Thoughts on First-Year Computer Science Instruction

Jonathan W. Berry
Elon College
berryj@elon.edu

July 10, 2000

1 Programming: Should We be Apologetic?

A common thread in the computer science education community is that we should stress mathematical and logical fundamentals rather than programming in the first course. Institutions such as SUNY Stony Brook do this, and their demographics seem to tolerate the relatively early mathematical rigor. However, the vast majority of CS1 courses seem to be very programming-intensive, considering the nature of the most popular books and the discussions on the SIGCSE mailing list.

Ideas such as SUNY Stony Brook's are intellectually appealing. Indeed, it would be nice to ensure that students have early exposure to mathematical reasoning, as this exposure might help them to grasp programming with high-level languages and other traditional first-year concepts more quickly and firmly. However, the opportunity cost is high. For example, programs in non-research institutions with small-enrollment departments might suffer. Also, early treatment of mathematical reasoning topics must necessarily be less rigorous that the more traditional second or third-year treatment.

Aside from pedagogical issues such as when to teach what, however, I think a much better argument for retaining a central focus on programming is that we should put our best foot forward in the first CS course. Programming, much maligned programming, gives a truer flavor of what CS is all about to most people in the field than, say, equivalence relations. The latter can be taught after the first year. I majored in computer science instead of mathematics because I really enjoyed writing programs that worked. Thus, for motivational reasons, I suggest that we keep programming central to first-year instruction.

I love Knuth's quote: "The chief goal of my work as educator and author is to help people learn how to write beautiful programs."

2 Language: I'd Love Scheme, but...

I have had a lot of experience developing in object-oriented Scheme (STk) as the leader of the LINK project. At first, I had great hopes for the educational applications of this workbench for graph computation, seeing Scheme as a vehicle to take discrete mathematics and programming concepts down to the junior high school level. Sometimes I can revive that optimism, but it is hard. Unfortunately, the Scheme interpreters I've used produce stack dumps and attempted error descriptions that are not adequate for educational purposes. Some versions of the STk interpreter can even get themselves into infinite error loops. I would not wish this on any student, let alone a struggling CS1 student. C++ syntax is involved, but at least the compiler will respond to errors accurately and in detail (unless you use templates).

3 Java

Unsure about switching our CS1 to Java, I have instead introduced Java in my upper-level courses. This past semester, I used Appel's *Modern Compiler Implementation Using Java* in our senior-level compilers course, and it was a remarkable experience. I was teaching CS2 concurrently using a popular C++ CS2 book, and the contrast was stark. Appel's juxtaposition of object-oriented and functional techniques led to tight, beautiful implementations of all of the components needed by a basic compiler. The linked lists were clean, small, and easy. Contrast this with the templated C++ linked lists presented in the other book to my CS2 students. The C++ structures are so large and unwieldy in comparison! I do sympathize

with the CS2 author, since I have a lot of templated C++ experience.

Seeing my CS2 students struggle with templated C++ list classes while my senior compiler students cruised to a working compiler via simple, clean Java structures [1] was a memorable experience.

In essence, there are not many true fundamentals in a traditional CS1 course: looping, branching, scoping, subroutines, objects, and manipulating vectors come to mind immediately. Of course, these translate into "looping, branching, scoping, subroutines, objects, manipulating vectors, and *learning tons of implementation details.*" As languages improve, so will CS1.

4 Evaluation: Up to Honesty, Projects Mean More to Me than Exams

I would be curious to learn how other educators make and grade their exams. I like to give problem solving examinations, even in CS1, and students cannot solve many problems in an hour. How do you grade an examination of 100 total points that includes 2 or more problems worth 20 points each? What does $\frac{14}{20}$ mean? What grade is assigned to hard work that is nevertheless "F" level work? $\frac{0}{20}$? $\frac{10}{20}$? These are important questions. I end up constructing partial orders of examination papers for each problem. My goal is give higher grades for better performances while retaining a reasonable distribution. This seems to lead to the assignment of many C-level grades to responses that are wrong in many ways.

Whenever I read a post on SIGCSE assuring us that examinations will catch those who cheat on the projects, I wonder how those examinations are constructed and graded. Pure multiple-choice examinations cannot measure creativity or perseverance, while problem-solving examinations probably pass many people who couldn't construct a program on their own.

An individually completed, correct project that involves some set of concepts is a much better indication of comprehension than an examination grade. After all, setting an hour time limit after which we must "give up" is the antithesis of most computer science work.

[1] This code can be written very cleanly in C++ as well (by ignoring certain features).

5 Another World: the FAA, the Aviation Industry, and Training

I am a long-time pilot with a commercial certificate and instrument and multi-engine ratings. My studies to become a Certified Flight Instructor have introduced me to the FAA's world of education, and it is interesting to compare this training-intensive world with our own SIGCSE world. One commonality between these two worlds is that they both lead students into challenging technical careers in which attention to detail is crucial. Aside from the clear differences in content and depth between these two worlds (airline pilots need a college degree, but need not major in a technical field), some stark contrasts are immediately apparent:

- In order to work a "real job" in the aviation industry, most professionals *must teach first.*
- In order to teach, professionals *must study the fundamentals of learning and obtain an instructor's certificate.*

There is a dogmatism about the FAA's approach to training and education that can be trying, but also somewhat refreshing. After all, many of the Federal Aviation Regulations are "written in blood." Doing things correctly is a matter of life and death, and students *must* meet the appropriate standards, or they will not be certified. It would be interesting to see the computer science community develop some set of baseline competency standards for its graduates. Simply passing a course does not indicate competency, nor does taking a major field achievement test. The standards should be performance-based (e.g., after fyi, could all students write a working List ADT (with methods *insert_front()* and *print()*) from scratch with no help, given 3 hours in which to work?).

Computer science educators must provide a deeper understanding of concepts than aviation instructors generally can. However, computer science educators, acting as judge and jury, have the ability to certify, through their grades and degrees, that their students are competent to enter the work force. Aviation instructors only act as coaches, passing their students off to FAA designated examiners for the final judgments.

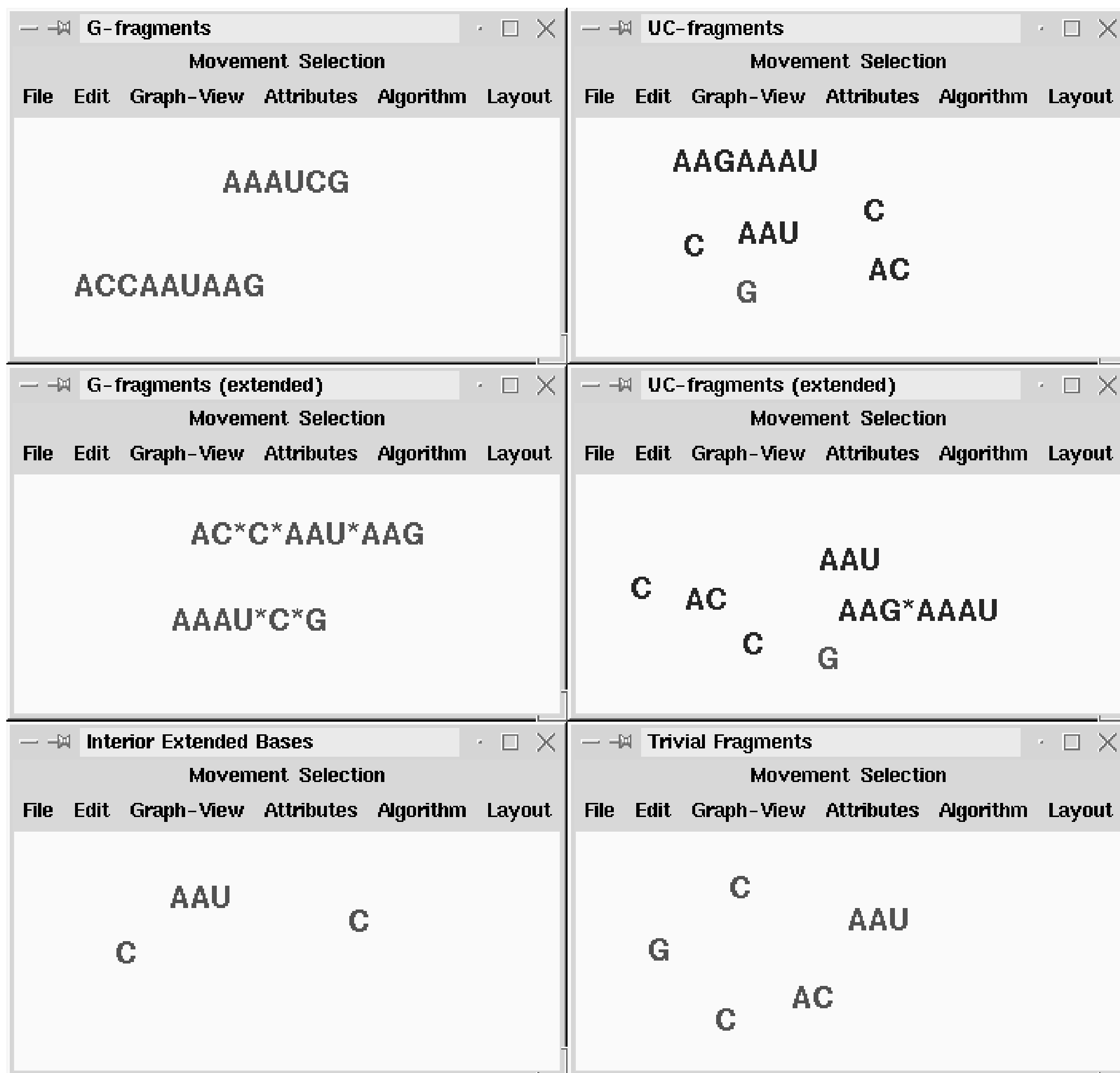

Figure 1: The RNA string application: fragments and extended bases

6 Conclusion

Good mathematics presentation is elegant and concise. Unfortunately, CS presentation is hampered by the state of the art in programming language design. My good experience with a textbook written by a programming paradigms expert leads me to suggest that we try to involve more of these professionals in first-year curriculum development.

My experience with the FAA's pilot training programs have given me a window into another, very different world of training. In some ways, it is not analogous to our SIGCSE world, but in others, it is. I have confidence that any FAA-certified pilot can fly an airplane safely in an appropriate set of conditions. However, I don't have the same level of confidence that any computer science graduate can write an involved program. The comparison is not quite fair, I realize, yet I think it is still an interesting comparison.

7 Preview: Scheme-Based Software Support for a Module on RNA Reconstruction

Lidia Luquet and Kenneth Price are writing a DIMACS module intended to present an RNA reconstruction technique. My students and I have developed software extensions to LINK to act as a compan-

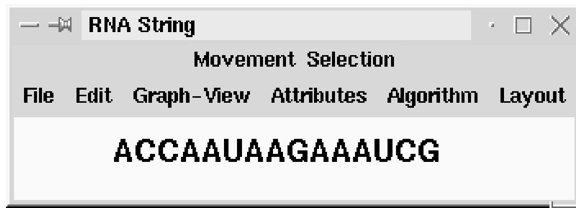

Figure 2: The RNA string application: initial string

ion to the module. None of this work is ready to be distributed as assignments, but I include some screenshots below to give a flavor of what will be possible. These graphics were developed using object-oriented Scheme, a language that could be accessible to first-year students (if only the interpreter gave reliable and usable error output).

The application breaks the strings into fragments, breaks those fragments into *extended bases*, then constructs a digraph. Eulerian circuits through this digraph can reconstruct the original RNA string.

I have developed a 50 page LINK tutorial that could be used as a supplement to a C++ or Java-based first-year CS course. LINK is free and provides GUI and OO-Scheme-based access to graph primitives. See:

`http://dimacs.rutgers.edu/Projects/LINK.html`

New versions of LINK have been under development for over two years, but problems with STk are delaying their release.

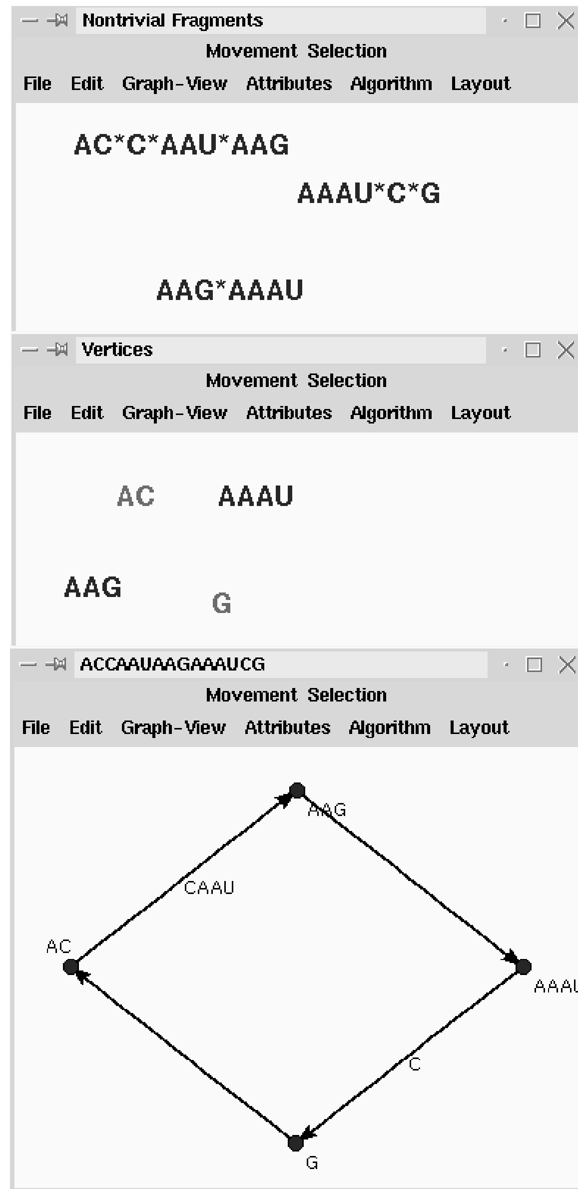

Figure 3: The RNA string application: building the graph