Updating 11 and 15

Ben Hescott
Computer Science
hescott@cs.tufts.edu

Norman Ramsey
Computer Science
nr@cs.tufts.edu

Abstract

We propose to revise COMP 11 and COMP 15 to help students become better programmers and problem solvers and to provide a stronger foundation for upper-level courses. We will introduce fundamental techniques early, and we will plan to revisit such techniques at greater depth in subsequent courses. Fundamental techniques for 11 include abstraction, recursion, systematic methods for problem-solving, and systematic methods for software development.COMP 15 will refine and expand this material and will add data structures, data encapsulation, modular reasoning, and reasoning about costs. We have identified a promising body of work to support a revised 11; it is the *Program by Design* method developed at Rice, which has been successful at both North American and European universities. To revise 15, we plan to do much of the work ourselves; the theme of the course will be *Abstraction + Cost Model = Data Structure*. We also plan for a part of 15 to teach students how to pick up a new programming language.

1. Introduction

This document proposes a plan for revising COMP 11 and 15. We have developed this plan through conversations with our colleagues extending back three years. The plan is also informed by interviews that Kathleen Fisher conducted in 2011, which involved all our faculty. Kathleen identified far more concerns than could be addressed in any imaginable 11 and 15. Our proposal focuses on the most important and most repeated concern: after 11 and 15, students should know how to program, and they should be able, on their own, to write a program that solves a problem. Our proposal does address some other concerns, and once we have a new 11 and 15 in place, we will have more room for improvement than we do today.

2. Goals

We set out to develop goals, identify a pedagogy (learning outcomes, foundational ideas, and problem-solving methods), and only then go shopping for a programming language and toolset (compiler and possibly IDE) that will support our chosen pedagogy. Our primary goal is to improve learning outcomes in COMP 11 and 15. The learning outcomes should determine the choice of syllabus, programming language(s), and tools. Students who achieve the desired outcomes in 11 and 15 should emerge with the skills needed to do ambitious work in subsequent courses, and they should also be attractive to potential employers.

Desirable outcomes for 11 include ability to use these concepts and techniques:

- Abstraction (both functional abstraction and data abstraction)
- Recursion
- Data-driven programming
- Contracts
- Systematic methods for problem solving
- Systematic methods for software development

Our statement of purpose for 11 says

> *COMP 11 is a course in problem-solving by computer, where students will learn to solve problems "starting from a blank page," and that will be available to every student who is motivated to work hard, regardless of background.*

"Starting from a blank page" means that a student has some opportunity to design a solution from scratch, as opposed to starting with a design or incomplete software prepared by the course staff.

COMP 15 should refine and expand material from 11, and should also teach data structures. Desirable outcomes for 15 include ability to use these concepts and techniques:

- Reasoning about data types at the abstract level
- Encapsulation, objects, object-oriented design (all words for the same idea)
- Interfaces, implementations, and modular reasoning
- Contracts that include cost models
- Reasoning about and measuring performance
- Generics, which are also called polymorphism
- Programming with compile-time type checking
- Programming at a low level of abstraction with arrays, lists, and trees
- Problem solving at a high level of abstraction, using sets, sequences, and dictionaries (finite maps)
- Problem solving and design at a medium scale, which requires solutions that are split into multiple files

Our statement of purpose for 15 says

> *COMP 15 is a course in data structures that emphasizes modularity, polymorphism, design, reuse, and cost models, and that prepares students to carry their skills into new contexts.*

We have two secondary goals for COMP 15: it should prepare students, at least a little bit, to "pick up" new programming languages on their own. And to continue to serve ECE students, COMP 15 should provide some preparation in C or C++ or a similar language.

Another goal is that students should finish the two-term sequence with a portfolio that is attractive to potential employers. Career Services tells us that employers like to hire students who claim competence in currently popular technologies, but that employers find it even more important to hire students who can solve problems and can communicate well. We will work with Career Services to position our majors as excellent problem-solvers with a solid understanding of computation. We believe that skills in popular technologies are primarily important to non-majors.

Finally, we have a tactical goal: Ben Hescott's teaching assignments, and especially his commitment to course development, must be appropriate for a tenure-track Assistant Professor. We must therefore limit the amount of new development we do in-house.

3. First term: *Program by Design*

For COMP 11, we propose to adopt an introductory course developed at Rice called *How to Design Programs*. The course is supported by a well-regarded textbook from MIT Press (Felleisen et al. 2001) and by the Racket programming language and environment, which are designed for beginning students. The course and the software are part of a larger educational effort called *Program by Design*.

Program by Design has been under development for over 15 years. The work is widely recognized as successful, and it has been used at peer institutions including Brown, Rice, and the University of Chicago. And for this work, Matthias Felleisen, the project's leader, received ACM's Karl V. Karlstrom Outstanding Educator award. The short citation reads "For his visionary and long-term contributions to K-12 outreach programs, innovative textbooks, and pedagogically motivated software." The full citation can be found at `http://goo.gl/slMBZ`.

The *How to Design Programs* course offers significant tactical advantages:

- The course is mature and has been designed to be adopted outside its home institution.
- Every summer, there are free workshops for new instructors.
- The infrastructure to support the course, including software, documentation, labs, and exercises, is truly impressive.

The tactical considerations are important, but we are proposing *How to Design Programs* because of its technical advantages:

- *How to Design Programs* is built around a systematic method of software development: the *design recipe*. The design recipe is grounded in timeless techniques invented in the 1970s and 1980s, including data-driven programming, test-driven development, and formal specification of programs using preconditions and postconditions. (Design recipes have much in common with the ideas and methods that Bruce Molay has infused into COMP 11.) Students start small and learn the full design recipe over the course of the term.

  The design recipe helps teachers as well as learners. When a student "gets stuck," a teaching assistant or instructor can ask such questions as "which step of the design recipe are you stuck on," and "show me your earlier steps." Instructors using *How to Design Programs* report that most students who get stuck are stuck because they ignored the design recipe and tried to go straight to code.
- The pedagogy and methods of *How to Design Programs* are supported by the DrRacket programming environment. DrRacket is a modern graphical development tool, but it is tuned to serve students, not professional engineers. Uniquely, DrRacket can be adjusted to any of several "language levels," which start at "Beginning Student" and go all the way up to "full Racket." The language levels steer students in the right direction by accepting only those programming constructs and techniques that have been taught in class. Moreover, at each language level, if something goes wrong, DrRacket provides error messages that make sense to students working at that level. Beginning students are not only prevented from using advanced features that they do not yet understand, but are also protected from accidentally triggering error messages that make sense only to advanced students or professionals.

The mature, effective pedagogy, together with support from a programming environment that has been shown to work with beginning students, should give us the opportunity to provide a solid foundation in programming, problem-solving, and design, while keeping COMP 11 exciting and enticing.

Implementation at Tufts

How to Design Programs is a complete course, and developing new content is optional. We expect the course to be successful at Tufts as is, and that several of our faculty could teach it well with only modest preparation. (Racket is based on Lisp and Scheme, so some of our faculty have prior knowledge that will help.)

For the first offering, we propose to teach *How to Design Programs* as it stands. In Fall 2013, Ben and Norman will co-teach lectures, possibly with a newly hired lecturer depending on enrollment. Outside of lecture, students will complete labs and projects.

Our lectures are now so big that each student has little time to participate in discussions and ask questions. The labs will address this problem using the new model developed for COMP 11 in 2011–2012. Labs will be small enough so that each student can review, discuss, and practice the main ideas of the week. During lab, TAs will help students work on exercises.

Labs will also reinforce the design recipes and will help students develop the habit of carefully planning, testing, and implementing their designs. Each lab will present and give students practice with a specific basic technique. Some of these techniques, like writing test data first and documenting functions with contracts, are part of the design recipes, and the lab will reinforce ideas from the textbook. Other techniques, like diagramming code using call trees and state diagrams, are less closely connected to the textbook but have already been shown to be successful here at Tufts.

We will develop projects designed for Tufts. We want projects that are connected to real-world problems and are also connected to our departmental strengths. One example of such a project is the "trigrams project" we have just deployed in COMP 11: students use trigram frequencies to identify the natural language in which a document is written. This project helps to demystify Google Translate and also connects with our strength in machine learning. Our students were very engaged and excited to be working on a problem with such an obvious real-world application, and they were particularly pleased to build a classifier and test it using documents they found on the web.

We have also discussed possible projects that are connected to computational biology (string problems) and to visualization (algorithmic botany). In the long run we want a portfolio of exciting projects that will motivate students to continue in our department.

For course staff, we will use mostly undergraduate students who have learned some functional programming in COMP 105 and would like to help teach it. We also hope, at least for the first offering, to get help from one or two mid-level graduate students who use functional programming in their research. All of our course staff will take advantage of the *Program by Design* workshops for teachers, which are typically offered every summer.

4. Second term: *Abstraction + Cost Model = Data Structure*

When we investigated ideas for updating COMP 11, we found several successful offerings. Teaching the first course is a problem in which colleagues worldwide have invested substantial resources.

The second course is another story. There is no obviously good pedagogy, textbook, or programming environment for us to adopt. For COMP 15, we therefore propose to develop a new course here at Tufts.

The new course will be called *Abstraction + Cost Model = Data Structure*, and it will combine classic data structures with the systematic methods of software development for which Barbara Liskov received the Turing Award in 2008. (Today these methods are called "object-oriented design.") We will also teach a little of Tony Hoare's methods of program specification, for which *he* received the Turing Award in 1980.

The approach we want to teach is that data structures arise organically from a designer's need to have a particular abstraction with a particular cost model. As an example here, we use the abstraction of the *set*. We assume that the elements of a set can be totally ordered—a requirement for all search trees, for example. The set abstraction can lead to a variety of data structures, depending on the cost model desired:

- If we have a set of event times for a simulation, and all we want to do is add an arbitrary time and remove the smallest time, at low cost, we can implement a simple heap.
- If we primarily want to remove the smallest time from a set, but we also want cheap set union, we can implement a leftist heap.
- If we want an efficient test for membership in the set, we can use a balanced binary tree. In this case, if we plan never to delete an element from the set (infinite-cost deletion), a red-black tree is probably the simplest data structure. If we need to support efficient deletion, some other data structure may be better.
- If we want a space-efficient set and can afford a *probabilistic* membership test, we can implement a Bloom filter.

Implicit in this example is the idea that for a given abstraction and cost model, we want to minimize programming effort by using the simplest data structure that does the job. We propose to develop a new textbook based on this approach. Such a textbook will be successful with our students and will have an excellent chance of being adopted by other programs. Because Norman already has experience writing a textbook, and because Ben needs time to develop his research program, Norman will be the primary author of the textbook.

Ultimately we would like to take the same approach that Bob Sedgewick has taken with his algorithms texts and that Andrew Appel has taken with his compilers text: offer the textbook in multiple editions which differ only in the programming language used for examples and exercises. For the first draft, we propose to use the language Standard ML.

- Standard ML offers a transition from *How to Design Programs* that will be easy for both teachers and learners. In *How to Design Programs*, a central role is played by *data descriptions*. In the Racket language, these are comments. In Standard ML, they become actual language constructs that are checked and enforced by the compiler. Thus, in the first term, students will be able to develop data-description skills without having to fight with the compiler. At the beginning of the second term, students will see that once the compiler *does* understand the structure of the data, compile-time type checking helps to guide the construction of the code and to eliminate silly errors. Standard ML in particular offers an additional advantage: any time the code contains a case analysis (like a `switch` statement), the compiler checks to be sure that all cases are covered. This extra check helps reinforce the design recipe.
- Standard ML can act as a gateway to the imperative and object-oriented styles used in subsequent courses. Standard ML supports imperative programming with loops, arrays, and mutable heap objects.
- Standard ML supports explicit, separately compiled interfaces. This language feature is central to our approach: it gives us a way to talk about an abstraction independently of any implementation. There is also a connection with Java interfaces.
- Standard ML has a published language definition and a number of implementations to choose from, ranging from a simple interactive interpreter to a state-of-the-art optimizing compiler. And there is considerable academic literature to draw on, including Chris Okasaki's superb 1999 monograph on purely functional data structures.

To see how will Standard ML might work in COMP 15, we've done some experiments with existing projects. In one afternoon, we implemented two kinds of heaps, including leftist heaps. While the *analysis* of a leftist heap is probably too advanced for 15, the implementation is not: the only nontrivial operation, heap merge, takes just 16 lines of code. We also investigated how well Standard ML might work with the *Song Search* project which has been used as a capstone in 15. Ben reports that the ability to write and compile an interface exposes a lot of the design decisions that his students have struggled with, and it brings those design decisions to light *before* the students have committed to any implementation.

After 11 and 15, students' problem-solving and programming skills should be portable to a mainstream language they have not yet studied. We propose to evaluate our progress toward this goal by devoting the last three weeks of COMP 15 to the acquisition of new programming languages. Each student will be required to learn a new language that he or she has not yet studied, and to reimplement one of the class projects in the new language. We will provide course modules for different options. At minimum, we will offer modules for a popular object-oriented language (Java), a popular scripting language (Python or Ruby), and a popular machine-level language (C or C++), and so on. Students in ECE will be advised to learn C.

5. Adjustments in subsequent courses and in the *Bulletin*

Our proposal is intended to help prepare our students to do better in upper-level courses. Nobody can foresee all the consequences, but we do expect major changes in 40 and minor changes in 105 and 160.

***11 and 15 in the* Bulletin** The changes we propose fit the current course descriptions in the *Bulletin*. But the treatment of AP credit will have to change. Students who want to use AP credit to skip 11 and start with 15 are unlikely to succeed, and we propose to forbid students from using AP credit to go straight into 15. Students with multiple AP credits who want to skip *both* 11 and 15 may or may not be well prepared, but we propose to permit it, primarily because of concerns about recruiting. Freshman advisers should discourage students from using AP credit to skip 11 and 15.

COMP 40 COMP 40 is the course most affected by the changes we propose:

- All of the difficult software-development technique which is currently taught in 40 will be moved to 11 and 15. This technique includes contracts, interfaces, modular software construction, and polymorphism.
- One purpose of COMP 40 is to teach how computations are executed on the machine. A revised 11 and 15 will teach control

and data at a high-level, so 40 will have to cover low-level concepts from scratch. Most important among these are pointers, pointer arithmetic, and explicit memory management.

At present, COMP 40 combines instruction in machine-level programming with an intensive programming experience—and it's too much. By moving the bulk of programming technique into 11 and 15, we will be able to reduce the load in 40 to sensible levels, and we will be able to continue to offer the kinds of projects that our stakeholders have come to value.

COMP 160 Several faculty have said 15 needs more programming and less theory. As part of teaching contracts with cost models, we will continue to teach students in 15 how to analyze the worst-case space and time costs of simple algorithms and data structures. But we don't want to limit students' programming experience to data structures that are easy to analyze. In particular, Standard ML makes "hard" data structures like leftist heaps or red-black trees very easy to implement. Students may well enter 160 with unbalanced portfolios: their skills in programming are likely to outstrip their skills in analysis of algorithms. We will work with those teaching 160 so that everyone's expectations are clear.

6. Summary

The three-course sequence 11/15/40 is currently popular and successful, but some of that success comes at a high cost. And we have reached a point where further incremental improvements are difficult. By rebalancing the technique taught in these three courses, we plan to preserve our currently good outcomes, continue attracting many students to our courses, and create new opportunities for making things even better.

References

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2001. *How to Design Programs: An Introduction to Programming and Computing.* MIT Press, Cambridge, MA.