood. Third, programming skills seem to transfer well from C to other languages students may need to learn later. Fourth, the class was to be taught by CS faculty and C or C++ or Java was their language of choice. C (C++) seemed easiest.

The learning objective from the CIS 101 course was that students be proficient with variables and data types (int, float, char) and be introduced to arrays, and that they be proficient with if/else and loops (while, do while, for) and be introduced to subroutines.

We used Perl in CIS 201 as our follow-on language. (For those unfamiliar with Perl, Appendix B gives a few short example programs written in this language.) This choice was also motivated by several factors. Scripting languages (like Perl, Python, and Ruby) are much faster for completing small- to medium-sized programming projects. Scripting skills are important for CS, IS, and IT students. Among scripting languages, Perl is well known and highly respected. It is not small like C, but it has a large body of open source shared archives (since 1995, the Comprehensive Perl Archive Network at http://CPAN.org/).

The learning objective from the CIS 201 course was that students transfer all their CIS 101 skills to Perl, thus seeing how easy it is to learn a second language, and in addition become skilled at database access and online programming (CGI) on a Linux platform. This also created the opportunity to introduce and develop skill with regular expressions. The capstone project was to build from scratch a small online store complete with shopping cart and inventory system.

2.2 Mistaken Assumptions

We assumed that after learning C for a semester, students would find it easy to learn Perl. They would conclude that it would be easy to learn additional languages later, as needed. This confidence was a major goal of using two different languages.

Unfortunately, for IS and IT students the single semester of C did not result in adequate programming skills on which to springboard into another language. Students had to relearn everything and the learning speed was only slightly faster than the first time they learned it.

In CIS 201, it was too easy to become frustrated by the slowness of students in demonstrating skills they should have already mastered. It was too easy to blame the CIS 101 teacher for failing to teach. It was too easy to blame the students for being stupid. In retrospect, it appears the problem was the course design in CIS 101. There was a mismatch between the expectations in CIS 101 and the abilities and interests of the students in CIS 101.

It was not immediately obvious, but instead of having two semesters of programming, students were having one semester of programming, twice. The synergy was missing.

3. TIPPING POINT

There were other frustrations with the existing programming sequence. The CS faculty did not see enough value in the CIS 201 class and wanted to remove their students from it, substituting an additional semester of Java. CS was one of the longest majors on campus in terms of credit hours, and felt the need to add new courses but also wanted to abandon old courses of limited value.

This gave rise to the question of whether CS students should learn a scripting language at all. It was decided that there was still a real need for scripting in CS. A suggestion was then made by CS to convert the CIS 101 class over to scripting. It was a totally unexpected suggestion, but it quickly developed broad support. The new approach would be to teach scripting in CIS 101 to all students, and more advanced scripting in CIS 201 to just the IS students. IS students would forego the learning of C.

The Resulting Curriculum

Starting August 2008 we converted the CIS 101 course into Perl and merged in the major features of the old CIS 201 class.

Under the old plan, the 201 class spent 1/3 of its time reviewing basic concepts from 101 but recasting them in the light of Perl. The next 1/3 of the course was online programming. The final 1/3 of the course was database using mySQL and simple queries (select, insert, update, no joins). Under the new plan it was hoped that the first 1/3 of the course would no longer be needed and new material could be added.

Because there were already students in the pipeline, in Fall 2008 both 101 and 201 were taught in Perl. In Winter 2009 the 201 students included some who had learned Perl before as well as some that had only learned C before. But over the course of Fall and Winter we made some interesting discoveries.

The immediate results were very interesting. During the first semester after the change, CIS 101 and CIS 201 were taught by the same instructor. CIS 101 students learned Perl as their first language, and CIS 201 students learned Perl for the first time, but as their second language. Remarkably, the 101 students did nearly as well as the 201 students. 201 students continued to perform largely as before. Also, CIS 101 students gave the teacher high ratings and CIS 201 students gave the teacher lower ratings.

Table 1 (in the Results section below) presents results in terms of learning objectives mastered by 101 and 201 students respectively. It shows that for many objectives, the top 90% of 101 students performed at the same level as 201 students.

Several theories emerged to explain this phenomenon. (1) Perhaps 201 students were frustrated that they were being asked to learn a second language when their first language had been boring. This frustration was realized as push-back against the course and decreased learning. (2) Perhaps 201 students had not learned C well enough that the programming skills were transferrable to a new language yet.

Whatever the reason (both seem to be plausible), it called into question our old theories about how best to teach programming. It was concluded by the faculty that if 101 students could perform at roughly a 201 level, then the 201 course was not needed.

It was decided to do away with CIS 201 and create instead a new capstone CIS 401 web programming course involving additional prerequisites and featuring the PHP language. The new course would have as prerequisites courses in webpage development (xhtml, css) and in database (SQL including join). This would allow greater development of marketable skills for our students.

CIS 201 was taught for the last time in May 2009. CIS 401 is being taught for the first time in September 2009.

4. PROGRAMMING IN ONE SEMESTER

The new model is for all things programming to be taught in CIS 101 such that students emerge with programming proficiency to the degree expected of IS graduates. That is a tall order, but we try to deliver on it by using the following approach.

First, students learn a single language. We teach them Perl but keep it simple, at least at first, in hopes that the skills will be transferrable to any other language they may need to learn. There is a great emphasis on portability of approach, skills, and knowledge. (For those unfamiliar with Perl, appendix B gives a few small sample programs.)

Second, we wrote our own textbook. This was a major step, not undertaken lightly, and not something we recommend to everyone. But the book, version 1.0, is in use and freely available to other schools for adoption as a primary text or as a supplement.

Third, we strongly emphasized online programming. We found that students respond enthusiastically to having their programs run on the web and being able to share them with friends near and far. At the same time we were very cautious to not delve too much into magic, where students do not really understand what they are doing but look up recipes in some index. This was greatly facilitated by having our own textbook.

4.1 Single Language

Ideally we might teach the language that all future employers will demand. Unfortunately, employers have not converged on a standard. Fortunately there are some favorite languages among employers, and most of these are similar to one another. We hope to teach a language that will be an easy basis for students to go on to other languages as their circumstances may demand.

We generally agree that any of several scripting languages could be used effectively. Our current choice is Perl. Factors in language selection include being typical, powerful, well known, portable, and well supported.

Typical: By this we mean that skills transfer well to other programming languages.

Powerful: By this we mean useful programs can be written fairly easily with the level of skill our students would achieve.

Well Known: By this we mean employers have heard of it, so it could be meaningfully listed on a resume.

Portable: By this we mean programs written for the Linux platform will also work on Microsoft or Macintosh and vice versa.

Well Supported: By this we mean there is a large user community that is actively helping each other and there are large collections and archives of program libraries available to everyone.

4.2 Textbook

Our textbook, Introduction to Programming Using Perl, is available free online as a PDF file at http://ipup.doncolton.com/.

The book is designed to support an introductory (100-level) college course that meets for about forty hours during a semester or quarter.

The book contains about 60 chapters divided into eight units for a total of about 340 pages. Most of the early chapters are designed to be read at the rate of one to two chapters per hour of time in the classroom.

We assume that students are not yet convinced of the importance of programming and may be taking the class simply because it is required. We motivate their study by emphasizing questions of "why" as well as showing "how." We focus on introductory issues.

Advanced topics are usually mentioned briefly but with enough detail that students can follow up by searching the Internet. We assume the students of today are skillful at using search engines and other tools to get in-depth answers on topics of interest to themselves, once they know the topics exist and what they are called.

4.3 Online Programming

Our students turned out to be VERY enthusiastic about online programming. We believe this is for two reasons. (a) It allows them to share their work with friends and family anywhere in the world. (b) It allows them to integrate graphical elements easily.

During the first semester of our new course, we deferred online programming until almost the end of the semester because of the difficulty of parsing online input. Enthusiastic student response to online programming made us to look for ways to teach online skills earlier in the course.

The downsides of online programming are (a) the difficulty of maintaining state (conversation), and (b) the difficulty of working with "open set" input. (Restricted, "canned," or "closed set" input turns out to be easily handled.)

We reorganized the course to give students very early success at creating online programs, starting from a simple graphics program that rolls dice or tosses a coin, and continuing closed set inputs, and finally reaching open set inputs through regular expressions.

Rolling dice or tossing coins requires very little beyond the "Hello, World" level of programming. We need only include a lesson on random numbers. The results are immediately impressive. Students have something to show off within the first two weeks of the semester. If/else is not required. Loops are not required. Subroutines are not required.

Closed-set input allows programs to respond to button presses. Because there are only a few possible inputs to consider, they can be handled through an explicit series of if/else statements. One classic game we program is Rock Paper Scissors, where the human pushes a button for one of the three, the computer program randomly selects one of the three, and a series of if/else statements resolve the winner. Graphics are included to improve the appeal of the program. We find this can be done by around the fourth week of a 14-week class.

Open-set input requires the use of mysterious library functions or an understanding of regular expressions. We take advantage of the opportunity to introduce regular expressions. However, because of the complexity involved, we feel this cannot be well done until about week 10 of a 14-week class.

5. LEARNING OBJECTIVES

Our learning objectives have been designed to be realistic for 80 percent of our students.

We do not assume any prior programming knowledge or experience. We do assume students have math skills to do simple algebra (solve x - 7 = 3) and that they have access to a computer with Perl installed.

5.1 Basic Expectations

Our standard is that basic material must be mastered so later courses can build on it. This is not a survey course or a high-level overview like, say, Art Appreciation. It is a "do it" class like, say, Drawing.

Because programming must become a basis on which other courses can build, we measure student performance on closed-book programming tests.

Mastery is divided into five topics: basics, decisions, iteration, arrays, and subroutines. By the end of the semester, students must master the first two topics to pass the class (with a D). They must master the first three to get a C. They must master all five to get a B. They must also demonstrate ability with some advanced material to earn an A.

Basics: Students write correct programs that use standard input and output to get information into and out of the computer. Programs run from a Graphical User Interface (GUI) or from a Command Line Interface (CLI). Students demonstrate the ability to use normal (scalar) variables to do calculations such as inches to centimeters. Students use fundamental mathematical operators including add, subtract, multiply, divide, and parentheses. Students understand that statements are executed in order, one after another, and that later statements can change the values of variables from what the earlier statements established. We introduce style rules including naming of variables and spacing of written programs.

Decisions (if/elsif/else): Students write correct programs that deal differently with alternate cases, such as whether to put AM or PM after the time, or whether a check will be honored or will bounce. This includes skill with Boolean operators (those yielding a True or False answer) such as comparatives (less than, greater than, equal to, not equal to) and conjunctives (and, or). This also includes following style rules of indentation and spacing to make complex programs more readable.

Loops (Repeated Actions): Students write correct programs that deal with repetition of actions, such as filling out a table. Style is also emphasized. Operators like **++** and **+=** are mastered. **next**, **last**, and **redo** are introduced.

Arrays/Lists (Repeated Data): Students write correct programs that deal with lists of information. The **foreach** loop is mastered. **push**, **pop**, **shift**, **unshift**, and indexing (**[1]** and **[-1]**) are mastered.

Organizing (Subroutines, Functions, Methods, Objects): Students correctly write subroutines to better organize and structure their code. Local and global scope of variables is understood.

5.2 Proving Mastery

To prove mastery students are given a Final Exam that has five sections (one per topic area) each with several programs of varying difficulty. Students must correctly write those programs in a topic area before we consider it to be mastered.

(We chose to do this instead of having everyone do a major project for the unfortunate reason that too many students were turning in project work they did not understand. They were apparently turning in someone else’s work as their own. However, once a student demonstrates adequate mastery on exams, we DO utilize a term project as a way to motivate the A students and separate them from the B students.)

A large number of sample problems from the Final Exams are given in the free PDF textbook.

Because students do not all learn at the same rate, we do not care when students demonstrate mastery as long as it is by the last day of class. The jury is still out on how much this simply invites students to procrastinate.

The course grade is based almost totally on the final exam, so little else really matters. This takes the teeth out of midterms and homework assignments. What do we do about that? An all-or-nothing final can be pretty scary. So we compromise by giving the Early Final.

5.3 The Early Final

About once a week we offer an actual final exam. The questions are different each time, but are basically the same or of the same difficulty. The rule is that if a student passes any section of the exam, they don't have to take that section again. This gives them a reason to take the tests and to make progress. The entire final is too much to take on the last day of class. Knowing that part of the final is completed seems to be a good motivator.

Toward the start of the semester, the weekly exam covers only material already studied in class, so it is much shorter than it will eventually become. We allow about ten minutes for the test for the first few weeks. As more material is covered in class, new sections are added, making the test longer. Student performance also spreads out, with some students having completed the early sections while other students continue to struggle. We increase the amount of time allowed to 20 or 30 minutes. The last few weeks of the semester we allow the full class period once a week for the exam.

Because each Early Final is actually the real final, all the normal rules apply. The exams are closely proctored and performance must be at a final-exam level. Toward the start of the semester very few students pass anything. Toward the end many students are passing things.

The unit tests throughout the textbook give actual test questions that have been used in the Finals to assess mastery of each topic. Students are also allowed to keep a copy of the exam and the work they did. One day later they can share their efforts with each other.

During the exams we allow the students to test their work by running it at their local machine. However, they are not allowed to use any notes or outside resources, including web pages. They are only allowed to test their programs by running them locally. If there are reference materials we wish to make available, we put them in the test itself.

Scoring: We grade programs "by hand," visually examining the student code. In addition to working, we expect student programs to demonstrate the requested programming style (indentation, spacing, comments, naming) to make the programs easy to read and understand.

6. RESULTS

It is probably not possible to give definitive results given the small sample size, but preliminary results are encouraging. The following table gives the percentage of students achieving **100%** in each level of skill during our Winter 2009 term. Winter represents the main transitional term, where students in 201 had a previous term probably in a different language. We do not have directly comparable performance metrics for 101 students under the old plan.

Nearly identical final exams were used across all sections of both courses. The same textbook was used in all courses. The same instructor taught all sections. The 101/201 column tells how well the 101 students did as a group compared to the 201 students.

Table 1: Mastery of Learning Objectives

Skill	101	201	101/201
Basics	93%	96%	97%
if / else	83%	92%	90%
Iteration	72%	80%	90%
Arrays	29%	32%	91%
Subroutines	9%	16%	56%
Online Skills	28%	24%	117%
Term Project	3%	20%	15%
N (students)	58	25	

Performance on basics, if/else, iteration, arrays, and online skills was all at the 90% or better level for the group of 101 students compared to the 201 students.

With subroutines, only half as many 101 students performed at the 201 level. With term projects, only a small fraction performed at the 201 level.

The online skills section is noticeably better for the 101 students, but this may not be statistically significant given the small sample size. It seems however to suggest that 101 students may have been riding a wave of excitement while 201 students were fighting pre-conceived notions of whether it would be interesting.

7. INSTRUCTOR RESPONSE

The primary author of this paper also wrote the textbook. His experience has been that it is wonderful to control the textbook in such an intimate way because it allows the book to be adjusted from time to time to match the performance of the students and to respond to the difficulties they face. However, writing a textbook is a major commitment and takes a lot of time.

Another instructor to teach a full course using the book responded: "I really liked the format and pace of the class. I would like to extend [the book's coverage] into [the next CS class] as well."

8. CONCLUSIONS

Online programming is HIGHLY motivational to students because it facilitates sharing their achievements and allows graphical and other creative elements to be involved.

Programming proficiency seems to be achievable in a one-semester course that is well structured and adequately supported.

A free textbook (PDF) is available for use as a primary text or as a supplementary text in classes such as this.

9. REFERENCES

ABET Computing Accreditation Commission, (2008). Criteria for Accrediting Computing Programs, Effective for Evaluations During the 2009-2010 Accreditation

Cycle. PDF accessed 2009-07-30 from http://abet.org/forms.shtml

Colton, Don, 2009. Introduction to Programming Using Perl. Available at http://ipup.doncolton.com/

Colton, Don, Leslie Fife, and Andrew Thompson. 2006. A Web-based Automatic Program Grader. ISEDJ 4(114). http://isedj.org/4/114/

Colton, Don, Leslie Fife, and Randy Winters, 2005, Building a Computer Program Grader. ISEDJ 3(6). http://isedj.org/3/6/

Evans, Gerald E., and Mark G. Simkin. 1989. What best predicts computer proficiency? *Commun. ACM* 32, no. 11: 1322-1327. doi:10.1145/68814.68817.

Gill, T. G, and C. F Holton. 2006. A self-paced introductory programming course. *Journal of Information Technology Education* 5: 95–105.

Gorgone, John T, Gordon B. Davis, Joseph S. Valacich, Heikki Topi, David L. Feinstein, Herbert E. Longenecker, Jr, (2002). Model Curriculum and Guidelines for Undergraduate Degree Programs (IS2002), ACM, AIS, AITP.

Guzdial, Mark, and Elliot Soloway. 2002. Teaching the Nintendo generation to program. *Commun. ACM* 45, no. 4: 17-21. doi:10.1145/505248.505261.

Jenkins, AMJ, and Davy, JR. 2001. Diversity and motivation in introductory programming. *Innovations in Teaching And Learning in Information and Computer Sciences* 1.

Jenkins, T. 2002. On the difficulty of learning to program. In *Proceedings of the 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences*, 53–58.

Lahtinen, Essi, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. 2005. A study of the difficulties of novice programmers. In *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, 14-18. Caparica, Portugal: ACM. doi:10.1145/1067445.1067453. http://portal.acm.org/citation.cfm?id=1067445.1067453.

Leutenegger, Scott, and Jeffrey Edgington. 2007. A games first approach to teaching introductory programming. *SIGCSE Bull.* 39, no. 1: 115-118. doi:10.1145/1227504.1227352.

Shackelford, Russell, James H. Cross II, Gordon Davies, John Impagliazzo, Reza Kamali, Richard LeBlanc, Barry Lunt, Andrew McGettrick, Robert Sloan, Heikki Topi, comprising the Joint Task Force for Computing Curricula 2005 (ACM, AIS, IEEE-CS), (2005). Computing Curricula 2005, The Overview Report covering undergraduate degree programs in Computer Engineering, Computer Science, Information Systems, Information Technology, Software Engineering. (30 Sep 2005) http://www.acm.org/education/curricula-recommendations

APPENDIX

Free IPUP Textbook

http://ipup.doncolton.com/

Detailed Table of Contents

APPENDIX B

Brief examples of Perl.

Following are some short programs in Perl to give a flavor for the language for those that may not be familiar with it.

Basic I/O

Print “Hello, World!”

```
print "Hello, World!\n"
```

Read in a name. Print on one line “I think (name) is a wonderful name.”

```
$name = <STDIN>;
chomp ( $name)
print "I think $name is a wonderful
name.\n"
```

Simple Calculation

Read in two numbers. Print their total.

```
$x = <STDIN>;
$y = <STDIN>;
$z = $x + $y;
print "$x + $y = $z\n";
```

Simple Decision

Read in a number. If greater than 10 print “Big”. If less than 5 print “Small”. Otherwise print “Medium”.

```
$x = <STDIN>;
if ( $x > 10 ) { print "Big" }
elsif ( $x < 5 ) { print "Small" }
else { print "Medium" }
```

Iteration

Read in a number. Print the numbers from 1 up to the number read in.

```
$max = <STDIN>;
for ( $i = 1; $i <= $max; $i++ ) {
  print "$i\n" }
```

Array by Index

Given an array abc, print the fifth element.

```
print "$abc[4]\n";
```

Given an array abc, print the next to last element.

```
print "$abc[-2]\n";
```

Subroutine

Write a subroutine abc that accepts two parameters and returns the sum of them.

```
sub abc {
  my ( $x, $y ) = @_;
  return $x + $y }
```

Online (CGI)

Write a program that rolls two dice and displays the results in a web browser. (Notice that the html is barely adequate. We teach html in a different course.) This presumes that 1.jpg through 6.jpg are available to display. rand(6) returns a uniformly distributed random number between 0.00 and 5.99.

```
print "content-type: text/html\n\n";
$d1 = int ( rand(6) ) + 1;
$d2 = int ( rand(6) ) + 1;
print "<img src='$d1.jpg'>";
print "<img src='$d2.jpg'>";
```