Personal Statement

Franklyn Turbak
Department of Computer Science
Wellesley College

October 2, 2001

In this document, I summarize my achievements and future plans in teaching, research, and service at Wellesley College for my tenure case. The document is organized into three sections:

1. **Teaching:** Describes my teaching philosophy, summarizes my course development work, and discusses some future plans for teaching.

2. **Research:** Summarizes my research projects, the student research I have advised, and future research plans. Unlike my Aug. 1 *Research Summary*, which was targeted at external reviewers who are experts in the programming language field, this section is targeted at a general audience. In this section, I also discuss the evaluation of computer science research — in particular, the relative importance of conference papers vs. archival journal publications.

3. **Service:** Summarizes my service contributions to Wellesley College, the Wellesley College Computer Science Department, and the computer science community.

In addition to the main document there are several appendices containing auxiliary materials:

- Appendices A–F contain detailed descriptions of my course development efforts and links to sample teaching materials on-line. Hardcopies of these materials are included in binders entitled *Personal Statement and Supporting Materials*. A web page with links to all sample materials can be found on-line at `http://cs.wellesley.edu/~fturbak/tenure/support.html`.[1] The complete set of teaching materials I have developed since Fall, 1997, are available on-line at `http://cs.wellesley.edu/~fturbak/teaching.html`.

- Appendix G contains a copy of the Computing Research Association's Best Practices Memo: *Evaluating Computer Scientists and Engineers for Promotion and Tenure*.

- Appendix H contains a sample copy of the questionnaire I developed for the Committee on Minority Recruitment, Hiring, and Retention.

[1] Various URL links, such as `http://cs.wellesley.edu/~fturbak`, are sprinkled throughout this document. These links are "active" (i.e., you can click on them) when the on-line PDF version of this document is viewed inside a browser such as Netscape Navigator or Internet Explorer using the Adobe Acrobat plug-in (standardly available on Wellesley College computers). The on-line PDF version of this document can be found at `http://cs.wellesley.edu/~fturbak/tenure/statement.pdf`.

1 Teaching

1.1 Teaching Philosophy

Becoming A Computer Scientist

I will never forget the week that I became a computer scientist. It was Spring break of 1981, and I was a freshman taking *6.001: Structure and Interpretation of Computer Programs*, MIT's introductory computer science course. This course used the Lisp programming language to explore the fundamentals of programming. We had an assignment on a *meta-circular interpreter* for Lisp – a Lisp program that can evaluate other Lisp programs. The notion that a program could execute other programs — even itself! — was both perplexing and mysterious to me. I spent an entire week trying to understand what the meta-circular program did and how it worked. Finally, after carefully tracing through the execution of the program on a concrete example, I had a Eureka experience: I realized that the meta-circular interpreter was a concise and precise encoding in Lisp of the informal rules that we had been taught throughout the semester for evaluating Lisp programs. What had been veiled in a cloud of mystery suddenly became clear, and I was overwhelmed with a sense of beauty and wonder for this profound and elegant idea.

This was a transformative experience. Although I had come to MIT to study electrical and mechanical engineering, after this experience I had an unquenchable thirst for knowledge about computer science and could not imagine a career in any other field. During the twenty years since that experience, I have found computer science to be a treasure-trove of simple and powerful ideas, clever techniques, challenging projects, and fun problems. But the meta-circular interpreter experience did more than just "convert" me to a computer scientist. It caused me to reflect on the learning process and think about pedagogical approaches that would help to demystify the beautiful ideas of computer science and make them more accessible to a wider audience. I started developing computer science teaching materials when I became a teaching assistant for 6.001 in Spring, 1984, and have continued to actively produce such materials (e.g., course notes, examples, and assignments) ever since then.

Big Ideas

In my courses, I strive to create an environment where students not only learn the concepts in computer science that I find so exciting, but they also develop an appreciation for the beauty of these concepts. While it may be too much to expect that my courses will induce in my students transformative experiences like the one I experienced (although I am very happy when this does happen!), I do expect that they will leave my courses with a firm grasp on the "big ideas" of computer science – i.e., the key concepts at the core of the field. Examples of such big ideas are:

- *Procedural Epistemology*: The essence of computer science is imperative ("how to") knowledge, as opposed to the declarative ("what is") knowledge of mathematics and other sciences. Not only can programs be executed by computers, but they are a means of transmitting imperative knowledge from one person to another.
- *Abstraction*: To separate the concerns of program clients from program implementers, patterns of computation should be encapsulated in "black boxes" with simple interfaces that hide unimportant details.
- *Modularity*: For flexibility and robustness, systems should be composed out of reusable mix-and-match parts.

- *Divide, Conquer, & Glue*: A fundamental technique for solving a big problem is breaking it into smaller problems and combining the solutions to these. Recursion and iteration are instances of this technique in programming, while induction and loop invariants are applications of these techniques in proof theory.
- *Programs as Data*: A program is a data structure that can be executed, analyzed, created, and transformed by other programs.
- *Formal Reasoning*: Not only can logical deduction be formalized and in some cases automated, but there is a deep connection between logic and computation.
- *Uncomputability*: Because there are an uncountable infinity of mathematical functions but only a countable infinity of programs to express them, there are many functions (including some very desirable ones) that can never be expressed as programs.
- *Computational Models*: In order to predict how a program will execute, it is essential to understand the model of computation underlying the programming language in which the program is written.
- *Tradeoffs*: Problems often have multiple conflicting goals that are mutually unsatisfiable. In these cases, it is necessary to make tradeoffs that involve relaxing one or more goals.
- *Iterative Nature of Problem Solving*: Since problems are rarely solved in a single pass, problem solving usually requires iterating a cycle of specification, design, implementation, testing, and debugging.
- *Real vs. Ideal*: Abstractions and models introduce idealizations that may not accurately reflect real aspects of the problem being solved or the situation being modeled.

The above ideas are so important that they transcend a particular course, and even the particular discipline of computer science. As would be expected of fundamental concepts, each of the ideas presented above arises in several courses, so students who take more than one course encounter them multiple times in different contexts. Moreover, many of the ideas and techniques are applicable to other domains – a fact that is particularly relevant in a liberal arts setting, which encourages the exploration of connections between disciplines. For instance:

- Procedures are not only encoded in programs executed on computers, but they are also encoded in recipes, instructions, musical scores, theatrical scripts, and architectural blueprints executed by human beings.
- Principles like abstraction and modularity are essential for controlling complexity in any field involving the creation of artifacts or use of models, including engineering, the physical and social sciences, and even the arts.
- Problem solving techniques learned in programming can be adapted to solving problems in almost any domain.
- Understanding the iterative design cycle, the limitations of models, and the notion of tradeoffs gives insight into many aspects of the modern world, from engineered artifacts to political processes.

- Reading, writing, and debugging programs hones reasoning skills that are essential for evaluating and/or crafting a logical argument on any topic.

With its emphasis on design and problem solving, computer science can be viewed as a kind of abstract form of engineering. Indeed, many of the big ideas sketched above are characteristic of engineering in general, not just computer science. Although there is a long historical precedent of divorcing engineering from a liberal arts education, I firmly believe that these engineering concepts have a central place in a liberal arts education. Not only do they give students insight into a world full of designed artifacts, but they encourage students to become designers and builders themselves – producers of technology rather than just consumers of it. Robbie Berg and I have written at length about this issue in our paper *Robotic Design Studio: Exploring the Big Ideas of Engineering in a Liberal Arts Environment*.

In addition to transcending disciplines, the big ideas of computer science transcend the particular technology that is used to teach them. For example, particular hardware, operating systems, programming languages, and application software are used as vehicles for presenting and experimenting with the big ideas. While it is important for students to be familiar with the current technology, it is important to impress upon them that the current technology is *not* the subject matter. Particular technology is ephemeral, but the big ideas are not! Moreover, while computer science is a discipline of incredible beauty, much of the popular technology is surprisingly ugly from the perspective of understandability, usability, reliability, and extensibility. Since I hope that my students will someday help to improve this sad state of affairs, I consider it imperative to teach them how to tease apart important concepts and techniques from poorly designed pieces of software in which they might happen to be embedded.

Presenting Material

When preparing to teach a course, I first determine what big ideas are covered in the course, and then strive to organize the course to highlight these ideas. This "first principles" approach to teaching sometimes leads to non-standard topics or an unconventional ordering of topics. For example, in CS111 (Introduction to Programming and Problem Solving), considerations involving the divide/conquer/glue problem-solving methodology compel me to teach recursion before iteration (see Section A and my paper *Teaching Recursion Before Iteration in CS1* for more details). Although such an approach is sometimes used in the teaching of so-called functional programming languages, it is highly unusual in the teaching of object-oriented languages such as Java (the programming language used in CS111). Nevertheless, I believe that organizing CS111 in this way gives the students a much better view of the "big picture" than traditional approaches for teaching Java. Other examples of unconventional course organizations I have developed include the interpreter-based approach I use in Programming Languages (CS251) (see Section E) and the onion-skin approach I use in my Compiler Design (CS301) class (see Section F). A disadvantage of my unconventional course organizations is that they make it difficult to use existing textbooks and problems. For this reason, most of my classes are based on notes and assignments that I have developed from scratch.

When it comes to presenting particular topics, I expose students to several different explanations of the same material. Many years of teaching experience has taught me that different students learn in different ways: some learn best from principles, others from examples; some prefer pictorial explanations, others prefer text; some like formal derivations while others like intuitive arguments. For this reason, even in courses where there is a textbook consistent with my course organization (such as CS231, Fundamental Algorithms), I present alternative approaches to the material in course notes.

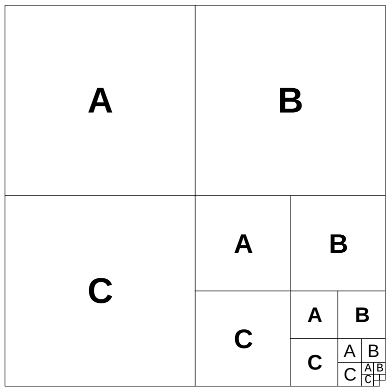
Wherever possible, I present both an informal and intuitive explanation of concepts (often involving pictures) as well as a formal symbolic explanation. Both of these are important: in my experience, intuition is incredibly valuable for planning the solutions to problems, but formal techniques are necessary to work out the details. As a concrete example of formal vs. informal explanations, consider calculating the infinite sum

$$S = \sum_{k=1}^{\infty} \frac{1}{4^k} = \frac{1}{4} + \frac{1}{16} + \frac{1}{64} + \ldots$$

For $0 < c < 1$, it is not difficult to show that the infinite sum $\sum_{k=0}^{\infty} a_0 \cdot c^k = a_0 + a_0 \cdot c^1 + a_0 \cdot c^2 + \ldots$ has the closed form solution $\frac{a_0}{1-c}$. In the case of calculating S, $a_0 = \frac{1}{4}$ and $c = \frac{1}{4}$, so the formula gives

$$S = \frac{\frac{1}{4}}{1 - \frac{1}{4}} = \frac{\frac{1}{4}}{\frac{3}{4}} = \frac{1}{3}$$

The same result can be achieved informally by a pictorial argument. Imagine dividing a unit square into quarters, labelling three of the quarters A, B, and C, and recursively subdividing the fourth quarter in the same fashion, as pictured below:



It is easy to argue that the sum of all squares labelled A (as well as of those labelled B and C) is the desired sum S, and since, in the infinite limit, the entire area of the unit square is completely partitioned into squares labelled by one of these three letters, it must be that $3S = 1$, so $S = \frac{1}{3}$.

Hands-on Activities

Although I focus significant energy on developing and presenting explanations of the subject matter of my courses, I recognize that this is only one facet of the learning process. While substantial amounts of information may be transmitted from teachers to students in the form of lectures, course notes, etc., such information transfer is just a prerequisite for learning. The real learning takes place when students actively apply this rather abstract form of knowledge in concrete contexts. Only then do they begin to develop a sense of personal ownership of the ideas that I believe is essential for true understanding. This belief has its roots in the constructionist learning theories of Piaget and Papert, which posit that people learn best when they are engaged in the design and construction of personally meaningful artifacts.

I have made a conscious effort to infuse my courses with hands-on activities that are inspired by constructionism. This is most evident in the Wintersession Robotic Design Studio course (CS115) that I teach with Robbie Berg. After being introduced to the fundamentals of robotics during the first half of the course, students spend the second half of the course working in groups to build a robot of their own design. The ground rules for these projects are intentionally very loose; students can build anything they imagine, subject to constraints on materials and time. Our experience in this course is that students are not only incredibly creative in their projects, but they learn a tremendous amount about robotics and general engineering principles in the process.

Most of my other courses are designed in such a way that students get significant hands-on experience with the subject matter in the context of programming activities:

- *Introduction to Programming and Problem Solving (CS111)* is organized around a collection of "microworlds" (see Section A) that serve as "pedagogical sand boxes" where students experiment with fundamental programming and problem solving techniques. Every week, students work on numerous challenging programming problems in these microworlds. Additionally, in the context of various optional contests and extra credit assignments, students are encouraged to extend programs from class and work on problems of their own design.

- In *Data Structures (CS230)*, students implement numerous data structures and algorithms by writing programs in Java (see Section C). Some of the assignments (such as the Mancala game) require students to write large programs from scratch, a significant departure from CS111, where assignments typically involve modifying existing programs.

- Although *Fundamental Algorithms (CS231)* is a theoretical course with no formal programming requirement, I have recently begun to encourage students in the class to implement various of the algorithms we study in a programming language of their choice. Many of our students want to learn programming languages (e.g., C++ and Perl) not formally taught elsewhere in our curriculum, and this provides them with an excellent opportunity to do so.

- In my *Programming Languages (CS251)* course, students modify and extend interpreters for numerous "mini-languages" that we study in the course (see Section E). The structure of this course differs from most other such courses at the undergraduate level, which either have no programming component, or involve programming in languages from several different paradigms. It is my belief that experimenting with programming language implementations gives students deeper insight into the dimensions of programming languages. It empowers them to be programming language designers and implementers, so that they can be producers, and not just consumers, of programming language technology.

- In *Compiler Design (CS301)*, students work in groups to implement a series of compilers (see Section F). Each of these is a large and complex program, and except for some provided tools, the students do their programming "from scratch".

In the above courses, there is a tension between the desire to cover the litany of standard concepts and the constructionist ideal of having students explore subject matter in the context of open-ended projects. Not only do open-ended projects give students more chance to "own" the material, but they also force them to grapple with real-world issues like refining fuzzy problem descriptions, exploring blind alleys, and determining when a project is "done". Thus far, I have usually resolved this tension by designing programming problems that have a required part with a constrained specification and an optional part involving more open-ended exploration. I also encourage my students to undertake an individual study project or research project while at Wellesley, so they get at least one substantial open-ended experience in computer science as undergraduates.

Challenging Assignments

The assignments I give in my courses are typically rather challenging. Indeed, I have a reputation for giving some of the most difficult and time-consuming assignments in our department, if not the entire college. Because I believe that most student learning takes place in the context of working on concrete problems, I carefully design each problem with a clear pedagogical goal of illustrating one or more concepts. For students who understand the concepts, the problems are relatively straightforward, and some students do finish them quickly. However, the problems are more challenging for students who have not yet mastered the concepts, including many who mistakenly think that they understand the material just because they have followed what I have presented in lecture. This is OK; they understand the material much better by the time they finish the problems. Athletes do not build muscles by lifting feathers.

Student Feedback

One of the most delightful aspects of teaching at Wellesley College is that students rise to the challenge of my problems and are fair in evaluating them. Based on the feedback I have received on my SEQs, students generally feel that while the assignments are time-consuming, the time was well spent in terms of what they learned. I am sensitive to the fact that the time students spend on my assignments is time they are not spending on other important aspects of a liberal arts education, such as extracurricular activities and leisure time. I ask students to keep track of and report the amount of time they spend on each problem, and use this information to revise assignments the next time I teach the course.

Deadlines and Extensions

Given the hectic nature of students' schedules (and my own!) I understand that there are times when it is not possible for them to complete an assignment on time. It really does not matter to me *when* students complete their assignments, only *that* they complete them. However, completing work late can have a cascading effect and put students so far behind that they cannot catch up.

With these considerations in mind, I have pioneered *lateness coupons*, which provide students with a limited amount of flexibility in terms of deadlines. In many of my courses, students receive ten lateness coupons at the beginning of the semester, which are not transferable between students. One lateness coupon allows one assignment to be handed in one day late. A student could use her coupons to turn in each of ten assignments one day late, or turn in a few of these assignments several days late. Once all lateness coupons have been used, late assignments receive no credit.

Lateness coupons work well in practice. It is possible to abuse them (e.g., use all coupons on the first few assignments and then be impossibly far behind), but such abuses are extremely rare. Indeed, many students never use any coupons, but those who do indicate that they provide valuable flexibility.

Collaboration

Collaboration plays an important role in my classes. Even though I generally require that they compose any written solutions or programs on their own, I strongly encourage students to work on problems in groups. Based on my own personal learning experiences, I believe that valuable learning takes place when students solve problems in a group. It is often the case that the knowledge of individual students is incomplete, but what is missing or partially formed differs from student to student. When working in groups, students share what they know and (think they) understand with others and at the same time improve areas in which their understanding is weak. In particular, group work allows every member to play the roles of both student and teacher; the well-known saying about teaching being the best way to learn a topic is particularly relevant in this context. Even a group member playing the role of "student" can make valuable progress by expressing in words what it is that she does not understand. Although students can also do this in the presence of a faculty member, many students are less shy about asking questions of their peers than of their instructors. Working in groups also gives students important practice in honing their interactive skills and preparing them for a world in which much work is group-oriented.

When it comes to programming, having a pair of students work at a single computer console seems to be particularly beneficial. At the beginning of the summer of 2000, my research colleague Patty Johann and I asked our six summer research students to work in pairs on various programs designed to introduce them to concepts and techniques for their summer research. Patty and I were working in the same computer lab as the students, so we were able to closely observe the kinds of interactions and conversations that took place between group members. We were impressed and excited by what we observed. Rather than just jumping in and writing code, which is what students are tempted to do when working alone, the students first discussed the problems and various approaches for solving them. There was not always agreement on the best solution strategy; this meant that each student was put in a position where she had to argue the benefits of her approach relative to the other one. This explicit planning phase helped the students to avoid "blind alleys" of exploration. In the coding phase, students swapped between roles as "driver" (the one writing the code) and "passenger" (the one looking over the driver's shoulder). Passengers often noted bugs in the code of the drivers. When it came to debugging, it was clear that two heads were better than one.

Inspired by this experience, I have been experimenting with pair-based programming in my Programming Languages course (CS251), where it was optional, and my Compiler Design course (CS301), where it was required. Although there are sometimes personality clashes and time scheduling problems between pair members, the pair-based approach to programming seems to work well in practice for the reasons Patty and I observed with our research students. I am considering experimenting with pair-based programming in earlier courses, such as CS111 and CS230. Interestingly, I recently learned that several software engineering managers are advocating that industry adopt a programming methodology called Extreme Programming, a cornerstone of which is that all code should be written by programmers working in pairs at a single computer monitor. This style of programming is claimed to yield higher quality programs in a shorter amount of time than traditional programming by two individuals.

Fun

Perhaps because computer science is such a young discipline and involves a fair amount of problem solving and puzzles, the computer science community is rather playful. I like to instill this spirit of playfulness into my teaching, because I believe that many students are often more receptive to learning when they are in a playful mood. Many of my assignments involve some sort of game or puzzle and/or have a humorous story line. For instance, my CS111 students explore concepts like abstraction, recursion, and backtracking in the context of a grid-shaped world where creatures called "buggles" drop and pick up bagels. My CS230 students get experience with object-oriented programming and graphical user interface design in Java by modifying a blackjack program and implementing a video Mancala game from scratch. In my robotics class, it is hard *not* to be in a playful mood when building with LEGOs. In recent years, I've set aside the last day of class in almost every course to be a final exam review session structured as a Jeopardy game played by teams of students vying for snack prizes from Trader Joe's. Via activities like these, I try to do my part in realizing the vision of computer science luminary Alan Perlis, who said, "I hope the field of computer science never loses its sense of fun."

1.2 Course Development

The dynamic nature of computer science means that computer science courses are always evolving. Unlike in more stable fields, it is rare to be able to teach a course using lectures, notes, and assignments unchanged from a previous semester. Even in fairly stable courses (like CS231, Fundamental Algorithms), I enjoy thinking of new ways to present concepts, experimenting with topic order and new topics, and developing new examples and problems every time I teach a course.

In my twelve semesters of teaching at Wellesley, I have developed extensive course materials in each of the six courses I have taught. Below, I give an extremely brief summary of my course development efforts in each course. Appendices A–F present more detailed descriptions of these efforts and links to sample materials. All course materials that I have created since Fall, 1997 are on-line at `http://cs.wellesley.edu/~fturbak/teaching.html`; interested readers are encouraged to browse this complete set of materials in addition to the samples in the appendices.

- *Introduction to Programming and Problem Solving (CS111)*: In 1997, I led the effort to use Java in place of Pascal as the programming language in this introductory course for majors. This was not just a cosmetic change in programming language, but a change in paradigm that required rebuilding the course from the ground up. I organized many of the lectures and assignments around a number of *microworlds* , simple but rich environments I developed for exploring key programming concepts and techniques. I also developed the *Java Execution Model* to explain the execution of Java programs at the level of an abstract machine. A "first principles" approach to teaching the big ideas of the course led me to adopt a non-standard ordering of topics, such as teaching recursion before iteration and linked lists before arrays. The rationale for these choices is discussed in my paper *Teaching Recursion Before Iteration in CS1*.

- *Robotic Design Studio (CS115)*: Since Wintersession '96, Robbie Berg and I have taught this course, which we developed as a way to engage students in design and hands-on building activities. The course has no prerequisites, and attracts students with a broad range of backgrounds and interests. The first half of the course introduces students to the basics of robotics via a series of challenges in which they study and modify existing robots and build simple ones of their own. In the second half of the course, students work in teams on an open-ended project in which they build a robot of their own design. The course culminates in a public exhibition that has become a signature event at the end of January. The philosophy behind the course is explained in our paper *Robotic Design Studio: Exploring the Big Idea of Engineering in a Liberal Arts Environment*

- *Data Structures (CS230)*: After teaching this course in Pascal for many semesters, I moved it to Java in Spring '98. As in CS111, this required significant effort. For both the Pascal and Java versions of the course, I have developed numerous programs to illustrate data structures and algorithms in class and on assignments.

- *Fundamental Algorithms (CS231)*: My key innovation in this course is the use of *worksheets* that we flesh out in class to present intuitive explanations and concrete examples of the mathematical techniques used in the textbook. Functional programming techniques I have encountered in my research have allowed me to present alternative, and in some cases much simpler, versions of algorithms in the textbook.

- *Theory of Programming Languages (CS251)*: The organization of this course is based on my belief that the only way to appreciate the dimensions and design choices of programming languages is to study and modify their implementations. Since implementations of real programming languages are too complex, we study interpreters for simple but representative *mini-languages*. I have designed a series of mini-languages in which each successive language is built by extending its predecessor, and have implemented interpreters for all of these languages. This allows for an "onion-skin" approach in which students are introduced to the features and dimensions of programming languages in layers.

- *Compiler Design (CS301)*: As in CS251, here I also use an "onion-skin" approach to introduce students to the theory, tools, and techniques of program translators. I defined a series of successively more complex subsets of Appel's *Tiger* language. After implementing a complete compiler for the simplest of these languages, students have acquired a working knowledge of the components of a compiler and are well-prepared to to explore more sophisticated features and techniques.

1.3 Teaching Impact

My course development work has had a major impact on the teaching of numerous computer science subjects at Wellesley. Even when I am not teaching a course, many of the materials I have developed continue to be used by other professors teaching the same course. This is especially true in CS111, CS230, and CS231.

My course development work has also had an impact on teaching at other institutions. For example:

- *Introduction to Programming and Problem Solving (CS111)*: I presented aspects of my CS111 course in a panel discussion at the Consortium for Computing in Small Colleges Third Annual Northeastern Conference (CCSCNE-98) at Sacred Heart University in April 1998, and also in a paper and talk at the Fourth Annual version of this conference at Providence College in April 1999. Furthermore, all course materials for CS111 are freely available on the web, and others are encouraged to use the materials and adapt them to their needs.

  One impact of these dissemination efforts was that they persuaded the Middlebury College Computer Science Department to switch to Java rather than to C in their introductory programming course. Indeed, since Fall, 1998, Middlebury has patterned its CX 121 Fundamentals of Computing Course after Wellesley's CS111, using many of the same microworlds and assignments. The CS111 materials have been reviewed and in some cases adopted by several other professors designing or teaching introductory programming courses in Java. These include Stephen Bloch (Adelphi University), Kim Bruce (Williams College), Bob Muller (Boston College), and Adonis Symvonis (University of Ioannina, Greece).

- *Robotic Design Studio (CS115)*: Robbie Berg and I have led two workshops based on our Robotic Design Studio course: a two-day workshop at Colby College in October, 1997 (sponsored by the New England Consortium of Undergraduate Science Education), and a half-day workshop at CCSCNE-98 at Sacred Heart University in April 1998. We also wrote a paper for and gave talks at the American Association for Artificial Intelligence Spring Symposium on Robotics in Education held at Stanford University in March, 2001. Additionally, since the early days of our course, we have posted all course materials on the web to encourage their use elsewhere.

These dissemination efforts have inspired robot courses and robot research modeled after our course at several other schools. After our Colby robotics workshop, Bates, Bowdoin, Colby, and Middlebury colleges all instituted either a robotics course or independent projects in robotics. Our materials have also inspired courses or have been used for projects by Alice Dean at Skidmore College, Elizabeth Adams at Richard Stockton College, Lonnie Fairchild at SUNY Plattsburg, and Bill Margolis at the National University of Samoa.

Because of our promotion of robot exhibitions as an alternative to robot competitions, the organizers of the national Botball robot tournaments[2] have invited Robbie and me to be members of the Boston Botball Committee next year. Our goal is to experiment with extending Botball tournaments to include non-competitive robot exhibition activities so that they appeal to a wider audience.

- *Compiler Design (CS301)*: Olin Shivers at Georgia Tech has reviewed my onion-skin approach for teaching compilers and is considering adapting this approach to his compiler class next year.

- *Graduate Programming Languages (MIT 6.821)*: Drafts of the notes I co-authored for MIT's graduate programming languages course have been used by professors for similar courses at several other universities. These include Kathy Yelick at Berkeley, Andrew Myers at Cornell, and Jan Komorowski at the Norwegian University of Science and Technology.

1.4 Future Teaching Plans

I have numerous long-term plans for enhancing existing courses, developing new courses, and disseminating the results of my course development efforts.

- Turning the MIT 6.821 course notes into a graduate programming languages textbook is a high priority. There are only a few chapters (particularly the one on compilation) that need work.

- While all CS111 materials are publicly accessible, the lack of a textbook that covers the topics (many of which are non-standard or presented in a non-standard order) and microworlds used in the course hinders its adoption elsewhere. I plan to write a textbook that will enhance the teaching of CS111 at Wellesley and make it easier to teach elsewhere.

- I plan to flesh out the current CS251 notes and extend them with material covering object-oriented programming and logic programming. My eventual goal is to write an undergraduate programming languages textbook based on these notes. Along the way, I plan to disseminate my interpreter-based approach to teaching programming languages via talks and workshops.

- In my algorithms course (CS231), I have recently been experimenting with presenting many algorithms in a graphical notation using tree rewriting and graph rewriting rules. Since these experiments have gone well, I am eager to explore how many of the algorithms in the course can benefit from being presented in this fashion. In this regard, it would be helpful to have tools for expressing and visualizing the execution of rewriting rules; developing such tools is on my research agenda (see Section 2.3).

[2] `http://www.botball.org`

- Based on my experience using the ML programming language in my research and teaching it in CS251 and CS301, I think that it is far superior to Java for introducing many of the big ideas of computer science. Since ML is particularly well-suited for expressing computations on tree-shaped data structures, I plan to experiment with using ML to teach CS230. Because ML is not well-known in the general computer science community, I plan to give tutorials to help popularize it.

- I am interested in developing several new courses, some of which are sketched below. I am by no means an expert in these areas, but plan to become fluent in them by developing and teaching these courses.

  - *Concurrent and Distributed Systems*: Almost all of our programming courses assume a *sequential* model of computation in which a program is being executed by single computational agent. But many problems are easier to decompose and/or can be solved more efficiently in a *concurrent* model where there are multiple computational agents. This course will present models of concurrent computation (e.g. process calculi, threads, message passing) and discuss key issues in concurrency, such as communication, synchronization, fairness, and deadlock. In contrast to CS331 (Parallel Machines and their Algorithms), which focuses on data-parallel decompositions of problems onto closely coupled, homogeneous processors, this course will emphasize control-parallel decompositions onto loosely coupled, perhaps heterogenous, processors. Students will study, design, and implement concurrent algorithms in several languages. Examples will be taken from robotics, graphical user interfaces, document processing, operating systems, and networks.
  - *Computer Security and Privacy*: Hackers, encryption, viruses, digital copyright, public surveillance software – hardly a day goes by when there are not major news stories involving the interaction between computers and privacy. This course will explore technical, ethical, and legal issues involved in protecting information and privacy in the digital age. Due to the interdisciplinary nature of such a course, I will seek collaborators in other departments, particularly philosophy and political science, to help develop the course.
  - *Automated Deduction*: The past decade has seen an explosion in work on systems on the automated manipulation of mathematical proofs. There are systems for automatically checking proofs, fleshing out skeletons of proofs, and even writing some proofs from scratch. In this course, we will study systems of logic and their realization in theorem proving systems like Isabelle, Elf, Coq, and Nuprl. We will explore the deep ties between programming and proof construction and will "program" proofs from several domains of mathematics and computer science.

2 Research

2.1 Research Overview

In this section, I present a high-level description of my research area and research projects and describe how they fit into the "big picture" of computer science. A detailed description of my research projects, publications, and roles in collaborative research, can be found in my August 1 *Research Summary*.

Modern Software

The state of modern software is disgraceful. Although we are often dazzled by the amazing features of the latest release of our favorite software packages, we spend a disproportionate amount of time cursing software that is tricky to install, difficult to use, unreliable, insecure, and incompatible with previous releases and other applications. We have been browbeaten into accepting that applications will regularly crash our computers, propagate viruses, corrupt our files, and steal our private information. To use a modern application, we typically have to sign a waiver that effectively says that there are no guarantees whatsoever that the software will work as advertised. On the side of software production, companies have to wrestle with large and complex projects that are poorly designed, behind schedule, over budget, insufficiently tested, and full of bugs.

In almost any other industry, this state of affairs would incite consumer rebellions, cause the launching of congressional investigations, and lead to a universal cry for improved standards and guarantees. And yet this does not happen. Why? My explanation is that people have become accustomed to the status quo and are not aware that the situation can be dramatically improved. Great strides have been made in the technology for building reliable and secure software systems. Although it may never be possible to guarantee that every complex program always behaves as specified, advances in programming languages, program testing and verification, and software engineering methodologies enable the construction of applications that are significantly more robust and, in some cases, even cheaper to produce than current software packages.

The software industry has been slow to adopt these new technologies for several reasons. First, there has been little demand from consumers to improve standards, so there is little motivation to do so. Second, most companies have invested heavily in software production technology that is decades old. Although there may be a long-term benefit to adopting new technologies, the short-term costs are high. Finally, there are some disadvantages to the new technologies that stand in the way of their adoption. A common disadvantage is that the newer technology often yields software that is somehow less efficient than that produced with existing technology.

Expressiveness vs. Efficiency

My area of research is the design, analysis, and implementation of expressive programming languages. A programming language is "expressive" to the extent that it enables a programmer to clearly and concisely encode in the language whatever computational patterns she imagines. There is often a wide gap between the computational conceptions of a programmer and what she can write down in code. Bridging this gap requires the programmer to translate between her "mental language" and the programming language – a difficult step where many errors can be introduced.

Expressive languages support means of abstraction that allow a programmer to effectively extend the language by capturing and naming arbitrary computational idioms. Such languages also provide ways to modularize programs into mix-and-match components that can be independently developed, tested, and debugged and reused in many different programs.

A problem with expressive programming languages is that there is a fundamental tension between expressiveness and efficiency. The very same abstraction and modularity features that make a programming language expressive stand in the way of efficient program implementation. Abstraction can be viewed as hiding details inside of "boxes". Naively executing a program that makes heavy use of abstractions involves creating, manipulating, and throwing away lots of boxes – operations that would not be necessary in a program that did not use abstractions.

Since boxes require space, and manipulating boxes takes time, programs using abstractions typically require more time and space resources than those written without such abstractions. The difference in resource consumption can be significant – an abstract high-level program can incur a time or space cost that is orders of magnitude larger than the cost of a carefully written low-level program. In some cases, costs are incurred by the mere presence of abstraction features in a language, whether or not they are actually used in a particular program.

Some programmers are willing to accept the efficiency costs associated with expressive languages. These costs are often offset by the benefits of using expressive languages, such as programs that takes less time to write and debug and which are more reliable. A fast program that does not compute the correct answer is not very valuable! In many applications, modern computers are so fast that the difference in efficiency is not noticeable for many problems. Even in cases where it is noticeable, programming at a higher level sometimes facilitates the implementation of more efficient algorithms, which can more than offset the inefficiencies due to expressiveness.

Nevertheless, many programmers and program managers resist using expressive languages. For applications used by thousands, even millions, of people, efficiency matters. Customers may complain about poor reliability, but they would probably complain even louder about a more reliable program that was ten times slower. Moreover, expressive languages like Scheme, ML, and Haskell are not well-known outside the ivory towers of academia and research institutions.

Due to these factors, the most popular programming languages in industry today are C and C++, relatively low-level languages in which it is notoriously difficult to craft reliable and secure software. However, because decades of work have been invested in the compilers for these languages, C and C++ programs are very efficient – a major reason for their popularity. One encouraging piece of news is that Java, a relatively expressive language, has begun to displace C and C++ in some areas.

Improving the Efficiency of Expressive Languages

The overarching goal of my research is to reduce the tension between expressiveness and efficiency in programming languages. That is, I want programmers to be able to have their expressiveness cake and eat it (efficiently) too! I am not alone in this quest; I am a member of a cadre of enthusiasts for expressive languages who aim to significantly reduce the efficiency costs associated with expressive language features. Over the past 25 years, this group has made much progress toward this goal. Today, using the best ML compilers, it is not uncommon for ML programs to execute in a time that is within a factor of two of a similar C or C++ program compiled with standard compilers. For some tasks, ML programs are even faster. Our ultimate goal is to continue improving the efficiency of expressive languages until there is no compelling technical reason *not* to use an expressive programming language.

The key idea for improving the efficiency of a program written in an expressive language is to translate it into another program that has the same meaning, but in which many of the sources of inefficiency have been removed. Because many inefficiencies are due to abstractions, one goal of the translation is to eliminate as many abstractions as possible. After all, the main purpose of the abstractions is to help the human programmer think about the program; they are (usually) not necessary in the execution phase of the program.

Tree-Based Programming

An effective way to modularize programs is to decompose them into components that produce and consume intermediate data structures. As a concrete example of this, consider the following ML functions:

```
down 0 = []
down n = n :: (down (n - 1))

prod [] = 1
prod (x::xs) = x * (prod xs)

fact n = prod (down n)
```

The `down` function creates a list of numbers from its argument down to 1. For instance, `down 5` denotes the list `[5,4,3,2,1]`. The `prod` function returns the product of all numbers in its argument list. For instance, `prod [7,2,3]` denotes the number 42. The `fact` function, when called on a number n, returns the product of the list of numbers from n down to 1; that is, it computes the factorial of n.

Note how the `fact` function is created as the composition of two other functions that can be useful in many other contexts as well. This is a hallmark of expressive programming. But, like many expressive programs, this definition of `fact` is less efficient than it could be. In this case, the inefficiency is due to the construction and traversal of the intermediate list `[`n`,`$n-1$`,...,3,2,1]`.

It is possible to get rid of this intermediate list by fusing the composition of `prod` and `down` into a single function, as follows:

```
fact 0 = 1
fact n = n * (fact (n - 1))
```

The transformed version of `fact` takes less time and space than the original because the intermediate list, whose only purpose was to communicate the results of the `down` function to the `prod` function, has been eliminated. However, in the more efficient version, it is less obvious that `fact(`n`)` calculates the product of the numbers between 1 and n. We shall refer to the original definition of `fact` as *modular* and the the transformed version as *monolithic*.

In the above example, linked lists are used to communicate information between components. They can be considered *virtual data structures* in the sense that they are not computationally necessary but are introduced solely to allow expressing the program in a modular way. More generally, such virtual structures can have a tree-shaped branching structure. I shall use the term *tree-based programming* for the style of programming in which programs are modularized by decomposing them into components that communicate via trees.

I have undertaken several research projects that improve the expressiveness of and reduce the inefficiencies of tree-based programming. Below is a brief description of these projects. See my *Research Summary* for more details.

- *Synchronized Lazy Aggregates*: Modular programs can sometimes require space resources that have higher complexity bounds than the corresponding monolithic programs. For my MIT doctoral research, I developed a tree-based programming system in which monolithic programs could be decomposed into modular programs with the same asymptotic space complexity. The key technical advance of this work was the invention of the *synchron*, a first-class barrier synchronization object. I used synchrons to create special trees, known as *synchronized lazy aggregates*, that allowed computations to be modularized without increasing spaces bounds.

- *Regular Tree Manipulation*: Cyclic data structures – structures that contain pointers back to some part of themselves – are ubiquitous in computer science, but there are not very good abstractions for manipulating them efficiently. In our *Cycle Therapy* work, Joe Wells and I developed elegant abstractions for creating and manipulating cyclic structures, which can be viewed as denoting infinite regular trees.

- *Deforestation*: It is possible to automate the process of eliminating virtual trees from programs – a process playfully known as *deforestation*. Several automatic deforestation techniques have been proposed, but few have been implemented in real programming languages, and even those that have been implemented have not been carefully tested and evaluated for how well they perform in practice. In the summer of 2000, I began working with Patty Johann (then at Bates College, now at Dickinson College) on a project to evaluate, compare, and improve several state-of-the-art deforestation techniques. The project is still ongoing, but we have made important progress, much of it with the help of seven undergraduates who have been involved in the project (see Section 2.2). A description of our summer research project with six of these students can be found in our paper, *Lumberjack Summer Camp: A Cross-Institutional Summer Research Experience In Computer Science*.

Compiling with Flow Types

Deforestation is an example of a so-called *source-to-source program transformation*, in which the output of the efficiency-improving translation is a program written in the same language as the input program. Many of the inefficiencies of expressive languages cannot easily be removed by source-to-source transformations, but can be removed via translations into so-called *intermediate languages*, which are lower-level programming languages used in the compilation process.

During the past decade, there has been a flurry of research on *typed intermediate languages*, intermediate languages in which each expression is annotated with information about the type of value calculated by that expression. Guided by this type information, a compiler for an expressive language can sometimes choose representations for data that are as efficient as those chosen by a C compiler. Without the type information, the compiler must usually choose a more abstract – and therefore less efficient – representation.

Most of the research I have done while at Wellesley has focused on improving the selection of efficient data representations by using a typed intermediate language based on a novel sophisticated type system. I have carried out this research in collaboration with members of the Church Project, a Boston-based research group that I co-founded in 1995. Our work has focused on *flow types*, in which the type system is embellished with fine-grained information about which values flow where in a program. The flow information encoded in the types enables the compiler to make more efficient data representation choices than with an unembellished type system.

The Church Project spans both theory and practice. In addition to developing the theory of flow types, we have implemented a compiler for the ML language based on flow types and have begun evaluating their practical impact.

I have made fundamental contributions to both the theoretical and practical facets of the Church Project. In terms of theory:

- I helped to show that a broad class of polyvariant flow analyses have the same information content as a type system with intersection and union types;
- I helped to design and prove formal properties about λ-CIL, a calculus with flow types;
- I showed a theoretical upper bound for the complexity of inferring finite-rank intersection types; and

- I helped to develop a new technique for proving the computational soundness of a calculus, and applied this technique to a module calculus.

In terms of practice, I was the chief architect and one of the four main implementers of an ML compiler that uses CIL, a flow-typed intermediate language based on the λ-CIL calculus, to select efficient representations of functions. I have also been active in our recent preliminary experimental work to evaluate the benefits and costs of flow types. My contributions to the Church Project, including descriptions of the papers that I have co-authored, are detailed in my *Research Summary*.

2.2 Student Research

I have advised numerous Wellesley students in research projects and independent study projects. Although the independent study projects have not always involved "true research" in the sense of discovering new results, they have been "research-like" in their engagement of students in the self-directed inquiry of problems that are more challenging and (at least initially) less well-specified than those encountered on class assignments.

Below is a brief description of the student projects that I have advised. Unless otherwise stated, the projects during the regular semester were CS350 independent study projects.

- Ramona Filipi '96 worked on a senior project, *Cooperative Query Answering on Musical Data*, in which she developed a system for querying a musical database for songs containing certain patterns of notes. (I co-advised Ramona with Matthew Merzbacher.)

- In her senior project *LEGO Robot Projects: An Innovative Way of Learning Science and Technology*, Ruth Chuang '96 developed materials for *Robotic Design Studio* and used similar materials to engage grade school students in science projects. (I co-advised this project with Robbie Berg.)

- In Spring '97, Crystal Ellsworth '96 implemented a simple database system in Scheme.

- In Summer '96, Laura Diao '98 undertook a summer research project, *Visual Robot Programming*, in which she developed some graphical programming tools in Java and designed a visual rule-based language for programming robots. Laura presented posters of her work at the Science Center summer poster session and at the CCSCNE-97 conference at Northeastern. After working as a consultant for several years, Laura has returned to Wellesley as a member of Information Services.

- For her Honors Thesis, entitled *Visual Graph Abstraction*, Anna Mitelman '97 built on Laura Diao's work by designing and implementing a Java program for viewing a graph of nodes and edges in terms of *supernodes* (each of which is a collection of nodes) connected by *superedges* (each of which is a collection of edges). Anna presented posters of her work at the Science Center summer poster session and at the CCSCNE-97 conference at Northeastern.

- In Spring '98, Cynthia Jones '97 experimented with a Scheme database system.

- In Spring '97, Elena Konstantinova '98 implemented a term rewriting system.

- In Summer '97, Yan Zhang '99 built a system for animating the sequence of trees produced by Elena Konstantinova's term rewriting system. I still use this system to illustrate the "shapes" of computational processes in CS111. Yan presented a poster of her work at the Science Center summer poster session.

- In Spring '99, Ann Hintzman '99 finished building a robot (started in Wintersession '99) that extinguished candles by positioning a balloon over the flame. She participated in the Trinity College Fire-Fighting Robot competition, and summarized her experiences in a report, *Popeye: The Design and Implementation of a Fire-Seeking Robot.*

- For her senior Honors Thesis, Yan Zhang '99 implemented and experimented with the "short-cut fusion" technique in a mini-language written in Scheme. She presented her work, *Experiments with Shortcut Deforestation*, at a Science Center spring poster session. Yan continues to explore her interest in programming languages as a graduate student in MIT's Laboratory for Computer Science.

- In Spring '00, Emily Horton '00 debugged and extended *LogoBlocks*, a visual language for robot programming created at the MIT Media Laboratory. She presented her work in a poster at a Science Center spring poster session. Emily was hired as a summer intern at the MIT Media Laboratory to continue her work.

- In Spring '00, Lisa Hazel '01 and Meredith Shotwell '01 finished building a robot (started in Wintersession '00) that extinguished candles with shaving cream. They presented their work in the Science Center spring poster session and on a web site. (I co-advised this project with Robbie Berg.)

- In Spring '00, for a CS250H project, Erika Symmonds '02, implemented the candle-finding navigation system for a robot she built in Wintersession '00 with her partner, Brian Kelly (a Wellesley Middle School teacher). Erika participated in the Trinity College Fire-Fighting Robot competition, and summarized her experiences in a report, *Bandebot: A Fire-Fighting Robot.* (I co-advised this project with Robbie Berg.)

- In Summer '00, four Wellesley students joined two Bates students in *Lumberjack Summer Camp*, a summer research project on deforestation supervised by Patty Johann (Bates College) and me. Here are brief descriptions of the four Wellesley student projects:

  - Kirsten Chevalier '01 explored the importance of inlining techniques in deforestation and experimented with Olaf Chitil's prototype implementation of his type-based deforestation technique.
  - Kate Golder '02 worked on a demodulizer for the Haskell programming language – a module combination tool that is necessary to prepare standard benchmarks for processing by many deforestation engines.
  - Holly Muenchow '02 investigated warm fusion, a deforestation technique, and experimented with László Németh's implementation of this technique.
  - Nausheen Eusuf '02 implemented tree-based programs in Haskell that will serve as good benchmarks for deforestation.

  All students wrote reports of their work and presented posters at the Science Center summer poster session. Kirsten and Kate also presented posters of their work at the CCSCNE-01 conference.

- For her Honors Thesis, Kirsten Chevalier '01 extended the prototype of Chitil's type-based deforestation engine to work on simple Haskell programs. Kirsten presented this work, *Exploring the Type Inference Approach to Deforestation*, at the Science Center spring poster session, the Ruhlman Conference, and the Mid-Atlantic Student Workshop on Programming Languages and Systems (MASPLAS-01). Kirsten continues to work on the deforestation project as a computer science graduate student at Berkeley.

- In Spring '01, Yukari Wada compared implementations of data structures in Java and C++, and summarized her experience in a web-based report.
- In Spring '01, Laura Hwang implemented solutions to the 2000 Boston area ACM programming contest problems in Java and C, and developed a set of web pages presenting her solutions.
- In Spring '01, Larissa Ranbom '02, Kirsten Chevalier '01, Emily Braunstein '01, and several students from other departments banded together to construct *Versailles*, an impressive interactive light sculpture made out of hundreds of discarded CDs that graced the mini-focus of the Science Center for several weeks.
- Building on her Summer '00 work, Nausheen Eusuf '02 is currently working on an Honor Thesis project in which she is investigating the expression of algorithms from automata theory in Haskell, a lazy functional programming language.
- Julie Weber '03 is currently continuing work on the Haskell demodulizer begun by Kate Golder '02.

2.3 Future Research Plans

In each of the programming language research projects that I have undertaken, numerous avenues of future research have suggested themselves. Additionally, there are a number of new research areas that I would like to explore. In the subsections below, I discuss a number of possible future research projects that I am considering.

The Church Project

Having invested thousands of hours in the construction of a working flow-typed compiler, my Church Project colleagues and I are planning a number of extensions and experiments that should require relatively little additional work:

- Thus far we have focused on a relatively small collection of representations for functions. We plan to implement and evaluate a broader range of function representations and to explore representation customizations for other data structures.
- There is much debate in our community as to whether flow-based inlining of functions is a good idea or whether more traditional syntax-based inlining techniques suffice. We believe that our compiler is an excellent workbench for investigating this question, and plan to undertake such a project in the near future.
- We plan to generalize the *known function optimization* used within our compiler to a flow-based *known value optimization* that works on any type of value. As with function inlining, we plan to compare flow-based and syntax-based algorithms to see whether the flow-based algorithms are sufficiently beneficial to justify the extra work of maintaining flow information.

There are also a number of longer-range projects that will require significantly more work:

- My work with Amtoft on the correspondence between polyvariant flow analysis and polymorphic type systems suggests that the Church compiler can be extended to encode a much broader range of flow analyses than it currently does. However, there is a key technical hurdle (the so-called deep vs. shallow subtype distinction) that separates the work I did with Amtoft and its practical application to the Church compiler. I plan to investigate a formal characterization of the relationship between deep and shallow subtyping, and to explore if such a characterization can be used in practice as the basis for applying our theoretical results on flow/type correspondences to the Church compiler.
- Our current compiler does not implement many standard optimizations, which makes it difficult to compare the code generate by our compiler to the code generated by compilers implementing these optimizations. We plan to address this in two ways. First, if we modify our compiler to produce C code as output rather than assembly code, we can leverage the back-end optimizations performed by good C compilers. Second, we plan to implement a number of important optimizations on our intermediate code. This is non-trivial, since the non-standard structure of our intermediate language makes it impossible to straightforwardly implement standard optimizations in our framework.
- One of our goals is to develop a flow-typed assembly language and investigate opportunities for flow-based optimizations at this level. However, this goal conflicts with the idea of generating C code rather than assembly code. We may have to implement *two* back-ends for our compiler to resolve this conflict.
- Our current compiler, as with many other research compilers in our community, is based on a the assumption that the whole program is being compiled at once. But in practice, many programs are constructed by combining several modules that are separately compiled. It is imperative to investigate whether flow types can be used as the basis for a system that supports separate compilation rather than just whole program compilation. One idea is to perform optimizations not only when modules are compiled, but also when they are linked together. My module calculus work with Machkasova is a first step in the study of modular program analyses. The fact that finite-rank intersection types (which I studied with Kfoury, Mairson, and Wells) have a so-called *principal typing* property, and therefore provide a modular analysis of a program fragment, makes them a promising candidate for the basis of a module type system that supports link-time optimization—an avenue I plan to explore.

Tree-Based Programming

I plan to continue my research in the area of tree-based programming via several projects:

- My highest priority in this area is to complete the study I have started with Johann and our students on the empirical evaluation of state-of-the art deforestation techniques. This will require finishing the implementation of Chitil's type-based deforestation technique, and developing a methodology for testing it and comparing it with two existing deforestation engines. This study will be valuable for reporting to the community which deforestation techniques work well in practice.
- Inspired by my work with flow analyses in the context of the Church Project, I plan to investigate flow-based approaches to deforestation – an area which, to my knowledge, has not yet been explored by anyone. It is my intuition that flow-based methods may be able to solve some known problems with current techniques.

- I am continuing work with Wells on cyclic representations of infinite regular trees and the efficient manipulation of such representations. Our goal is to develop a libraries of operations for manipulating regular trees that are efficiently implementable in a wide range of programming languages. Our *Cycle Therapy* work is a first step in this direction, but significant practical engineering work remains to be done.

- In teaching data structures, algorithms, and programming languages, I have found that much material can be simplified if it is presented in terms of rules for rewriting trees and graphs. Building on some previous work on the animation of tree rewriting done in conjunction with my students (see Section 2.2), I plan to develop tools for expressing algorithms via rewriting rules, animating the execution of such rules, and translating the rules into algorithms in a wide range of programming languages. Such tools would be useful in both teaching and research.

Robotics

In the process of teaching *Robotic Design Studio*, I have encountered several research problems that I would like to explore. For example, students are often frustrated by the difficulty of debugging their robots. The sorts of debugging and tracing tools with which some students are familiar from other programming languages are not available in the systems used to program the HandyBoard and Cricket robot controllers used in our class. How to specify and collect trace information on these controllers is far from obvious, as is how to transmit the collected information from the controller to a host computer, where it can be viewed and manipulated. Some of the bugs encountered by students involve resource limitations of the controllers; these could potentially be discovered by the compiler rather than waiting for the robot to misbehave when it runs. I would like to develop better compilers and debugging facilities for these kinds of robots, so that programming them is less frustrating.

Security

Computer security is an area in which I have become increasingly interested. I am not a computer security expert, but I would like to learn more about it by undertaking research projects in the area. In particular, I am interested in working on systems in which programs are guaranteed to satisfy certain security properties – e.g., they do not access private information, surreptitiously send email (a means by which viruses proliferate), or exceed certain resource bounds when executing. A number of systems with this flavor have been developed in the past decade, but research in this area is still embryonic and much remains to be done. I am also interesting in learning more about computer viruses in an effort to develop better ways to detect and eliminate them.

Document Preparation

The document you are reading is prepare using Latex, a powerful document preparation program that is used by large numbers of computer scientists, mathematicians, and others to write documents with sophisticated formatting. In contrast to more popular document preparation systems (e.g., Microsoft Word), Latex allows the document writer to define arbitrary formatting abstractions that greatly facilitate writing and modifying documents. Unfortunately, the language for expressing these abstractions is difficult to use – indeed, it is almost universally agreed to be one of the worst programming languages ever designed. Nevertheless, people use it anyway (cursing all the way) because it gives such beautiful output. I would like to improve this situation by helping to develop an elegant programming language for document formatting in which the specifications of the formatting abstractions themselves are as beautiful as the formatted output they describe.

2.4 Evaluating Computer Science Research

It is my understanding that the number of archival journal papers authored by a tenure candidate is often a major factor in tenure decisions in the sciences. Given this state of affairs, I feel it is necessary to discuss my publication record, which is weighted heavily toward conference and workshop papers, in the context of accepted standards of evaluation in the field of computer science.

In computer science, especially in the area of programming languages, the preferred means of publication for one's current research is one of a number of selective annual conferences or workshops. Not only are these publications carefully reviewed (typically by three to five referees), but the competition for acceptance is often fierce; acceptance rates are typically in the 25%-40% range, but in some conferences are even smaller. The fact that a paper appears in the proceedings of a selective conference attests to its novelty, relevance, and quality. Indeed, it is folklore in our community that it is *more* difficult to publish a paper in a good conference than in an archival journal. Other advantages of conferences are that (1) the turnover time to publication is typically far shorter than that for a journal and (2) conference papers are presented in a public forum whereas journal papers are not.

Journal publications still have a place in computer science. Since it is often impossible to give a full account of one's research within the small page limits of a conference proceedings, authors will often publish an expanded (and more polished) version of a conference paper as a journal paper or a technical report. However, the pressure to publish new work in conferences often means that expanded versions of older papers often do not appear for many years, if at all.

As evidence for the above claims, I call attention to the Computing Research Association's "Best Practices Memo" entitled *Evaluating Computer Scientists and Engineers for Promotion and Tenure*, which appeared in the September 1999 issue of *Computing Research News*. Below is a relevant quote from the introduction of the memo. (The full memo appears in Appendix G of this document.)

> The evaluation of computer science and engineering faculty for promotion and tenure has generally followed the dictate of "publish or perish", where "publish" has had its standard academic meaning of "publish in archival journals" ... Relying on journal publications as the sole demonstration of scholarly achievement, especially counting such publications to determine whether they exceed a prescribed threshold, ignores significant evidence of accomplishment in computer science and engineering. For example, conference publication is preferred in the field, and computational artifacts – software, chips, etc. – are a tangible means of conveying ideas and insight. Obligating faculty to be evaluated by this traditional standard handicaps their careers, and indirectly harms the field.

It is worth emphasizing that the authors who prepared this memo (David Patterson, Lawrence Snyder, and Jeffrey Ullman), are leaders in the field with outstanding publication records. For example, in the automatically generated author ranking of the NEC ResearchIndex engine[3], Jeffrey Ullman is ranked as the most cited computer scientist of all time, and David Patterson is ranked as the 29th most cited.

[3] `http://citeseer.nj.nec.com/mostcited.html`.

Further confirmation of the importance of conference publications relative to journal publications in computer science comes from the estimated "impact rating" reported by the NEC ResearchIndex engine[4]. The engine automatically ranks conferences, workshops, and journals according to the average number of citations per article[5]. In the top 25 publication venues ranked by impact, only 4 are journals (the rest are conferences and workshops), and only 27 of the top 100 items are journals.

In my Activities Sheets, I have listed the acceptance rates (where known) for all papers that I have published in refereed conferences and workshops. In all cases except one, the acceptance rates are in the 28%-37% range, indicating that these conferences and workshops are highly selective. The one exception is the "Cycle Therapy" paper that appears in this year's *Third International Conference on Principles and Practice of Declarative Programming (PPDP)*, where the acceptance rate was 48% (19 papers accepted out of 40 submitted). For reasons that are not well understood, the number of papers submitted to PPDP this year (40) was significantly smaller than the number submitted last year (65). It is also worth reporting the ResearchIndex impact rating of the conferences and journals in which my papers appear[6]:

Venue	**ResearchIndex Impact Rating**
Journal of Functional Programming (JFP)	top 9.20%
International Conference on Functional Programming (ICFP)[7]	top 10.02%
Conference on Human Factors in Computing Systems (CHI)	top 11.88%
European Symposium on Programming (ESOP)	top 13.40%
Communications of the ACM (CACM)	top 27.38%
Principles and Practice of Declarative Programming (PPDP)[8]	top 66.89%

I would also like to comment on the fact that all my publications (except for one) were collaborative ventures with one or more co-authors. As computer science has matured as a field, single-author papers have become more rare. Especially in experimental subdisciplines of computer science, it is nearly impossible for a single person to design, build, and evaluate complex software and hardware systems. This has certainly been my experience in the construction of the Church Project compiler, which required the investment of thousands of man-hours from the four main implementors (Allyn Dimock, Ian Westmacott, Bob Muller, and me) and significant work from other individuals. A similar story is unfolding in my on-going work in the empirical evaluation of deforestation techniques with Patty Johann and our students. Even in purely theoretical work, there are many advantages to the collaboration of authors with different backgrounds and different strengths. Such collaboration allows investigation of more complex topics than would otherwise be possible and that the work is often of higher quality because it has to pass the "criticism filters" of multiple authors.

[4]`http://citeseer.nj.nec.com/impact.html`.

[5]Citations from articles with an author in common with the cited article are excluded from the count.

[6]Several publication venues for my papers are not ranked by ResearchIndex and so do not appear in the list.

[7]Although ICFP has a good impact rating, it is worth noting that it is a relatively new conference, formed in 1996, that resulted from merging two older conferences with higher impact ratings: Lisp and Functional Programming (top 1.74%) and Functional Programming and Computer Architecture (top 2.56%).

[8]The poor impact rating of PPDP is at least partially explained by the fact that it is a new conference, formed in 1999, as the result of merging two older conferences with higher impact ratings: Programming Language Implementation and Logic Programming (top 26.45%) and Algebraic and Logic Programming (top 39.16%).

3 Service

In this section, I expand on some of the service items listed in my *Activities Sheets*.

3.1 College Service

I have served on two college committees:

1. For two academic years (Fall '96 – Spring '98), I was was the Group C representative on the Committee on Minority Recruitment, Hiring, and Retention (CMRHR). One of the main goals of CMRHR during these years was to to determine whether data showed any salient differences between minority and non-minority faculty members in terms of the trajectories of their careers at Wellesley. For instance, were there noticeable differences in hiring and retention between minority and non-minority faculty members along dimensions like part-time vs. full-time, tenure-track vs. non-tenure-track, and, for faculty members who had left the college, *when* they left the college in their career path? We also wanted to explore to what extent factors other than ethnicity – such as place of birth and citizenship (when hired) – were correlated with points on these dimensions.

   Since the information we wanted was not readily available, we decided to request that each department provide us with this information for every faculty member hired in the department since 1985. Asking for detailed information about so many individuals would be burdensome for departments (especially large ones), so it was important to streamline the information gathering process. I took the lead on designing and implementing the questionnaires given to departments. I wrote a program that produced a personalized one-page questionnaire for every faculty member being studied and automatically filled in as much information as possible given information already in Banner. (See Appendix H for a sample questionnaire.) This way, departments could focus on filling in the missing information rather than spending time on what was already known. The feedback from pilot tests with a few departments was positive, so we distributed the questionnaires to all departments. I continued to help out with the questionnaire and tabulating the results from it even after officially leaving the committee for my junior leave (Fall '98 – Spring '99).

2. I am entering my third academic year (starting in Fall '99) as the Group C representative of the Committee on Educational Research and Development (ERD). In my first term (Fall '99 – Spring '00) I, working with Wilbur Rich, was responsible for processing the Curriculum Development Grants. In my second term, (Fall '00 – Spring '01) I was responsible for reviewing the travel-related Pedagogy Improvement Grants. Since Fall '99, I have also served as the ERD liaison to the Hughes Curriculum Development Award Committee at the Science Center.

In addition to committee service, I have been active as a first-year advisor. I served as a first-year advisor for the Cazenove dormitory team in the academic year starting 1995, and advised first-year students in the 1997 and 1999 academic years. It is gratifying to watch first-year students grow and mature during their years at Wellesley.

I have also been a tenure-track interviewee for the Mellon Interviews on the Faculty Life Cycle, a panelist for an LTC panel on deadlines and extensions, and a "reflexive photographer" for the planning of the new campus center.

3.2 Departmental Service

In addition to course development (especially leading the Java charge) and student mentorship , I have contributed to the computer science department in other significant ways:

- Since 1995, I have been the coach for the Wellesley programming team, which participates in the annual ACM programming contest. I formed a Programming and Problem Solving Club that served as an informal venue for practicing problem solving skills and programming contest questions. This club eventually evolved into the student-run Wellesley Association for Computing, which sponsors other computer science activities as well (e.g. attending CS seminars).
- Every year, I spend significant time helping to administer the department's Unix/Linux machines, for which there is no dedicated system administrator.
- From Summer '00 – January '01, I volunteered to supervise the initial stages of switching program development environments for CS111, even though I was not officially teaching the course.
- I have taught unofficial "lunchtime courses" on Java and C programming.
- I have organized social events like student dinner parties and a department hike.
- As the department's "information czar", I post CS-related news articles, distribute information on CS internships and jobs, etc. I am currently in charge of revamping the department's web pages. In addition to updating the pages and reorganizing the information to be more accessible, I plan to add pages on student and faculty research, information on finding internships and jobs, advice columns from CS alums, and pages marketing the department to high school students.

3.3 Professional Service

In addition to standard service activities, such as reviewing papers for conferences and journals, I have also helped to organize some conferences. Most recently, I served as a program committee member for the Types in Compilation Workshop held in Montreal in September, 2000. In this capacity, I carefully read and wrote detailed referee reports for ten of the submitted papers, and helped to choose the papers accepted at the workshop. I also served as the vendors chair for the Consortium for Computing in Small Colleges Second Annual Northeastern Conference held at Northeastern University in April, 1997, and the local coordinator for the Forum on Parallel Computing Curricula held at Wellesley in spring 1995.

A CS111: Introduction to Programming and Problem Solving

CS111 is the first course in the computer science major. It introduces students to fundamental programming and problem solving techniques. Soon after I came to Wellesley in Spring '95, the department began to discuss revamping CS111 and the follow-on course, CS230 (Data Structures), including changing the programming language (Pascal) used in these courses. The sense was that the limitations and quirks of Pascal made it less than ideal for teaching the subject matter of CS111 and CS230.

I volunteered to lead the effort to switch to Java and revamp the introductory programming curriculum. I redesigned CS111 to use Java for Fall '97, which necessitated rebuilding the course from scratch. Inspired by my belief in constructionism (see Section 1.1), I developed a collection of Java *microworlds* – simple but rich environments in which students can "play with" fundamental programming techniques.

One example of such a microworld is *BuggleWorld*. In this world (inspired by Seymour Papert's turtle microworld and Richard Pattis's Karel the Robot), simple robot-like creatures called "buggles" can be programmed to explore a two-dimensional grid of cells. Buggles can turn, move forward and backward, paint their current cell with a color, pick up and drop bagels (their favorite food), and sense when they are facing a wall or over a bagel. These simple actions are enough to support a surprisingly wide range of activities. Buggles can be programmed to draw letters, jump hurdles, find and eat bagels, explore mazes, and make complex rug patterns with colors and bagels. Through these playful exercises, students learn fundamental concepts like method-based abstraction, conditionals, recursion, iteration, and state variables. The graphical nature of the environment makes it visually compelling and provides important debugging clues when programs are not working correctly.

BuggleWorld is only one of many microworlds I developed for CS111. Other microworlds allow the creation of Escher-like pictures, fractal line drawings, screen savers, and playing card games. These microworlds enable posing simple, motivating, and challenging programming problems that would be significantly harder to specify without the scaffolding provided by the microworlds. In addition to implementing the above microworlds in Java, I (in conjunction with Stanzi Royden, Elaine Yang, LeeAnn Tzeng, Jennifer Stephan, and Jean Herbst) developed a wealth of examples, assignments, lectures, and notes based on these microworlds. Visit the following URL for examples of these:

http://cs.wellesley.edu/~fturbak/tenure/cs111.html

In order to understand how a Java program runs, a programmer must have a model of Java execution. I invented a visual model called the Java Execution Model (JEM) that accurately explains at an abstract level what happens when a Java program is executed. In CS111, I teach the JEM as a means of understanding the dynamic nature of computation and as a tool for debugging programs that do not work. It is worth noting that of the dozens of introductory and advanced Java textbooks I have consulted, not a single one presents a general execution model of Java.

In addition to developing Java microworlds and associated materials, I also changed the content and structure of the course to emphasize "big ideas" and "first principles". This led to some non-standard topics and topic ordering. For example, an important theme of CS111 is the *divide/conquer/glue* problem solving methodology, which is presented early in the course. It turns out that the key concept of *recursion* is a special case of divide/conquer/glue in which each of the subproblems has the same structure as the original problem. Furthermore, the key concept of *iteration* can be viewed as a special case of recursion where (1) there is exactly one subproblem and (2) there is no “glue” step. Based on these observations, I explicitly teach recursion as a special case of divide/conquer/glue and then teach iteration as a special case of recursion. (See my paper *Teaching Recursion Before Iteration in CS1* for more details.) Similar reasoning compels me to teach lists before arrays. Although these choices are sometimes made in the teaching of so-called functional programming languages, they are highly unusual in the teaching of object-oriented languages such as Java. Nevertheless, I believe that organizing CS111 in this way gives the students a much better view of the “big picture” than traditional approaches for teaching Java.

B CS115: Robotic Design Studio

Ever since the year I arrived at Wellesley, I have been collaborating with Robbie Berg in the Physics department on *Robotic Design Studio*, an introductory robotics course held during Wintersession. The course arose out of our passion for constructionism and our observation that activities involving designing and building are relatively rare on the Wellesley campus, especially in the sciences. Our goal was to create a course where all students, regardless of background, could get hands-on experience building complex artifacts of their own design.

Robotics is an ideal domain for engaging students in such activities. With the advent of palm-sized robot-controlling computers developed at the MIT Media Lab, small independent robots with interesting behaviors can be built from LEGO parts, motors, and simple sensors, making robotics accessible to people of all ages and backgrounds. The interdisciplinary nature of robots – combining mechanical know-how, programming skills, simple electronics, a sense of aesthetics, and, in some cases, understanding of animal-like behavior – makes them appealing to students with a wide range of backgrounds, especially those seeking the sort of interdisciplinary experiences that are hallmarks of liberal arts education. The fact that almost any sort of dynamic behavior can be realized in robots means that robotics serves as a kind of blank canvas encouraging creative design experiences and personally meaningful projects.

During Wintersession '96 and '97, we held non-credit pilot courses in which we experimented with various kinds of robotics technology and pedagogy. These pilot courses were supported by grants from the National Science Foundation, the Howard Hughes Medical Institute, and Wellesley's Educational Research and Development Committee. The most important aspect of these courses was that they culminated in an open-ended final group project that students exhibited in a public forum. This exhibition format – effectively a kind of robotics "talent show" – stands in stark contrast to the competitive events that characterize most robotics courses elsewhere.

Inspired by the success of the pilot course, in Wintersession '98 we started offering the robotics course for one half-unit of credit, and have taught the for-credit course every Wintersession since. The course is extremely popular – it is overbooked every Wintersession, and the final exhibition draws crowds of about 200 people from the Wellesley community. Every year, we are amazed by the ingenuity and creativity of the student projects. Other than attending the exhibitions in person, the best way to get a sense for these projects is to look at the on-line museum of past student projects at `http://cs.wellesley.edu/rds/museum.html`.

Together, Robbie and I have developed a wide range of materials for this class, from sample robots to handouts. We introduce students to the basics of robotics via a series of design challenges in which they either modify an existing model robot (known as "SciBorg") or build one of their own. Visit the following URL for examples of materials from this course:

`http://cs.wellesley.edu/~fturbak/tenure/cs115.html`

After six years of teaching the robotics course, we have come to the conclusion that the essence of the course is introducing students to the big ideas of engineering. Although teaching engineering in a liberal arts college is often considered heresy, we believe that every liberal arts student should be exposed to the key ideas of engineering. For a detailed exposition of our argument, see our journal paper, *Robotic Design Studio: Exploring the Big Idea of Engineering in a Liberal Arts Environment*. (This paper has recently been accepted for publication in the *Journal of Science Education and Technology*.)

C CS230: Data Structures

Although I have not taught this course since Fall '99, it is the regular semester course I have taught most frequently at Wellesley (six times). In the first four incarnations of this course, I developed an extensive set of notes, assignments, and programs that used Pascal to illustrate basic data structures and algorithms. Among the programs I developed for examples and assignments were cellular automata, maze drawing and solving programs, fractal drawing programs, a simple version of Weizenbaum's celebrated Eliza program (a software psychiatrist), and a simple hypertext system. These programs, many of which had a graphical or interactive component, provided compelling examples of how the ideas in the course could be used in practice. For example, on one assignment involving the hypertext system, students used graph traversal techniques to implement a very simple version of a web-crawling text-indexing search engine like Google.

A novel aspect of my teaching in this course was an emphasis on passing functions as arguments to other functions. Because of my background in functional programming, I recognized the importance of this often-neglected feature of Pascal, which is essential for abstracting over common data structure traversal patterns.

In Spring '98, I overhauled CS230 to complete the move from Pascal to Java in our introductory curriculum. This required major changes to the material covered in the course. As part of redesigning CS111, I moved much material that was previously a focus of CS230 (such as recursion, linked lists, and many microworlds illustrating these concepts) into CS111. It was necessary to move some topics formerly emphasized in CS111 (such as an in-depth coverage of arrays) to CS230, and to cover significant new material specific to Java (e.g., details of object-oriented programming and graphical user interfaces). I adapted many programs from the Pascal version of the course to Java, and wrote several new ones as well (such as a bitmap editor and a Mancala game).

Samples of materials from both the Pascal and Java versions of the course can be found by visiting the following URL:

`http://cs.wellesley.edu/~fturbak/tenure/cs230.html`

D CS231: Fundamental Algorithms

This is one of the few courses in which I rely on a textbook (*Introduction to Algorithms*, by Cormen, Leiserson, and Rivest) for good explanations of most material and for many of its excellent problems. Nevertheless, students in the course often do not have (or at least think they do not have) the level of mathematical sophistication assumed in many chapters of the book. My approach in the course is two-pronged: (1) to raise their level of mathematical sophistication, so that they can better understand the textbook reading and do the textbook problems; and (2) to present intuitive explanations of results that are presented formally in the book.

Towards this end, I have prepared numerous "worksheets" that highlight key concepts, present intuitive explanations, and give many concrete examples. The worksheets contain numerous blank spots where students are expected to fill in answers, draw pictures, derive formulae, etc. In many of my lectures, we walk through the material on a worksheet and interactively flesh out the parts that are blank. I find that these worksheets strike a nice balance between not giving out any notes (in which case students must take copious notes, making it difficult to focus on the high-level points of the lecture) and giving the students completely fleshed out lecture notes (in which case they may not be as actively engaged with the material).

Inspired by my research in the area of functional programming languages, I have recently been experimenting with presenting functional versions of algorithms rather than the traditional imperative versions. In many cases, the functional versions are more elegant and easier to understand and analyze than their imperative counterparts, though they may not be as efficient in practice. For example, in Spring '01 I presented a functional approach to red-black trees based on the version in Chris Okasaki's book, *Purely Functional Data Structures*; this was much more understandable than the complex description in the textbook. This semester, I am covering the sorting of lists in the functional style in addition to array sorting because it is easier to teach induction proofs in the context of lists rather than arrays.

Visit the following URL for examples of materials from this course:

`http://cs.wellesley.edu/~fturbak/tenure/cs231.html`

E CS251: Theory of Programming Languages

Typically, courses on programming languages are either organized around the historical development of programming languages or around different programming paradigms, such as imperative, functional, object-oriented, and logic programming. Students in such courses often get a feel for different paradigms by writing simple programs in a representative language from each paradigm.

In contrast, I organize my programming languages course around dimensions of programming language design and the choices that are available in each dimension. For example, some important questions in various dimensions are:

- Are functions in the language first-order or higher-order?
- Is the language block structured?
- What is the parameter-passing mechanism of functions?
- Are free variables statically scoped or dynamically scoped?
- Are values untyped, dynamically typed, or statically typed?
- Does the language support parametric and/or ad hoc polymorphism?
- Is type equality structural or by name?
- Is there no inheritance, single inheritance, or multiple inheritance?

The best way I know to get a sense for what the dimensions are and what the choices are in each dimension is to study interpreters. An interpreter is a program written in an *implementation language* that executes programs written in a *source language*. Choices along a source language dimension often clearly show up as a programming choice in an interpreter for that language. For instance, in the implementation of variable scoping, there are two so-called "environments" that are available for determining the meaning of a variable name; one corresponds to static scoping while the other corresponds to dynamic scoping. The person implementing the interpreter has to make a choice between these two environments, and is thus forced to step into the realm of programming language design.

A practical problem is that interpreters for real programming languages are too complex to study in an undergraduate course. To circumvent this problem, I have designed a hierarchy of "mini-languages" that are very simple, but capture the essential features of real programming languages:

- INTEX is an simple language of integer expressions.
- BINDEX = INTEX + a name-binding construct.
- IBEX = BINDEX + conditionals and lists.
- FOFL = IBEX + first-order functions.
- FOBS = FOFL + block structure.
- HOFL = FOBS + higher-order functions.
- HOIL = HOFL + imperative features.

We spend most of the course studying interpreters that I developed for each of these languages. Students get experience in changing the meaning of a mini-language feature or adding a new mini-language feature by modifying the interpreter for the mini-language. Visit the following URL for sample materials from the course:

`http://cs.wellesley.edu/~fturbak/tenure/cs251.html`

The fact that each mini-language and interpreter builds upon a previous one allows for an "onion-skin" approach in which students get hands-on experience with one level of complexity before moving on to the next. Along the way, we relate what we have learned to real programming languages, such as Scheme, ML, Haskell, Java, C, and C++. The interpreters we study are written in Scheme and ML, so students get significant programming experience in these real languages.

F CS301: Compiler Design

Because my research work involves the implementation of an experimental compiler, CS301 is a course close to my heart. It is a “capstone” course, in the sense that it requires students to use knowledge they have acquired from a broad range of other CS courses, both theoretical and practical. They put this knowledge to work in the context of implementing compilers – large and complex program translators that they write mostly from scratch in groups. Our department does not offer a software engineering course, so it is important for students to have the opportunity to wrestle with big group software projects in the context of upper-level courses like CS301.

I have taught CS301 twice: once in Fall ’96, when I used Java for the programming projects, and in Fall ’00, when I used the ML programming language (also taught in CS251) for this purpose. In Fall ’00, I loosely followed Andrew Appel’s recent textbook, *Modern Compiler Implementation in ML*. The textbook is organized around a project for building a compiler from Tiger (a mini-language designed by Appel for his book) to MIPS assembly code. As in many other compiler courses, students are expected to complete a stage of the compiler every week or two and then assemble them at the end of the course into a complete compiler.

There are two problems with this approach:

1. Several design choices in the early stages of Appel’s compiler are influenced by factors that do not become apparent until students study later stages of the compiler. This is problematic from a pedagogical standpoint; I am uncomfortable with saying “ignore these details now; we’ll understand them later in the course”.

2. In a compiler course, it is not uncommon to fall into the “front-end rut”, where it takes more time than planned to cover the wealth of theory and techniques for the early stages of a compiler, like scanning and parsing. If this happens, then it is necessary to shorten or omit the coverage of many “back-end” stages of the compiler. In this case, there is the danger that the student projects may not reach “a point of closure” where all of the stages can be assembled into a working compiler.

To address these concerns, I developed an onion-skin approach to teaching the compiler course that was inspired by a similar approach I use for teaching interpreters in CS251. I organized the course as a sequence of passes through both front-end and back-end material. On each pass, we explored details and aspects not covered in the previous pass. I also decomposed Appel’s Tiger language into four subsets, each one more powerful than the previous one:

1. Kitty: a simple language supporting integers, loops, and input/output.

2. Bocat = Kitty + primitive datatypes (booleans, characters) + explicit types + functions.

3. Cougar = Bobcat + compound data (structures, arrays) + recursive datatypes.

4. Tiger = Cougar + block structure.

During the first six weeks of the course, we covered sufficient front-end and back-end material that students were able to complete a working compiler translating Kitty to MIPS assembly code. With this experience under their belts, they had a good working knowledge of the different phases of a compiler, and were ready to explore some of those phases in more detail. During the rest of the semester, they explored more advanced techniques in the context of a Bobcat compiler. Although time did not permit implementation of the Cougar and Tiger compilers, the onion-skin approach allowed the students to get hands-on experience with all stages of two working compilers.

Some of the materials I developed for CS301 can be found at the following URL:

`http://cs.wellesley.edu/~fturbak/tenure/cs301.html`

G Evaluating Computer Scientists

Attached is a copy of the following memo:

> Computing Research Association (represented by David Patterson, Lawrence Snyder, and Jeffrey Ullman). Evaluating Computer Scientists and Engineers for Promotion and Tenure. *Computing Research News*, September 1999.

This can be found on-line at `http://www.cra.org/reports/tenure_review.html`.

H Sample CMRHR Questionnaire

Attached is a sample of the CMRHR questionnaire I designed. This can be found on-line at `http://cs.wellesley.edu/~fturbak/tenure/cmrhr-questionnaire.pdf`.