Modern Technical Collaboration

Isaac Dupree

March 15, 2013

Submitted to Marlboro College in partial fulfillment of the requirements for
a Bachelor of Arts in Computer Science
with a concentration in Collaborative Programming

Sponsors: Jim Mahoney and Carol Hendrickson
Outside Evaluator: Andrew Cencini

Plan of Concentration judgment criteria as submitted on Final Plan Application

Field of study

COMPUTER SCIENCE/Collaborative Programming

Plan summary

An inquiry into collaborative programming through a series of essays and coding projects.

Components

30% A collection of essays discussing collaborative decision-making in software development. (“project”, Part I)

25% Prototyping a user customization language for the open-source game Lasercake. (Part II)

15% A paper examining Marlboro College’s communications infrastructure and its interrelation with disaster resiliency. (Part III)

30% Three CS exams in algorithms, programming languages, and the Linux operating system. (“independent”, Part IV)

Copyright note

I hereby license all computer source code printed in this Plan under the permissive MIT license[1], with the exception of code quoted in Part I which are licensed under the free/libre/open-source licenses of their respective projects. The prose and pictures remain under my proprietary copyright unless otherwise specified; I may change this if you ask.

[1] Copyright © 2012 Isaac Dupree

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Contents

Preface

I grew up in a family focused on mathematics, science and computers. I learned to program computers at a young age. To balance this out, I took some science classes at Marlboro but spent most of my time growing my arts and social-sciences. I took a dozen dance classes. I became a writing tutor. Meanwhile, I found summer software development internships and kept programming.

My Plan focuses on how social forces affect technical work (Part I), and on how, in times of disaster, technical infrastructure affects social existence (Part III). In Part II I build technical infrastructure for a computer language intended to be accessible to curious non-programmers; alas, this section is mostly technical: only a partial prototype and a few notes regard accessibility. The exams (Part IV) demonstrate my technical competence.

As a coincidence, around the same time as I started working on my Plan, I started working on Lasercake with my sibling Eli. Lasercake is an open-source, open-world game about the environment. It began when we were having fun discussing how to simulate water physics with better asymptotic complexity than Dwarf Fortress[2] did. Then we programmed those ideas. We added more to the simulation. We chose to consider it a project. We chose a name, created an online presence, and created core principles. In early 2013, we made Lasercake playable and posted an official release on `http://www.lasercake.net/`.

We each have a history of creating and playing games. I lost most of my interest in playing computer games a few years ago, when real life became too interesting; it amuses me that I am working on a game now. Eli has a website `http://www.elidupree.com/` and a plan to become Internet-famous. Ze[3] is multi-talented enough that I think this plan may work.

[2] Dwarf Fortress is a popular game where the player can build machines in three dimensions. Its graphics are a grid of colored Unicode characters. The player controls dwarves that fight, eat and build. Homepage: `http://www.bay12games.com/dwarves/`. All un-dated links in this thesis were accessed in 2012 or Q1 2013.

[3] gender-neutral pronoun

I introduce Peter Elbow's presentation of the "doubting game" and the "believing game" in Part I. The doubting game looks for everything wrong with an idea to make it better; the believing game looks for everything right with an idea to make it better. These ideas play a part in many places in this Plan of Concentration. Positive visions are important for creating things, as shown especially in Part I. On the flip side, looking at "what can go wrong?" can illuminate. In Part III I reference times of disaster to reveal the resilience of our technical infrastructure. In Linux exam subsection 24.4 and subsection 24.5 I examine what can go wrong in the presence of hostile cyber intruders. Attackers are a worst case; thus a good benchmark of understanding a system fully is whether I understand it well enough to thwart attackers. In Part II I design a language implementation that can thwart accidental or intentional attacks while remaining a useful part of a game.

Parts

Part I This essay discusses collaborative decision-making in software development. I reference numerous books and essays, and draw personal examples from Lasercake and tutoring.

Part II In this part I prototype a language for programming Lasercake's robots. The work here largely explores the technical foundation of a well-behaved sandboxed language. The work also contains some usability experiments and research. The source code is placed in Appendix Part V.

Part III This research paper describes infrastructure that Marlboro College depends upon. I use the lens of disasters to reveal the the resiliency of this infrastructure. The paper draws on interviews and personal experience of foul weather and disasters at Marlboro College.

Part IV I show technical competence in algorithms, in programming languages, and in the GNU/Linux operating system.

Linguistic style

As computer programmers (and Brits) are wont to do, I end quotation marks before punctuation whenever the punctuation is not part of what the quoted person said, "like this". I use singular "they" when referring to an indefinite person; it is sometimes the clearest way to say something. I happily use both British and American spelling, since I like most British spellings better but I am an American. Where I refer to specific people, I use their preferred pronoun.

Part I

Collaboration Between Programmers

1 The Cast Of Characters

I am interleaving acknowledgments with introducing the characters in my essays. After all, these essays are here to show what I know about people working together. *Thank you, everyone.* I am phrasing the acknowledgments *(in italics)* in a “without you, what would I lack” way as part of my essays’ theme. Examining where things hypothetically or actually go poorly shows the boundaries of our abilities. It fails to show what is commonplace. These acknowledgments help lead into my essay on computer-programming collaboration.

I go to Marlboro College, a small liberal arts college in Vermont. We have one professor in most disciplines, including Computer Science. Jim Mahoney is our CS professor. The number of CS students can be counted on one or two hands. I have worked on code with several of them, chiefly Sam Auciello, Elias Zeidan, and Noah Bedford. I started programming about 11 years ago, Sam about 8, and Elias a couple years ago. Noah is just learning programming proper, but has worked with gadgets and with Linux shell for many years. Jim graduated from MIT before any of us students were born, working on physics with heavy use of computers.[4] *Without a CS professor, I might not have gone to Marlboro. Without my fellow students, I would have not have learned much about in-person tech collaboration. Jim’s and Marlboro College’s flexibility allowed me to write this Plan of Concentration. My many dance and social science classes helped me understand better what it’s like for people to work together. This is important to my Plan: When I came to Marlboro, and again now*

[4] Everyone at Marlboro goes by their first name. Everything in the description of characters is from personal correspondence or personal knowledge. Jim’s experience is also at `http://www.marlboro.edu/academics/faculty/mahoney_jim/`

that they've both improved, my technical skills are stronger than my social skills. Before Marlboro, I think I had reached a point where, in order to get a technical project done, my social sense was the most limiting factor. Thus, studying social things was necessary for my technical strength to increase.

I grew up in Massachusetts with parents who both worked as computer programmers. When I asked at a young age how my computer worked, they were able to tell me. My sibling Eli eventually learned computer programming too. Eli and I talk about computer languages, mathematics, fiction, and everything under the sun often. Since January 2012, Eli and I have been on-and-off working on creating an educational game-simulation called Lasercake. *Without Eli, I would have weaker philosophy and a less confident direction in life. Without my parents, I might not have learned about computers until much later in life. (Also the usual acknowledgement: without my parents, I wouldn't exist and I might not have been raised so healthily.)*

I went to public schools. My parents were so knowledgeable about so many things that I didn't learn much additional from school until high school. In the meantime, I used the Internet for open-source. I learned to make graphical programs. I contributed to an indie game editor. I switched from Macintosh (which my family used) to GNU/Linux.[5] *Without the Internet, I would have learned technical things from books and would have had no one to share my work with. Without an audience, I might have grown less interested.*

Marlboro's emphasis on clear writing is lovely. It helped me develop my ability to think clearly. I learned to make precise statements, while still defending Truth by being ready for my statements to be wrong rather than by hedging every statement I ever make. *Without clear thinking, I could not learn by experiment and observation so well: a critical skill for understanding how computer systems work. Without a willingness to be wrong, it is much harder to be a reasonable participant in technical debates.*

[5] I list my significant programming acts at `http://www.idupree.com/coder`.

In 2009, I did Google Summer of Code, an internship-like program that pays students to contribute to open-source projects over the Internet. In 2011, I had an internship with Fog Creek Software. These both gave me chances to learn what serious technical work is like. *These internships helped show me what it's like to work with many people on a large body of code. Without them, Lasercake would be more of a mess*[6]*; I would be more worried about my employment prospects; and I would be less aware of my weaknesses as a paid programmer and thus unable to work on improving these.* I also found some tech meetups in the city, and visited a few software companies: in-person gives me such a different perspective than only online.

There are too many people to acknowledge at Marlboro – I learned from each teacher's and so many students' styles. My ex Plan sponsors deserve a note. John Sheehy, writing professor, taught me non-directive tutoring long ago, and was briefly a sponsor. This tutoring is one practice that helps us listen and hear each other. Kristin Horrigan, dance professor, was my advisor or sponsor for most of my time at Marlboro. She has listened to me even when I was being nonsensical, and taught me so much about working together in dance that is applicable to the rest of life. The choreographic feedback method pioneered by Liz Lerman[22], in particular, influenced how I see giving and receiving feedback, even when I am not using the process explicitly.

[6] It would have more "technical debt". "Technical debt" refers to the state of code that might work now, but is difficult to change because of how it's written. Experience and knowledge of good programming practices helps one write code that has less technical debt, but there's always some. It's called "debt" because you pay it back by refactoring the code to be more malleable before you can do substantive changes.

2 Annotated bibliography

2.1 Socially oriented works[7]

There are many books I've read, either for class or fun, that discuss ways of working together. Lerman's methods share much thinking in common with *The Practical Tutor* [23], *Nonviolent Communication* [38], and *Getting To Yes* [8]. *On Conflict And Consensus* [4] takes similar philosophies of explicit, sensitive communication and deliberate listening, and applies them to group decision making. The Haskell community uses this consensus-based process for its standard-library decisions; I was a small part of how we came to that decision, and I serve on the Haskell Platform Steering Committee, whose role is to facilitate, not to hold any authority.[8] Starhawk's recent *The Empowerment Manual* [42] shows more deeply the ways of thinking of the concise *On Conflict And Consensus* and the group processes related to it.

There are several essays and books by computer programmers that focus on their collaboration, reviewed in more detail below. *The Mythical Man-Month* [3] describes the time costs of working with others and ways to make this more efficient. Stallman's Free Software philosophies [41] left a mark on my software choices and showed me some risks of unfreedom in software. (It did not prepare me for how today's FOSS[9] culture is particularly unfriendly to women, even more than proprietary or academic computer programming is.[10]) "The Cathedral and the Bazaar" [34] showed a way of working on a project transparently and open to casual contributors; by the time I got into open-source, this essay had so permeated the culture that I learned most of its suggestions by way of contributors just stating them as the way things are.

Turtles, Termites and Traffic Jams is a book about a learning system where one can

[7] The distinction between this and the next subsection 'Technically oriented works' is fairly arbitrary.

[8] `http://trac.haskell.org/haskell-platform/wiki/Members#SteeringCommittee`

[9] Free and Open Source Software

[10] See `http://infotrope.net/2009/07/25/standing-out-in-the-crowd-my-oscon-keynote/`, mentioned on `http://geekfeminism.wikia.com/wiki/FLOSS`

write programs for little creatures and watch what they do as a result. It comes from the constructivist school of learning. Learning methods are significant to this paper, as a great deal of our knowledge about how to program effectively is passed on from person to person, whether in-person, online, or through books or code.

I find that a critical part of creating a program is finding an effective way to think about the problem it addresses. I find my tutoring (of academic paper writing) often to be about helping the tutee find more ways to think about the paper they're writing. Helping with computer programs is often similar.[11] This is all about learning from each other and/or creating knowledge through clever thought.

2.2 Technically oriented works

There is an occasional tradition of programmers and researchers writing about the craft of programming. The writings in this list are stops along the way that were either broadly influential, or important to me.

"Go-To Statement Considered Harmful" by Edsger W. Dijkstra, 1968 [6]: This is a famous historical essay. It was titled "A Case against the GO TO Statement" by its author but renamed by its publisher. It is one of the early thoughts that there are good and bad ways to write programs. It is still remembered today. So many essays "[Something] considered harmful" were written in homage to it that someone wrote an essay " 'Considered Harmful' Essays Considered Harmful" [24]. "Go to" statements are rare enough today that it is mainly of historical interest. Its principles are still true. It's good for code's meaning and effect to be easy for humans to understand.

The Mythical Man-Month: Essays on Software Engineering, by Frederick P. Brooks, 1975 [3]: Brooks is a computer programmer. He had been in the profession for decades when he

[11] Tutoring about a computer program also has abilities that essays don't, such as testing and debugging by seeing what it does when executed. Would you say this knowledge of how things go wrong is created by the computer? In some ways it works to think of a computer like a sapient being.

published this book. He draws from examples of project management in his career at IBM. The title chapter describes why a software project often can't be completed faster by adding more people to it: the work needed may be unable to be split into independent components (one or more for each person), and even if it can, the pairwise communication time between developers increase quickly with more developers. The experiences he describes are from a time when only large companies could afford a computer, but the book has aged remarkably well. Most of its points still ring true today.

Mindstorms: Children, Computers, And Powerful Ideas, by Seymour Papert, 1980 [29]: By this time, some ordinary people could get access to a computer. This was before the World Wide Web. Papert started to think about how children can learn through computers. He describes children learning by playing with code. He co-created a language, Logo, for this purpose. In Logo, there is a "turtle" cursor that the code can move around as if it's a robot, and that can draw draw on the screen as it moves.

Free Software, Free Society: selected essays of Richard M. Stallman [41]: Richard Stallman was part of the MIT "hacker" culture where sharing programs was the norm. Then he ran into someone who was legally bound not to disclose their program to him. He got angry at this sign of the loss of a sharing culture. He cared enough that in the 1980s he created most of an operating system, GNU, that was legally bound to be free to share and modify, and the GPL copyright license by which this was enforced. (In 1991 Linus Torvalds created as a hobby the kernel Linux, and happened to license it under GPL. This plus GNU formed a complete operating system. The foibles of history caused this system to be called merely "Linux" by most people.)

Turtles, Termites and Traffic Jams: Explorations in Massively Parallel Microworlds, by Mitchel Resnick, 1994 [36]: Resnick builds on Papert's ideas. Resnick builds a language StarLogo where there can be thousands of "turtles" at once, each running the same program. It is a way to explore emergent behavior that can come out of simple turtle programs. Resnick also aims at children's experimentation.

"The Cathedral And The Bazaar", by Eric S. Raymond, 1997 [34]: It was common, even in free and open-source software, for a small group of developers to work together in private and occasionally release a new version of their software to the public. Raymond calls this the "cathedral" model of development. He observed that Linux (the kernel started by Linus Torvalds) was being very successful at releasing often, discussing in public, and accepting contributions quickly from many users. He dubbed that the "bazaar" model. This essay quickly became famous. It eventually led GNU projects and many others to switch to the bazaar model. Raymond cites Brooks' The Mythical Man-Month often. He notes "Brooks's Law predicts that the complexity and communication costs of a project rise with the square of the number of developers, while work done only rises linearly.", and speculates the bazaar model mitigates that by having just a few core developers who communicate with everyone else who are working on mostly separate tasks.[12]

"How To Ask Questions The Smart Way", by Eric S. Raymond, 2001 [35]: This document, also by Raymond, has been referenced by a substantial fraction of the open-source projects and similar forums I've participated in. It is a guide to how to participate in a free/open-source community when, like all people, you are not all-knowing and all-understanding. By the time this was written, bazaar-style development was common. The Web had been in full force for several years; the essay's second bullet point for "Before You Ask" is "Try to find an answer by searching the Web.". Search engines had existed for a few years; Google, which was significantly better than those that came before, was founded in 1998.[13]

"Beating the Averages", by Paul Graham, 2001 [13]: Paul Graham notes that by using better tools, a programmer can be significantly more productive than those who don't. He tells a story of a startup of his that used Lisp (the better tool) rather than the popular

[12] Raymond, section How Many Eyeballs Tame Complexity

[13] `https://www.google.com/about/company/`

languages of the time.

“The CADT Model”, by Jamie Zawinski, 2003 [46]: This short essay laments the common development non-methodology of letting users report bugs, mostly ignoring them, and then rewriting the relevant part of the software (so it now has different bugs, not yet known but probably in similar quantity). He says “Let’s call it the ‘Cascade of Attention-Deficit Teenagers’ model, or ‘CADT’ for short.”

Code Reading: The Open Source Perspective, by Diomidis Spinellis, 2003 [40]: This book notes that reading code is an effective way to improve one’s craft at programming, and thanks the free and open-source movements for producing so much code that everyone is permitted to read. It is inspired by the 1970s *Commentary on the Unix Operating System* by John Lions.[14]. It notes that, in the intervening time, not much had been written on the subject. The first source code I read of a large program was of the game Angband. I didn’t understand its code very well at first, but much later I came back to it and understood how it was mostly sensible code and how to read it easily. I believe that being exposed to the code, even without understanding it, helped me learn what a nontrivial program’s code might look like.

Producing OSS, by Karl Fogel, 2005 [9]: This is a matter-of-fact guidebook about how to run an open-source project, covering everything from mailing-lists and bug tracking systems to choosing a copyright license and style of leadership. Where there are multiple options in common practice, it describes them all. This book follows the principle of freedom for itself: it can be read for free online and shared or modified under Creative Commons Attribution-ShareAlike,[15] and it can be bought in print from publishers. Karl is updating it for 2013, funded by through Kickstarter donations.[16]

Open Advice: FOSS: What We Wish We Had Known When We Started, edited by Lydia

[14] Spinellis, p. xxv

[15] `http://producingoss.com/en/copyright.html`

[16] `http://www.kickstarter.com/projects/kfogel/updating-producing-open-source-software-for-2nd-ed`

Pintscher, 2012 [32]: This book is a collection of 42 prominent open-source programmers' essays responding to the title prompt. It is free/libre online and sold in print.

10 PRINT CHR$(205.5+RND(1)); : GOTO 10, by ten people, 2012 [27]: This is the story of the folklore BASIC program that is the book's title. The program creates art. It is delightful to see how much can be written about one line of code; how many perspectives there are on its pieces, its whole, its effect, and its social existence. Programs as short as this one need not be version-controlled; they can recorded naturally in the text of an email, for example, and a change to the program is just as naturally represented by a complete copy of the new version. It is free/libre (except for commercial use) online and sold in print.

"What Colour are your bits?", by Matthew Skala, 2004 [39]: A popular philosophical essay contrasting digital data with the intuitive laws and expectations we might assume make sense for it.

"What Every Computer Scientist Should Know About Floating-Point Arithmetic", 1991 [10]; "Gay marriage: the database engineering perspective", 2008 [15] (2008): popular technical essays about the perils of floating-point arithmetic and about relational database engineering, respectively.

3 Knowledge dynamics: Competence

Lasercake's philosophy is constructionism[17] if there's a world, imagined or real, where the player can make things, then the player will play around and learn. My impression is that this sort of learning is natural for some people and doesn't work for some people. It works for me and for enough people for us to build things for these people.

[17] See *Turtles, Termites and Traffic Jams* p. 23 [36].

3.1 Learning by doing: Lasercake's water system[18]

The water system we set out to create began with a few premises. The world consists of a three-dimensional grid of equal box-shaped tiles[19], each containing air, water or rock. Water should move fairly realistically. The asymptotic efficiency should be fairly good for large amounts of water flowing.

Eli was inspired by having built large engineering marvels in the freeware/non-Free[20] adventure/engineering game Dwarf Fortress. One creation used water and lava to build a large sphinx statue that could have lava pour out from its mouth on command. The Dwarf Fortress simulation, however, became slow when Eli created large aqueducts. If there were n contiguous water tiles, the time taken to simulate the tiles flowing was proportional to n^2.[21] Dwarf Fortress is tile-based and three-dimensional, and Eli and I started brainstorming how we could do better.

We considered keeping track of connected groups of water tiles as single entities. We wrote a simulation based on this idea. It turned out to work well. So we poked it to find out what unrealistic things that it did. For example, in theory, water in a bucket with a spout should spurt out the side of the bucket. It should spurt faster if the spout is lower on the bucket.

It turned out that our implementation didn't do this, so we made it do so. We made the

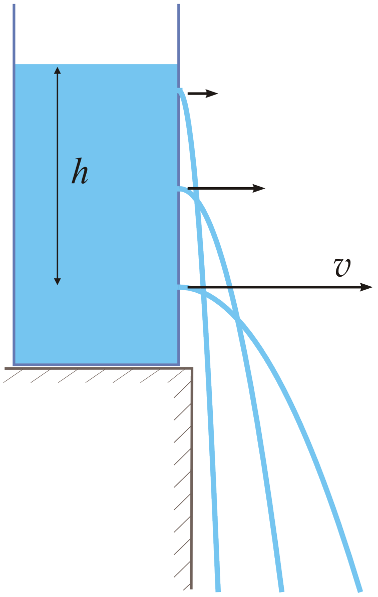
[18] This section describes Lasercake's water system as of 2012 and early 2013. We are likely to change it again in the future.

[19] The tiles could be cubes, but we chose to make the boxes' height be about 5 times shorter than their widths to make landscapes look more realistic.

[20] Free/Libre/Open-Source Software, by definition, permits everyone to modify the software and distribute those modified versions. Dwarf Fortress 0.34.11's readme.txt states:

Copyright (c) 2002-2012. All rights are retained by Tarn Adams, save the following: you may redistribute the binary and accompanying files, unmodified, provided you do so free of charge. If you'd like to distribute a modified version of the game or portion of the archive and are worried about copyright infringement, please contact Tarn Adams at t...@bay12games.com.

[21] This is an observation about Dwarf Fortress in 2011, based on it empirically being very slow and documentation stating this specific asymptotic complexity. Formally, its water simulation was $O(n^2)$ in the number of contiguous water tiles. To our knowledge, it still is this way, but that doesn't affect Lasercake's history.



Water spurts out of a bucket at different speeds at different heights.[22]

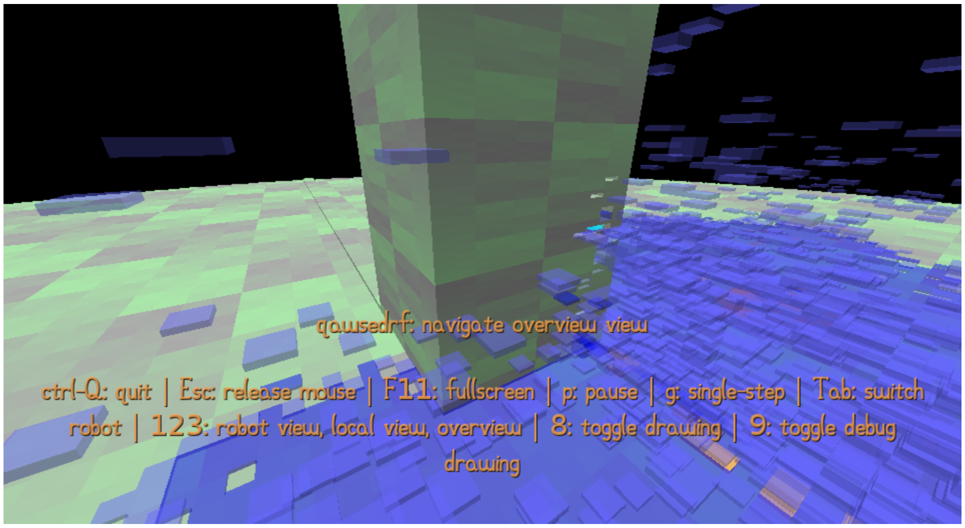
code measure the amount of water pressure above anywhere that water is flowing to, then apply the physics-correct amount of force.

Then we found another problem, and fixed that, and so on. We documented it. We extended the implementation to simulate similar but simpler granular flow (sand, flour, gravel). We found some of its ideas just didn't work.[23] We documented its existing problems. We changed it more. We fixed the documentation. We moved on to other parts of the Lasercake simulation, but occasionally came back to this.

3.1.1 Programming as writing

This process can be called "iterating on a design". Here, though, I'm interested in nontechnical ways of seeing this process. For me, coding is like writing. I write something, look for things that make sense in what I wrote, then write more things, repeating until I'm happy. I show the writing to someone, then, armed with their perspective, re-write more. As my

[23] It's hard to remember which ideas these were. They were re-formed into ideas that are more internally consistent.



qawsedrf: navigate overview view

ctrl-Q: quit | Esc: release mouse | F11: fullscreen | p: pause | g: single-step | Tab: switch robot | 123: robot view, local view, overview | 8: toggle drawing | 9: toggle debug drawing

Water shoots out of holes in the side of a water tower and puddles on the ground in Lasercake 0.22.

thoughts become clearer about a topic, I become exponentially[24] faster at writing about it. Beginning to write is one of the best ways to find this clarity.

Peter Elbow in *Writing Without Teachers* describes this process as “growing” and “cooking” writing [7]. He says that people conventionally think about writing as a process with two steps, deciding what you’re going to say and then saying it. Instead, he argues, one often cannot know what one is going to say ahead of time.[25]

> “Think of writing then not as a way to transmit a message but as a way to grow and cook a message. Writing is a way to end up thinking something you couldn’t have started out thinking. Writing is, in fact, a transaction with words whereby you *free* yourself from what you presently think, feel and perceive.”[26]

Coding is like writing: a process widely regarded to be mysterious done chiefly by banging on a keyboard to make a sequence of words and symbols. Coding is unlike writing in that, not only is one writing for other humans, one is writing for a machine.

[24] Dramatically faster, with a horizontal asymptote at my typing speed. I don’t know the actual function.

[25] pp. 14-15.

[26] p. 15.

3.1.2 Machine as reader

The machine's feedback is ruthless, unchanging, and unemotional. But it is also free[27] and consistent; the machine never gets angry. Some machine feedback is trivial, like "error: missing semicolon". Some is helpful, like "error: tried to use a distance (meters) as a time quantity (seconds)".[28] Some feedback is something a human editor could never give. When we piled on dozens of rules to make the water simulation do what we wanted, even we didn't know all the ways these rules could combine. But our computers were happy to do billions of computations based on these few rules. We then could see what behaviour emerged.[29]

To some extent, this latter sort of feedback also applies to programs that aren't simulations. As Brooks notes, larger programs take more time to make than smaller ones, to a greater extent than the proportional amount one might expect.[30] For a program with n parts that interact with each other, there are n^2 interactions. The machine may eventually try each of the interactions. If one hasn't been tried before and the program is incorrect, then the machine will happily do that incorrect thing in that case. As Perrow notes in *Normal Accidents*, for tightly coupled complex systems, unexpected failures are nigh-inevitable [30]. I discuss this in more detail in Part III. For computer programs, it means trying to decrease unnecessary interactions between components, and making a failure in one component less likely to cascade to the next component and cause it to fail.[31]

[27] "Free" laying aside the serious problems of the digital divide. Computers are not cheap. Developer tools on some platforms are not free. But once one has them, they never get bored, the way a friend might, of giving feedback. They rarely give feedback unasked. They are like an introspective, eccentric friend.

[28] These are not real quotes of error messages, but they are real errors caught by the computer. Some actual error messages are as clear as these, and some are extremely obtuse. Real messages also say which line of code they refer to. For example, I introduced a typo into a piece of code, replacing 'show' with 'shod', and the compiler reported:

```
main.cpp:975:17: error: 'class LasercakeGLWidget' has no member named 'shod'
```

[29] This last sort of feedback is why Mitchel Resnick developed StarLogo. He writes, "[A] user might write simple programs for thousands of 'artificial ants', then watch the colony-level behaviors that arise from all the interactions." Mitchel Resnick, *Turtles, Termites and Traffic Jams: Explorations in Massively Parallel Microworlds* [36], pp. 5-6.

[30] Frederick P. Brooks, *The Mythical Man-Month: Essays on Software Engineering* [3], p. 88.

[31] Some specific programming language elements that can help: objects with private data; modules; functional programming that discourages or bans mutation. These alone are not sufficient, however.

In many programming languages, like Python (*dynamically typed* languages), the computer will only notice errors while running a program. Meanwhile, the incorrect program might have done something incorrect. If the program does things with files, in an unlucky case it might have deleted some of one's files. Alternatively, if part of the program is rarely run but has an obvious flaw, that flaw might not be caught for a long time. I am personally a fan of languages which catch most errors before running the program (*statically typed* languages).[32] 'Types' are categories of values, such as 'integer', 'list', and 'text'. If you accidentally try to use an integer value where you need a list, sooner or later it won't work.

Another reason I like statically typed languages is the human aspect. These languages give the programmer a way to tell the machine what types things in the program have. As a positive side-effect, the programmer now has a precise way to talk to other programmers about these types.[33] And talking with other programmers is important.

3.1.3 Human as reader of code

> *Most programming courses and textbooks focus on how to write programs from scratch. However, 40% to 70% of the effort that goes into a software system is expended after the system is first written. That effort invariably involves reading, understanding, and modifying the original code.*
>
> –Diomidis Spinellis, *Code Reading: The Open Source Perspective*, p.1

To some extent, our water physics is a "write-only" system. It is code whose most natural way to improve is the CADT model.[34] We wrote it and fiddled with it, but the more tweaks

[32] I find, for example, that Haskell catches about 2/3 of my mistakes before running a program, and C++ (also statically typed, but different) catches about 1/3 of my mistakes. Lasercake is C++. Python (dynamically typed) only catches about a tenth of my mistakes before running my program; tools like 'pychecker' that check Python code for mistakes catch, maybe, another tenth of my mistakes. Not all static typing is worth it: I find C's static typing ineffective enough that it has the downside of being irritating without the upside of catching my mistakes.

[33] Python lets one talk about lists, but Haskell or C++ lets one get specific and talk about *lists of numbers*, or, say, *lists of pairs of names and numbers*.

[34] "CADT": A pattern called "Cascade of Attention-Deficit Teenagers" by Jamie Zawinski as a deserved

we add to make it more realistic, the more likely it is to go wrong in some other way.

I insisted that we at least document it so that we'd even know what it did later. The prose documenting it is 200 lines[35] at the beginning of a 2000 line source-code file.[36] Eli understood (and understands) this system better than I do. So I couldn't write the full documentation. I wrote documentation for another system I did understand. This motivated Eli to document things better. Eli already had the intellectual idea that documentation was good. Perhaps[37] learning information from my documentation helped Eli see how to document. And at least for me, I find it motivating when a piece of code I'm working on keeps improving even when it's not my work. When Sam and Elias and I were working together on a class programming project[38] we all commented on the niceness of this.

Next (still wanting water-system documentation, and with Eli's acceptance of this method) I wrote documentation for it myself, knowing that some of what I said was probably wrong, and committed this documentation[39] This let Eli tell me what was wrong with my documentation and fix it.[40] Later, after Eli rewrote the code again, ze wrote a new comment at the top of the file describing the new semantics.[41]

insult. Instead of fixing bugs in an existing piece of code to gradually make it better, one rewrites the entire piece of code now and then, replacing old bugs with new and different bugs. `http://www.jwz.org/doc/cadt.html` [46]

[35] It's 1900 words.

[36] tile_physics.cpp

[37] I don't speak for Eli. In fact, our minds work differently in many ways.

[38] A mathematics class project to re-implement part of the computer proof of the four-color theorem. I extended it using GraphViz to create visualizations of parts of the four-color theorem arguments: `http://www.idupree.com/four-color/`. The code lives at `https://github.com/olleicua/Py4colorReduction`.

[39] commit `3d8bb5a697`, "Add long comment describing water physics mildly thoroughly." by Isaac

[40] commit `c7101c2add`, "edited Isaac's water rules comment a bit" by Eli

[41] commit `83e5ddea5f`, "Wrote a giant comment describing the tile physics." by Eli

A major risk of any documentation that's not machine-checked to be consistent with the code is that it might be wrong. It's quite common for people to change the code without seeing all related documentation that needs to be updated. Wrong documentation is worse than none. To remedy this, Brooks suggests making programs self-documenting to the extent possible. Every identifier in code should be named meaningfully. Where the code uses standard algorithms, concisely cite thorough treatments of these algorithms. This way code modifications will naturally update the "documentation" as well. (Brooks, *The Mythical Man-Month*, pp. 174-176 [3].) Some programmers, for these reasons, state that comments in code are a "code smell" — something that indicates the code is not so good. See `http://www.codinghorror.com/blog/2006/05/code-smells.html`.

This documentation is very helpful, but I still don't entirely understand the code. After a year, Eli didn't remember all the details. The comment and the clearly chosen names in the code helped. Water tiles that are flying alone through the air are not "groupable", and water tiles that are in a mass of water tiles are "groupable". Groupable water tiles that are surrounded by other water tiles are "interior"; groupable water tiles that are not surrounded by groupable water are "surface tiles". "Suckable" tiles are water tiles at the highest elevation of a water group, and "pushable" tiles are air tiles next to the water group. Thus, water flows by a water tile being sucked out of a suckable tile and placed in a pushable tile. We had long discussions to find good words for these concepts. Good words for concepts remove one level of complexity from thinking about complicated code.

A year after writing the water code, we were adding a system of units to our code. We've always intended certain values to represent meters, others to represent seconds, and so on. I created a system[42] that lets us write the units of quantities using the language's type system. This lets the compiler check computations for dimensional consistency. This also ensures that we know what units every variable is supposed to have. As we'd predicted, implementing this system found some bugs and mistakes in our code. I adapted and fixed most of the code to use units explicitly. At the end of the week I had done most of it and was struggling to figure out what units the water physics *meant* in many places. I discussed these with Eli and suggested ze convert the water physics code to use explicit units.

I "suggested". What does this mean? Well, I felt frustrated. I imagined that Eli could do it. I could have told Eli to do it or passive-aggressively convinced zem to.[43] *I used Nonviolent Communication[38] principles that made me less coercive. I reflected on the situation: I had started doing this code in a day, and imagined it'd take a day. It turned out to take a week. I was also busy writing this thesis. It was code especially bad to take a break from and finish*

[42] defined in units.hpp

[43] Not that these aggressions would *work*. But that doesn't always keep me from doing them.

later. It made minor changes to almost every code file; it would've been painful for us to merge these changes with other code changes that Eli was soon to be working on. I discussed my situation with Eli so that we could decide what worked best for each of us. I appreciate that Eli chose to help out in this conundrum, even though it was mostly my fault. It was in part an interpersonal request, but the logic of the project's needs also gave us a common vision.

It helped that Eli was familiar with the water physics code: Eli was able to tell which parts of the code were just plain nonsensical,[44] and which parts pretty much make sense. Some of the knowledge Eli used to apply units to this code resided in the code itself, some in the comments, and some knowledge resided in Eli's mind. Then Eli and I created some knowledge. We discussed what to do with the broken parts of the code. We deduced what would break things least. (The computer, of course, provided mechanical knowledge: it found every instance of assigning a distance to a variable that represents velocity, for example. The computer also gave us fuzzy knowledge by simulating the code and letting us see that it still looked pretty much like it did before.)

4 The uses of negativity and positivity: Perspectives/Vision

I'll continue using the water physics as an example.

Peter Elbow talks about the "doubting game" and the "believing game".[45] The doubting game is the pursuit of knowledge by doubting the validity of statements, or arguments, or perhaps code. One looks for everything wrong with an argument. Where there are flaws, one tries to adjust the argument to lack those flaws. In the end, if all goes well (which is

[44] The water physics has lots of hacks (pieces of code that may be pragmatic but don't entirely make sense) that help it seem more realistic without actually being that realistic.

Science is an important thing in Lasercake, and one of the things this code fails to maintain is conservation of energy, so we are going to rewrite this code again.

[45] Peter Elbow, *Writing Without Teachers* [7], p. 147.

rare), one is left with an argument that is free of flaws.

The "believing game" is the pursuit of knowledge by imagining things. In the imagining thing, one takes a proposition and thinks of cool things it could imply. It does the building of ideas that the doubting game needs; one can only make trivial fixes to an argument without imagining. One sets aside doubts for a while because they get in the way.

We play the doubting game with the water physics when we say "It looks unrealistic in this situation". Then the believing game is "How might we make it look realistic?". Then we implement this and doubt it: it helps most examples (though some are worse now), and it makes the simulation slower (but maybe we can imagine again a way to restore the speed). Elbow describes this alternation as a dialectic.

We couldn't have made the water simulation without the believing game. Our spark was Dwarf Fortress, which has $O(n^2)$ (n = volume) water-flow computation time.[46] This is far too slow; no adaptation of its approach would be good enough. We'd like at least as fast as $O(n)$. As it turns out, our approach can be even better, $O(n^{2/3})$, much of the time: we only need to look at the surface tiles, not the main body of the water, to simulate it well.

As someone who cares about the quality of Lasercake, I have applied both "games" to the water physics more broadly. I insisted to Eli, once we had a water system that was somewhat decent, that we make a point to avoid regressions. Regressions are when a newer version of a program is worse in some aspect than an older version of a program. They are often looked at as worse than regular bugs.[47] If each version is better-and-not-worse than the previous version, we can be sure we're making progress. If a new version is better-and-worse than the old version, it's less obvious.

Eli was somewhat dubious, for good reasons. Often a change is mostly an improvement

[46] $O(n^2)$ here means that the time taken is proportional to the volume squared. $O(n)$ means that time and volume are proportional. $O(n^{2/3})$ indicates that time increases somewhat more slowly than volume.

[47] For example, Linus Torvalds and Jonathan Corbet note at Linux.conf.au (a conference) how thoroughly Linux kernel regressions are discouraged: `http://apcmag.com/linus_torvalds_on_regression_laziness_and_having_his_code_rejected.htm`

despite significant downsides. The water code was still not perfect and also had a few fundamental limitations. One of the fundamental limitations is that the simulation has no idea which direction the water is flowing. If a leaf lands on the water or a fish swims in it, the simulation can't tell it which way to drift. For water in twisty tunnels, the direction is not always obvious. Yet we'd like this feature.

So we've also been dreaming and brainstorming and trying things. We make sure to keep these attempts separate from the present Lasercake water code until they're ready to replace it. This lets these dreams be fanciful. Some of them are good but too slow. Eli has lots of mathematical people with whom to brainstorm how to do something similar but faster.[48] And indeed we will need to change the water physics. For one thing, the current code does not nearly conserve energy (kinetic and potential energy), and it's not trivial to make it so. We can likely rewrite tile-based water in a similar conceptual model and achieve it.

But Lasercake is a game about the environment. We plan to implement sunlight and wind and diffusion and several other things. The math we've found that can run these fast enough (with our constraints about what we do and don't wish to model) is continuous 3D geometry with arbitrary polygons. Tiles in a grid are not a good fit for this. If we shall have have those 3D polygonal structures anyway, maybe we can use them *instead of* tiles to represent terrain and water as well. Robots already moved freely in 3D space, forcing conceptual complication to let tile physics and robot physics interact. The unrealisticness of tiles was never entirely satisfying. Our tiles are two meters high. That's not a large distance in-game, but neither is it something that a wheeled robot could realistically climb up.

We could never have dreamed of these things without the "believing game".

[48] Higher levels of mathematics are largely about dreaming of abstract structures, not about questions with a single right-or-wrong answer. Mathematics is about rules, but the rules are like paints on a canvas: we get to choose out of billions of possibilities and find out if they work well together. Eli's major is mathematics; my parents do maths; Marlboro's maths professor Matt Ollis helped me see something about how to see mathematics.

4.1 Vision

> "What is energy? Where does the energy we use come from? How do we gather and process materials, and what effects does that have on the planet we live on?
>
> The Lasercake project aims to help people understand these things through the powerful medium of computer games.
>
> In Lasercake (the game), the player can use robots to build industrial projects – but unlike in similar games, every part of the world is based on real-life science. Mine waste has to be dumped somewhere and causes pollution. Energy is conserved. Solar panels, wind turbines, and so on, harvest realistic amounts of energy."[49]

Eli takes more lead on creative vision like this; I take more lead regarding the technologies we build on. These match our strengths.

We are aiming to create a project that people can join. Many open-source projects function this way. Starhawk observes "Collaborative groups are generally started by people with a vision... [A] clear vision is actually a gift to a group. An articulated vision lets prospective members know what they might be choosing to join, and it creates a standard against which your decisions can be judged."[50] I did not expect our current vision when we began Lasercake. I envisioned it more as a place to make contraptions and play with robot programs. Then I listened to this vision of environmental science for a while and grew to like it. It is a sound vision. It can lead us for a long time. It is not a project that I know everyone else to be doing too.[51]

On Conflict and Consensus describes the dynamics of groups that use consensus decision-making process.[52] It is a concise handbook on using that process. Consensus is the only

[49] `http://www.lasercake.net/blurb`

[50] Starhawk, *The Empowerment Manual: A Guide for Collaborative Groups*, [42].

[51] By contrast, there are thousands of Sudoku iPhone apps, yet people still make more. It's tempting to seek success by doing what someone successful has done. Yet creating something new gives the distinctive position of being there first. Our vision, as it happens, also helps explain how Lasercake is not here to mimic Minecraft, a well-known game that also has tiles and is about building things (`https://minecraft.net/`).

[52] C.T. Lawrence Butler and Amy Rothstein, *On Conflict and Consensus: a handbook on Formal Consensus decisionmaking* [4].

nonviolent decisionmaking process[53], so it's worth knowing. The handbook shows how a statement of purpose is useful for a collaborative group. "If the group discusses and writes down its foundation of principles at the start, it is much easier to determine group versus individual concerns later on." Principles help people work together exactly because the principles of the group are not tied inextricably to the principles of its members. Of course, people who choose to be in the group probably personally agree with most of its principles. The process still accommodates people who don't.

4.2 Negative vision

In some parts of Lasercake, playing the doubting game facilitates a positive vision.

The simulation has a strict view of determinism. Given the exact same initial conditions and user input, the simulation should end up in the same state, every time on every platform. To make this the case, we need to be extra careful in many ways, doubting anything that might disturb the determinism.[54] If we're not sure whether something would compromise

[53] See e.g. On Conflict and Consensus p.5, "Formal Consensus is the least violent decisionmaking process". Graeber's *Fragments of an Anarchist Anthropology* makes an interesting argument that majority rule is fundamentally based on the idea that we vote to find out who would win if it came to physical blows. On consensus, "It is much easier ... to figure out what most members of [a] community want to do, than to figure out how to convince those who do not to go along with it. ... If there is no way to compel those who find a majority decision distasteful to go along with it, then the last thing one would want to do is to hold a vote: a public contest which someone will be seen to lose. ... What is seen as an elaborate and difficult process of finding consensus is, in fact, a long process of making sure no one walks away feeling that their views have been totally ignored." [12], p. 89.

[54] Examples (technical):

We use explicitly-sized integer types, e.g. int32_t, because the size of int or long can vary. To prevent unspecified behavior on overflow, we add the compiler flag `-fwrapv`. To prevent non-determinism-related bugs due to overflow, and because `-fwrapv` is nonstandard, we have an integer wrapper class bounds_checked_int that throws an exception if an operation would overflow.

We do not use floating-point in the simulation. The graphics code uses floating-point, but is carefully designed not to affect any other part of the simulation. It is still too difficult to guarantee on every platform that it will round exactly the same. (Also, we are interested in principles like conservation of energy. Rounding error will be necessary with floating-point or integer; if we use integers it is easier to measure the precise amount of rounding error and make up for it.)

We avoid mutable global variables. This is not strictly necessary for determinism, but it makes it easier to avoid mistakes, and is also good coding practice.

We ought to define our own hash function for hash-tables, because the standard does not define which hash function it uses. We can try to never depend on the order of elements in a hash table, or to make that order determinism, it's better not to use it. Determinism is not as trivial to test as whether the Lasercake compiles and runs. It is most likely to go wrong on obscure platforms we do not have access to.

But once we have determinism, we can do very cool things. We can store a complete history of the game just by storing the user input data. (This makes it feasible to travel back in time!) We can let players double-check each others' computation of the game state, making nonconsensual cheating harder. Determinism makes parallel computation easier. Determinism makes debugging easier. And so forth.

Similarly but with a twist, we care about security. The simulation is complicated and written in a memory-unsafe language (C++). This makes it more likely that a user can use bugs to gain control of the program. Robots are going to be programmable; user programs make it even easier for users to try to poke holes in the simulation code. There will be networked games; thus people can send their malicious game acts across the wire. I feel it is not quite ethical to make this game without some greater protection. Luckily for us, Chromium (Google Chrome) has this problem a thousand times worse. Its developers have found ways for a program to protect itself and documented them.[55]

If it were not for the concept of computer security, I might not have realized the risk and not try to protect against it. I'm not even sure whether computer security is positive thinking ("safety") or negative thinking ("nothing goes wrong").

deterministic, or both. I am not certain that the standard defines how unordered_map turns hashes into positions in the table; if we continue to use hash tables we may want to copy a particular implementation version and stick with it.

These are some of the risks that the "doubting game" has found for us regarding determinism.

[55] `http://dev.chromium.org/developers/design-documents/sandbox`

5 Roles and platforms

5.1 Technical infrastructure

Open-source projects often have an email list, an IRC channel (real-time group chat), a bug tracker, a revision control system, a website, and sometimes a wiki, forums, blog, and/or planet[56]. Archives of e-mail and sometimes IRC are posted publicly online. This is a lot of things! IRC helps developers help each other, if they are online at the same time. E-mail is used for important decisions or information that nobody should miss because of their timezone. In-person discussion (where feasible) can be even better than IRC, but it excludes people who were not there. It is bad for a collaborative group when someone feels excluded from a decision-making process.[57]

Revision control systems make it easier for two or more developers to work on code at the same time without getting in each others' way. They also let developers give changes a Wikipedia-style edit summary. This makes it more feasible to search a project's history – even if it is a one-person project![58]

Bug-trackers exist because people find problems before they find solutions. In the meantime, we record the problems in a bug tracker so we don't forget what they were.

We chose[59] GitHub for hosting the revision control. It is free-of-charge for open-source projects, and extremely popular. GitHub has integrated issue tracking (bug tracking); we use this. It has a wiki which we're not currently using. It does not offer e-mail lists. The

[56] Planet: an aggregation of project members' blogs, such as `http://planet.haskell.org/` or `http://planet.debian.org/`.

[57] Many of my sources note this – *Nonviolent Communication*; *Getting to Yes*; *On Conflict and Consensus*; *Fragments of an Anarchist Anthropology*; and so on. I've seen software developers note this effect as well.

[58] I've often found it helpful in my own small personal projects to help remember what I've done. The system also shows me what I've just changed in files I'm working on. This lets me find the many accidental, unwitting text deletions or insertions that I make! It also helps me organize changes into logical pieces. This improved my coding. It also took me years to get used to doing well. Sam is noticing a similar progression while learning revision-control systems.

[59] at my initiative, because I was more familiar with the choices

Mailman e-mail-list management software is something of a standard in free/libre/open-source projects, but I didn't feel that the reliableness of me and a cheap server at uptime and antispam was particularly reliable. I'm aware of hardly any services that host mailing lists – for free or even for a fee! I chose Google Groups, after consulting with several people and finding no other solutions I found suitable. It functions both by email and by Web, thus (sort of) allowing it to be both our forum and our e-mail list, until such time as Google shuts down this service.[60] We made an IRC channel on Freenode, the most well-known open-source-focused IRC network. We also considered OFTC for IRC, which is technically/socially somewhat better and used by Debian. We chose Freenode mostly because we already had Freenode accounts and because sometimes even if one refers to "OFTC channel #lasercake", a listener won't notice and will go to the #lasercake channel on Freenode and be confused. Our website `http://www.lasercake.net/` is static and currently hosted by Amazon Web Services (S3, Cloudfront and Route 53).

I referred to the advice in Producing OSS [9] several times while trying to make these infrastructure choices. The advice was not specific enough for me to find any better platform choices than the ones we chose.[61] We do not have much technical lock-in to any of the platforms we chose. Version control with Git is trivial to move to another revision-control host. Google Groups might delete its archives, but we have copies of all list messages by email because we are on the list. GitHub's issue tracker, alas, does not have such portable data. We own the domain, so it's easy to switch web hosting as necessary.

Access to the basic project resources ought to be shared equally between at least Eli and me, in case one of us disappears (or goes on vacation, or is busy). On GitHub, we made a GitHub Organization that met this need. We haven't figured out how to convince

[60] Lately Google has been shutting down many services each year, and changing many others. If Google requires anything more than an e-mail address for one to join the list, for example, then it will no longer be suitable for use as an open-source project's mailing list.

[61] At the time; it may have been updated and improved by the time you read this.

all the services we use to let us share authority. Alas, each has its own methods. Sharing passwords is insufficient for sites that require legal-identity-based registration (e.g. Internet domain registrars), especially since we each have our own individual domains that should not be shared-access.

My knowledge may be slightly out of date: do open-source projects have Facebook pages? I haven't a clue! Social marketing principles suggest to make only pages/accounts if they will be successfully kept updated throughout a long period of time (e.g. years); dead pages are worse than no pages.

5.2 Social infrastructure

Community and etiquette: We notice that some gaming communities become angry or misogynistic. We don't wish to tolerate injustice; we state our principles. We have moderator powers if necessary, though we dislike censorship. We try to welcome the variety of people who would be wonderful at helping out: "We want your cool skills! The two of us could do this project on our own, but it will be more awesome if you help us out. There's a lot of different things that go into a big project like Lasercake – art, sustainable design, computer programming, geology, physics, sound design, gender studies, and many other things besides." We've been researching how people/demographics different from ourselves interact with games.

"Release early, release often".[62] This is a well-known phrase in free/open-source software communities. It is part of the famous essay "The Cathedral and the Bazaar" [34] that advocated developing software in the open ("bazaar" style). Delaying to release until everything is perfect tends to mean not releasing at all, or not for far too long. It doesn't mean releasing any random code. Eli led a push towards making Lasercake playable before adding all the features we want. This allowed us to release a game that is already moderately cool (though

[62] http://www.catb.org/esr/writings/homesteading/cathedral-bazaar/ar01s04.html

horribly incomplete).[63]

Releasing: This release, I took on the role of Release Manager, as e.g. projects like LLVM or Darcs have. This means being responsible for tracking the flow of revision control changes leading up to release, tracking release-critical bugs, deciding when to have release-candidates and when to make the final release. In our project it also means *creating* the released programs for each OS plus source archives. Eli pointed out that I was in fact a Release Engineer because so much code had to be written to do this first release. The social aspects of release management become more critical as the number of developers increases. Complex changes should not go in just before a release because they're relatively untested and thus likely to have serious bugs. Releases are intended to be somewhat more likely to be stable than random points in development; big changes can wait till after a release.

Decision-making: Projects have various structures, not mutually exclusive with each other. Some projects have a "BDFL" (Benevolent Dictator For Life), such as Python and Linux, who is typically the person who started the project and remains well-regarded. This gives an easy way to resolve unimportant (or important) disputes. Many projects have a group of "committers" who have the authority to add changes to the main code repository. These people often function as project leaders. They (hopefully) review contributions from people who are not committers. In any case, it is common for people who do more work on a project to have more influence in it. Darcs has a fairly conscious approach to this. Many projects just do it and it mostly works out. Whatever the final authority is like, it helps to have a clear mission statement and principles (as we attempt to do). This helps everyone be

[63] It's also interesting to note the transition *The Mythical Man-Month* notes from 'Program' to 'Programming Product' (p. 5, [3]). A 'program' only need run on the developer's own system; a 'programming product' is useful for many others. Indeed, just getting the program to compile, and run, and have basic things like an icon, and run on different computers than it was compiled on, for all of Mac and Windows and Linux (we run Linux), was two weeks of effort (which mostly won't need to be repeated). Then there's documentation. There's finding the inevitable many obscure bugs. Brooks' estimate is that a programming product takes three times as long to make as the similar mere program. My intuition says this is probably accurate. Of course, there are joys to be had in creating something shareable.

on the same page in the first place.

The Mythical Man-Month suggests that having a socially equal group of developers is not the most efficient way to develop software. Brooks has a vision of a 'surgical team' (p. 32) where each team member fills a specialized role, many people's role being a helper or helping the helpers. It resolves the problem of disputes by putting everyone into a hierarchy of authority. This may or may not be socially dubious. As regards artistic vision, it is usually wise for one person to be the undisputed lead.[64] I'm happy for Eli to be the artistic lead; it is more to Eli's temperament than mine. And I am usually the decision maker on technology platform issues. Regardless, brainstorming with each other is often useful, however, and we intentionally avoid decisions that the other person can't live with.

From another perspective, Eli and I match the tech-startup meme "co-founders".[65] It is certainly helpful that we've known each other a long time. It's probably necessary that we broadly have similar moral/ethical perspectives. It's super helpful that we don't have exactly the same social and technical backgrounds: each of our weaknesses is often filled in by the other one's strengths.

6 Tutoring the creation of a parser

Now we move away from discussion of Lasercake, and into an example of tutoring specific code. I have training in nondirective tutoring,[66] though I was not acting in any official tutor role here. In nondirective tutoring, tutors listen. Tutors ask questions or share thoughts; the goal is to help a tutee find out answers for themself. The idea is that any understanding that a tutee finds is much stronger if they find it themself than if they take instructions. More broadly, giving instructions teaches the tutee to be dependent on tutors, while helping

[64] My experience regarding this is in dance performance.

[65] A typical article: `http://venturehacks.com/articles/pick-cofounder` and typical startup-culture discussion about that article: `https://news.ycombinator.com/item?id=938320`.

[66] See e.g. *The Practical Tutor* by Emily Meyer and Louise Z. Smith [23].

the tutee be clever helps them help themself.

Techniques from Liz Lerman's Critical Response [22] process inform my feedback process. Lerman's process looks for meaning in creators'/tutees' creations[67] and puts the creator in control of what sort of feedback they'd like. It has a series of steps[68], but I find a few big ideas in it. Positive/negative judgments are rarely useful feedback (creatively or emotionally). Speaking is not harmless, but is an act of power; thus the creator gets to choose what feedback they'd like to hear and what feedback they don't. Also, mindful/nonjudgmental observation is really cool.

6.1 Parser!

Sam, my fellow Computer Science student, was writing a parser (see "What is a parser?", Fig. 6.2).

This parser was for a simple "logic language" (Sam's words) that she invented. It can say things like "true or (false and true)". Sam's evaluator tells you whether each line of logic

[67] Liz Lerman's field is dance, but the method is designed to work on any creative product. To a large degree, I believe that code is a creative work. It is written; there are many ways to do it; a large part of the work is done by staring dreamily into space. Code has rules, but so does dance: in dance, one can't teleport across the stage. One must use muscles (or other specific mechanisms that obey the laws of physics). On a larger scale, dance has narrative structure and code has function, class and module organization. Each of these has many ways to do badly and also many ways to do well. In either case, the underlying dance moves or code execution steps remain mostly intact, even while the piece as a whole suffers from a deficiency of sense.

Some Perl programmers blur the line between programming and art even more by writing executable poetry. `http://c2.com/cgi/wiki?PerlPoetry` Knuth, the famous algorithmist, saw programming as an art form in most every way: `http://paulgraham.com/knuth.html`. Brooks' *The Mythical Man-Month* compares programming to poetry and magic (p. 7-8): "[The programmer] build [their] castles in the air, from air, creating by exertion of the imagination." ... "The magic of myth and legend has come true in our time. One types the correct incantation on a keyboard, and a display screen comes to life, showing things that never were nor could be." ... yet "The computer resembles the magic of legend in this respect, too. If one character, one pause, of the incantation is not strictly in proper form, the magic doesn't work." What if there were more fancy in how we talked about programming every day?

[68] 1. "Statements of Meaning": specific things that the viewer noticed in the creators' work. 2. "Artist as Questioner": the creator asks things they desire to hear about, often specific but not necessarily. 3. "Neutral Questions from Responders": Responders ask questions about the work, aiming not to ask questions that judge through words or manner (hint: "why is your code so disorganized?" is judgmental; "how did you choose the organization of your code?" might not be). 4. "Permissioned Opinions": Responders ask if the creator would like to hear an opinion about a specific subject. The creator can happily say yes or no.

is true or false. Well, it does now; it was unfinished then. Sam, our professor Jim, and I, were in a room, debugging Sam's code.

I was listening, absorbing the context of the situation, some of which I'd missed. Sam and Jim had been corresponding on this a little before I'd entered the situation. One can wreak unintended havoc by being late and repeating arguments that have already been had.[69] *Listening for a while is pretty handy.*

Sam had a series of test cases. 'false' parsed fine, but even "true or (false and true)" was failing. The largest test case, which uses all Sam's language features, failed too:

```
p = true
q = false
(p and q) implies (p or q)
```

This last example, we all reckoned, should evaluate to $[true, false, true]$. The first two lines define p and q to be true and false respectively; a definition gives the whole line that truth value. Given those values of p and q, the last line is true.[70]

[69] Texts on consensus process, for example, make note of this.

[70] In fact, the last line would be true for all p and q. The language does not have quantifiers that could express this. Sam designed this language to be simple as a way to get started writing languages.

6.2 What is a parser?

Consider the program

true or (false and true)

Before analyzing its meaning, we **lex** and **parse** it. The lexer turns the string into a sequence of tokens (words or symbols):

true or (false and true) newline

The parser turns that sequence of tokens into an abstract syntax tree (AST):

```
        or
       /  \
   true    and
          /   \
      false    true
```

at which point the program can evaluate it.

$$lexer : String \rightarrow [Token]$$[71]

$$parser : [Token] \rightarrow AST$$

$$evaluate : AST \rightarrow Maybe\ [Bool]$$

6.3 Wherein We Debug Sam's Parser

I had not seen Sam's code before. Jim had read an earlier version of Sam's code. The earlier version was just as broken.

Sam told us about her code.

Sam split the task of parsing into two parts: a declarative grammar[72], and JavaScript code that uses one of those grammar descriptions to parse a series of tokens. One can think

[71] *String* means plain text. $[x]$ means list of x. "$name : argument \rightarrow result$" means name is a function that converts such arguments to results. (Inspired by Haskell.)

[72] Sam's grammar is represented in the implementation language, JavaScript, in a rigid, structured manner. This technique is sometimes called an EDSL (embedded domain-specific language).

of the code part in either of these mathematically equivalent ways:

$$\mathit{parse} : (\mathit{Grammar},\ [\mathit{Token}]) \to \mathit{AST}$$

$$\mathit{create\_parser} : \mathit{Grammar} \to ([\mathit{Token}] \to \mathit{AST})$$

Jim and Sam had read a book about parsing.

Jim wanted Sam to implement the depth-first search that they'd been intending. Jim was critiquing that Sam was just pushing around things in the grammar to see if it would work, rather than really understanding it.

I looked at the grammar and Sam's example test cases. I spied a mistake in Sam's grammar code. Because of Jim's critique, I just told Sam that there was a mistake in the grammar, not what it was. Sam asked for a hint, so I scrolled the editor window to include the relevant lines [shown below][73] and said that scrolling was my hint.

```
expression = unambiguous-expression  infix-operator
                        unambiguous-expression  newline
           | unambiguous-expression  newline

unambiguous-expression = literal
                       | variable
                       | ( expression )
```

That was enough for Sam to notice the mistake, so my obtuseness didn't hurt. In fact, it gave Sam a chance to build an understanding in her own mind. The mistake is that a parenthesized `expression` shouldn't require a newline before the end parenthesis. Instead, newlines after each line should be required somewhere else in the grammar. It was easy for Sam to find a fine place. In fact, Sam had had it there, she said, before some tinkering.

What are the risks of tinkering, then? What safeguards is it useful to have in place? I use version control and/or copies of code when tinkering something to try and understand it; this method is still rather messy. I considered saying something. Instead I used silence

[73] Shown in EBNF syntax rather than Sam's DSL, for simplicity. For example, *unambiguous-expression* can represent a literal ('true' or 'false'), a variable, or an expression with parentheses around it.

to let Sam learn from the frustrations. Sam already seemed to feel this frustration. Besides, this afternoon there was no room left in Sam's mind to handle extra pushiness.

As it happens, it was easy for me to spot the mistake because I've done lots of tinkering with parsing.[74] I often find myself being helpful in this way: I'm familiar with something and can be a set of eyes. I can see how someone's code relates to typical patterns. Then I can tell the person how it relates to these patterns (I like this better than telling the person "you're doing it wrong").

The afternoon went on in this vein for some time.

7 Conclusion

Collaboration is complicated! But it's not hard. A little bit of listening or doing the positive things you're good at goes a long way. Searching the web is a key tool for programmers regardless of skill. Having a supportive environment is really nice; there should be more of this. Whatever paths you choose, have fun!

8 A sample IRC real-time chat conversation I had while developing

This is a fairly typical IRC conversation log. The interactive real-time nature of the conversation lets us help each other debug confusing things more efficiently than e-mail would.

```
(2013-02-03 16:03:37) isaac: awake
(2013-02-03 16:03:42) eli: hi!
(2013-02-03 16:12:54) isaac: so, you know the various constants near the bottom of world_constants.hpp?
(2013-02-03 16:13:50) isaac: What dimensions are each of them supposed to have?!
(2013-02-03 16:14:09) isaac: (you there?)
```

[74] Tinkering I've done: Parsing Wesnoth's WML macro language in C++, and Haskell, and Haskell with Parsec (all top-down parsers); trying Boost.Spirit parser combinator library; modifying GHC's EBNF parser regarding doc-strings; etc.

(2013-02-03 16:15:22) *eli*: hmm
(2013-02-03 16:15:35) *eli*: starting at ?
(2013-02-03 16:15:57) **isaac**: min_convincing_speed
(2013-02-03 16:16:11) **isaac**: just to be sure
(2013-02-03 16:16:28) **isaac**: I assume that's proportional to m/s, gravity acceleration is proportional to m/s^2,
(2013-02-03 16:16:41) *eli*: oh dimensions
(2013-02-03 16:17:03) *eli*: friction is also a m/s^2
(2013-02-03 16:17:33) *eli*: pressure_per_depth_in_tile_heights must be what it says it is
(2013-02-03 16:18:11) *eli*: kg m^-2 s^-2 lol
(2013-02-03 16:20:17) *eli*: air resistance should be... m/s^2 / m^2/s^2 so just inverse meters?
(2013-02-03 16:20:44) **isaac**: tile_physics.cpp takes the sqrt of a variable called "pressure", which seems suspect for usual pressure units...
(2013-02-03 16:20:45) *eli*: looks like it's flipped from that so regular meters
(2013-02-03 16:20:55) *eli*: which means it'd be more resistance the lower the number is
(2013-02-03 16:21:02) *eli*: hmm
(2013-02-03 16:22:24) **isaac**: air resistance doesn't have units of m or m^-1 in real life!!
(2013-02-03 16:22:38) *eli*: it doesn't?
(2013-02-03 16:22:51) *eli*: isn't it acceleration proportional to the square of the speed?
(2013-02-03 16:23:01) **isaac**: hmm
(2013-02-03 16:23:31) **isaac**: looking things up
(2013-02-03 16:27:04) **isaac**: wikipedia doesn't have any *examples* for my questions! It says drag is dimensionless for high Reynolds Number
(2013-02-03 16:27:11) *eli*: supposed to be sqrt(2gh); pressure is (like) proportional to height
(2013-02-03 16:27:17) *eli*: (for the pressure stuff)
(2013-02-03 16:27:42) *eli*: so sqrt (2gp * some constant factor)
(2013-02-03 16:27:55) *eli*: which is, let's see
(2013-02-03 16:28:49) *eli*: m/(kg*s^2)
(2013-02-03 16:29:05) *eli*: no wait
(2013-02-03 16:29:13) *eli*: ms^2/kg
(2013-02-03 16:29:21) *eli*: m^2s^2/kg?
(2013-02-03 16:29:35) *eli*: yeah m^2s^2/kg
(2013-02-03 16:30:02) *eli*: multiply it with g and you get m/kg
(2013-02-03 16:30:20) *eli*: i dunno where I'm going with this
(2013-02-03 16:31:19) **isaac**: well, you had more to do with writing that code than me, and I don't want to unit-ize it the *wrong way*!
(2013-02-03 16:31:29) *eli*: long story short it's probably wrong
(2013-02-03 16:34:04) *eli*: pressure_per_depth_in_tile_heights, to be consistent with the use of 'pressure' in tile_physics.cpp, has to be... what, m^2/s^2?
(2013-02-03 16:34:52) **isaac**: I can't be sure because I don't know which numbers in tile_physics.cpp are actually supposed to be m versus not...
(2013-02-03 16:35:04) *eli*: but it looks like its signature makes it m^2/s^3

(2013-02-03 16:35:15) *eli*: yeah it's bad

(2013-02-03 16:35:16) **isaac**: it's like trying to do type inference on an incorrect, un-annotated Haskell program :P

(2013-02-03 16:35:32) *eli*: lol

(2013-02-03 16:36:25) **isaac**: (unrelated note: I read a paper about type inference of units as implemented in C# or F# (I forget which); it turned out to work pretty well)

(2013-02-03 16:38:06) *eli*: p = \rho g h

(2013-02-03 16:38:41) *eli*: \rho is density, kg m^-3

(2013-02-03 16:39:03) **isaac**: is that a physics statement?

(2013-02-03 16:39:06) *eli*: yes

(2013-02-03 16:39:08) **isaac**: good

(2013-02-03 16:39:15) *eli*: so h = ...

(2013-02-03 16:39:34) *eli*: p / (\rho g)

(2013-02-03 16:40:00) *eli*: here \rho is 1 g / cm^3

(2013-02-03 16:41:18) *eli*: or ... 1000 kg / m^3 ?

(2013-02-03 16:42:40) *eli*: h = p / (98000 kg / (m^2s^2))

(2013-02-03 16:42:40) *eli*: h = p m^2s^2 / (98000 kg)

(2013-02-03 16:42:57) *eli*: I propose we change pressure to be measured in pascals and then fix the rest of the units

(2013-02-03 16:43:17) *eli*: first let me check if the NUMBER is right

(2013-02-03 16:43:56) *eli*: (because if it's not... well, I guess we'll have to change it!)

(2013-02-03 16:44:47) **isaac**: complication: I want to check whether the compiler stinks at optimizing unit-ized code, which means I need unit-ized and non-unit-ized code with the same behaviour, which means I'm trying to fix the behaviour bugs in master and merge those fixes into the unitizing branch

(2013-02-03 16:45:32) *eli*: interesting

(2013-02-03 16:46:10) *eli*: p = \rho g h = 1000 kg/m^3 * 9.8 m/s^2 * h = 98000 kg/(m^2s^2) h; p/h = 98000 kg/(m^2s^2)

(2013-02-03 16:46:46) *eli*: or for h = tile_height = 2m

(2013-02-03 16:47:03) *eli*: p = 196000 kg/(m^1s^2)

(2013-02-03 16:47:24) *eli*: okay that's probably the correct value for what pressure_per_depth_in_tile_height SAYS it is

(2013-02-03 16:47:57) *eli*: but that's not what it really is

(2013-02-03 16:48:40) *eli*: really it's "gh"

(2013-02-03 16:48:52) *eli*: gh per tile height

(2013-02-03 16:48:57) *eli*: uh

(2013-02-03 16:49:17) *eli*: not per exactly

(2013-02-03 16:49:26) *eli*: gh when tile_height = 2 meters

(2013-02-03 16:49:45) *eli*: so 196 m^2/s^2

(2013-02-03 16:49:56) *eli*: which appears to be what it is, thank goodness

(2013-02-03 16:50:18) *eli*: Okay you could either

(2013-02-03 16:51:03) *eli*: set it to 196 m^2/s^2 (or rather gravity_acceleration_magnitude * tile_height)

(2013-02-03 16:51:06) *eli*: or

(2013-02-03 16:52:49) *eli*: fix the terminology in your master by changing it to 196000 kg/(m^1s^2) = gravity_acceleration_magnitude * tile_height * uhhhhh

(2013-02-03 16:53:29) *eli*: ... without units there's nothing actually incorrect because the volume units can be whatever you want

(2013-02-03 16:53:41) **isaac**: hmm

(2013-02-03 16:53:49) *eli*: so it really can be interpreted as sqrt(2 * pressure * some units)

(2013-02-03 16:54:21) *eli*: I should go to dinner soon

(2013-02-03 16:55:03) **isaac**: i'm lost as to where I should start unit-izing this

(2013-02-03 16:55:11) **isaac**: did you give me enough info?

(2013-02-03 16:55:12) *eli*: heh

(2013-02-03 16:55:15) *eli*: um

(2013-02-03 16:55:53) *eli*: idle_progress_reduction_rate is in m/s, max_object_speed_through_water is in m/s

(2013-02-03 16:56:09) *eli*: that meets the extent of the questions you specifically asked

(2013-02-03 16:56:19) **isaac**: what does "idle_progress_reduction_rate" mean?

(2013-02-03 16:56:46) *eli*: it is the rate at which tiles lose progress if they're not gaining progress in that direction, IIRC

(2013-02-03 16:56:51) **isaac**: the speed at which semi-displaced tiles move towards a stable state

(2013-02-03 16:56:52) **isaac**: ok

(2013-02-03 16:56:55) *eli*: yes

(2013-02-03 16:57:08) **isaac**: for some reason it doesn't use any sort of accelleration

(2013-02-03 16:57:26) *eli*: why would it?

(2013-02-03 16:57:55) **isaac**: uh because constant speeds as physical constants don't seem familiar from real life?

(2013-02-03 16:57:59) *eli*: heh

(2013-02-03 16:58:09) *eli*: the water physics are many kludges

(2013-02-03 16:58:14) *eli*: btw on an unrelated note

(2013-02-03 16:58:34) **isaac**: it could go away logarithmically... Anyway I'll keep m/s for that

(2013-02-03 16:58:50) *eli*: the exact shadows I'm working with can probably compute exact coverage/time over a period of time too

(2013-02-03 16:59:03) *eli*: like for solar panels to produce exactly the right amount of energy

(2013-02-03 16:59:07) **isaac**: that's nice

(2013-02-03 16:59:13) *eli*: Well, subject to rounding error of course.

(2013-02-03 16:59:22) *eli*: sorry coverage * time

(2013-02-03 16:59:28) **isaac**: which kind of obscuring objects do they support?

(2013-02-03 16:59:41) *eli*: currently tile sized things only

(2013-02-03 16:59:46) *eli*: hmm

(2013-02-03 17:00:32) *eli*: I don't currently know of a specific reason why it wouldn't support arbitrary axis-aligned bounding boxes.

(2013-02-03 17:01:07) *eli*: However I might need to do more complicated stuff to support arbitrary polyhedra.

(2013-02-03 17:02:17) *eli*: off to dinner unless you need me for something right now

(2013-02-03 17:03:05) **isaac**: ok

9 Glossary

audience – the reader/viewer/interpreter of a work, either intended or actual. Pieces of code often have as an audience both computers and other programmers.

bug – when a program is intended to do one thing and actually does another. A bug is a specific problem, such as "When I click Menu->Find in this program, it crashes instead of finding something". Most nontrivial programs have many bugs.

C – a programming language.

C++ – a programming language.

Carol Hendrickson – Anthropology professor at Marlboro; co-sponsor of this Plan of Concentration.

code – the stuff that a computer program is made of is called code, just as an essay is made of writing.

debugging – finding the cause of a bug and fixing it. Sometimes, searching for any bug, fixing it, then searching for another bug, repeatedly.

Eli Dupree – my sibling, who's two years younger than me, knows about the same languages I do and learned most of them at about the same age.

Elias Zeidan – fellow Computer Science student; started programming in Python while in college.

Fog Creek Software – a New York City–based software company at which I interned. It is well known among programmers for its founder's blog, `http://www.joelonsoftware.com/`.

FogBugz – the software product I worked on at Fog Creek Software.

four colour theorem – a famous mathematical theorem that says that any planar map (given a precise definition) can be colored with no two neighboring countries the same color using only four colors.

Free Software – Free/Libre/Open-Source software (sometimes FOSS or FLOSS) is software that meets a certain set of conditions: anyone who has a copy of the program must have the right to run it, read its source code, change it, and share their changes with others under the same terms. Richard Stallman is responsible for defining this and thus leading so much software to be licensed freely under consistent license terms.

git – a revision control system.

GNU – a brand for software created or overseen by Richard Stallman's Free Software Foundation.

graph – in mathematics, a network of "nodes" that has "edges" that connect pairs of nodes.

Haskell – a programming language.

Isaac Dupree – me; started programming in C and C++ about a decade ago from this writing, then learned Haskell, among many other languages.

JavaScript – a programming language.

Jim Mahoney – Marlboro's computer science professor; knows many programming languages; sponsor of this Plan of Concentration.

John Sheehy – writing professor at Marlboro; a former sponsor.

Kristin Horrigan – dance professor at Marlboro; my former advisor and sponsor.

language – see "programming language".

Lasercake – an open-source, open-world game about the environment, written in C++, being created by Eli and me: `http://www.lasercake.net/`.

mercurial (hg) – a revision control system.

open source – see Free Software.

Perl – a programming language

programming language – a formal way of structuring text in a computer to do something; there are many. Names of things in a program in any programming language are typically chosen from the writer's human language (e.g. English).

Python – a programming language

refactoring – rewriting code without changing what it does. This can make it easier to make substantive changes to a program, or to avoid introducing bugs by changing too many things (program structure and program effect) at the same time.

revision control system – a program that manages code changes. It lets one record each change to the code and that change's author, and lets collaborators easily share and merge each others' changes into a combined work.

Richard Stallman (rms) – pioneer of Free Software.

Ruby – a programming language.

Sam Auciello – a fellow Computer Science student; started programming in JavaScript 8 years ago; knows many languages; likes the language Ruby.

technical debt – "Technical debt" refers to the state of code that might work now, but is difficult to change because of how it's written. Experience and knowledge of good programming practices helps one write code that has less technical debt, but there's always some. It's called "debt" because you pay it back by refactoring the code to be more malleable before you can do substantive changes.

tutoring – helping someone with something they're trying to do. I practice non-directive tutoring. The philosophy of non-directive tutoring is to not tell people what to do, and to prefer letting statements be the tutee's words and judgment, because this helps tutees learn better. I am one of the writing tutor staff at Marlboro College; we practice non-directive tutoring. Informally I tutor people at computer programming.

version control system – see revision control system.

Part II

Prototyping a User Customization Language

10 Intro

In the simulation-game Lasercake that I am helping create, there are robots. Players will be able to control these robots by writing programs for them. So we must choose a language for controlling these simulated robots.

I created two prototypes. The first is online at `http://www.idupree.com/starplay/` and is written in JavaScript+HTML (with the UI using CoffeeScript, jQuery and Backbone.js). The second is in Haskell and has only a command-line interface. Both of their code is online at `https://github.com/idupree/Starplay`; the JavaScript+HTML. (It also appears in Appendix Part V.)

10.1 Criteria

We have goals for this language, some common and some uncommon:

Usability The programs should be easy to write and debug. The learning curve should be shallow. (Many existing languages have this goal.)[75]

Expressiveness The language should be as good for computation as a typical modern

[75] A major part of usability is not part of the language itself. A usable language has good tutorials, syntax hilighting, IDEs, REPLs, etc. Arduino, for example, is excellent (though its language is C++-based, lacks memory protection, and generally isn't what Lasercake wants). Writing a Lisp or a reimplementation of Lua would be better than making up our own syntax, because it would still have many of those documentation and tooling benefits. I personally like purely functional languages, but it seems that imperative programming is better known and easier to get into, so we should support mutation.

programming language. In my biased opinion, this means it must embed the lambda calculus (as most modern languages do). It need not be good for large-scale programs: Lasercake is a world of many robots with small programs, and with human supervision available for robots that get confused.

Determinism Given the same game conditions and same code, the language must produce the same result every time. It must be the same on all platforms as well, e.g. on 32-bit Windows and 64-bit Mac. This determinism makes it easier to debug Lasercake, and provides a variety of interesting yet rare benefits.[76] It is easy to achieve if you set out to create a deterministic language, but if you didn't (very few languages try; the pure parts of Haskell are the only one I know), nondeterministic elements are likely to creep in.

Serializability A robot's code execution state must be able to be saved and restored exactly. After all, there will be a "save game" feature and there will be networked gaming, which involves serializing game state to send over the wire.

Sandboxing A robot's code must not be able to affect anything but the robot's choices: not other robots, not the game world (unless they have psychokinesis), and certainly not the user's email or credit card information. We expect users to share robot code with each other (via saved games, networked games, and/or social sharing). We should not allow users to accidentally or intentionally cause denial-of-service or arbitrary-code-execution on each other's games or machines.

[76] A deterministic game simulation allows us, should we wish, to simulate the game in parallel on every user's computer in a multiplayer game. This might reduce perceived latency, allow players to catch each other cheating, or other interesting things. This allows us to let users "go back in time" and watch an accurate replay without recording every game moment or state change; a much smaller record of every user input would be sufficient. This determinism does not prevent randomness (we keep pseudorandom seeds explicitly as part of the game state).

Most languages do not meet the difficult categories (Determinism, Serializability, Sandboxing).

JavaScript implementations have Sandboxing but lack Determinism: they expose floating-point, random numbers, implementation-defined hash-table ordering, etc. These implementations are complex and not easily hackable. (They are complex partly in order to be fast). Replacing their number or object semantics would be significant work!

OS-level sandboxes are not good enough to deterministically protect the game from a robot program that gets into an infinite loop; see discussion of Chromium's sandbox on page 62.

Lua is designed to be easily hackable and embeddable. It has Sandboxing. It could have Determinism if we replaced its number type (trivial), replaced its hash-tables (doable), did something about its less-than operation comparing reference types (probably doable), and checked the complete Lua codebase for anything else we missed (doable). In order for it to have Sandboxing+Serializability, we need to be able to save its execution state. Pluto[77] is a decent[78] Lua serialization library, and Persistence[79] appears to be better (for this purpose).

11 JavaScript+HTML

In this code I explored what a user interface for controlling a large number of robots might look like. I took some inspiration from StarLogo[36]: the simulated agents are called "turtles" 🐢 and the tiles are called "patches" 🟩. One can also make StarPlay rules that apply to the whole world ◯ or that are run just once to initialize it ✳.

The simulation is controlled by a list of rules. There is a default initial set. The user can

[77] `http://luaforge.net/projects/pluto/`

[78] With limitations. "Tamed Pluto" makes its serialized format portable `http://luaos.net/pages/tamed-pluto.php`. It serializes closures by saving bytecode, even though malicious bytecode can crash Lua `http://lua-users.org/lists/lua-l/2010-01/msg01382.html`.

[79] `https://code.google.com/p/corsix-th/wiki/Persistance`. It claims only to support 'double'-compatible number types (probably adaptable), and can't persist coroutines thoroughly.

add, delete or change rules. Each rule is run every simulation frame once for each object it applies to, but only if its condition evaluates to true for that object on that frame.

The rule names do not make a difference unless you make a rule that references another rule. Therefore, the '+' button that starts a new rule puts a default name in. We could use a number and have a machine-like aesthetic. Instead, we choose a random inoffensive word that is not currently in use. This hopefully makes it friendly or amusing. The user is free to change the name to whatever they like. They need not hurry.

The language in the rules is either CoffeeScript or a Lisp-like syntax. I created the Lisp-like language implementation. It is somewhat okay. One day I concluded it would be a lot easier for me to try language-implementation things in Haskell, and stopped working on this web-based prototype.

The JavaScript+HTML language is implemented using symbolic evaluation of expressions. It substitutes variables too eagerly; this makes recursion impossible.

It does not explicitly represent the state of computation. It uses the JavaScript stack to implement the Lispy stack. JavaScript does not have continuations, threads, coroutines, or a serializable stack. Therefore, this implementation could never proceed partway through a computation, pause, save the game (to disk or a server), and then load the game without recomputing the work that's already been done. It has no way to persist the stack. This is a problem; we aim to efficiently and transparently allow robots to have long-running computations. The Haskell-based prototype does not have this problem.

12 Haskell

The prototype written in Haskell is just an interpreter; it has no game elements. '`runghc Lispy.hs test.scm`'[80] interprets the contents of 'test.scm' as an expression, evaluates it,

[80] On the command-line. Tested with GHC 7.6.1. As far as I know, this program should work with GHC 6.12 or later. To make it run faster, GHC can compile it: `ghc -O2 Lispy.hs -o LispyExe; ./LispyExe`

and (if its evaluation did not exceed the number of computation steps Lispy.hs allows) prints the result.

The syntax resembles Scheme. The built-in operators are ‘lambda’, ‘if’, ‘letrec’, ‘begin’, and all the functions and constants listed in Lispy.Types.builtinNames (in Lispy/Types.hs). Unlike Scheme, functions have a fixed number of arguments (‘+’ takes exactly two arguments, no more). Like Scheme and Lua, ‘or’ and ‘and’ return their first or second argument. Every value is truthy except for ‘nil’. ‘(not nil)’ is ‘true’. ‘(if condition then else)’ requires both a then and an else branch. ‘letrec’ is local variable binding; it allows recursion and is as powerful as Scheme’s ‘let∗’ and ‘letrec’ combined.

The table builtins are inspired by Lua. In Lua, there is a single composite data type called the ‘table’. It is an associative array implemented by (more or less) a hash table. Sequences are represented by a table with keys 1, 2, 3 ... n.

In this prototype, tables are purely-functional, key-ordered associative containers. They are implemented as self-balancing binary search trees. Tables-as-sequences start at 0 by default rather than 1, though it doesn’t make a difference: sequence iteration uses the keys’ ordering and ignores actual key values unless you look at them.

Using names like ‘lambda’ and ‘letrec’ is unsuitable for a friendly robot language, but renaming them will be trivial. Even switching the concrete syntax is relatively trivial: I deliberately avoided Lisp-specific language concepts such as macros. For prototyping, I’m sticking with programming-language-theory names and trivial Lisp-like syntax.

12.1 Evaluation

The Haskell implementation code is purely functional. No input, output or mutation is done except for the `main` function reading the code from disk and logging messages. This is not a primary goal. However, a purely functional implementation means it is trivial to keep

`test.scm`

snapshots of earlier evaluation states. This makes it easy to time-travel backwards to an earlier evaluation state. That ability might only be useful for implementing time travel, but it's nice to know it can be done.

The Haskell code does not use the Haskell stack to implement the Lispy stack. Instead, it uses an explicit data structure for the stack. This allows it to single-step[81] the execution of the Lispy code and pause whenever it likes. The code's stack data structure is similar to a cons list, with the top of the stack at the head of the list.

The Haskell code compiles Lispy expressions to bytecode that is then interpreted. We retain source-level information by storing, alongside each bytecode, a reference to the source code element that created it (AST[82] node, text, and line/column). This is enough information to serialize the evaluation-state in whatever intuitive format we desire. Things are a bit complicated when a single AST node corresponds to several bytecodes. Alas, some syntax is intrinsically complicated (e.g. 'if') and cannot cleanly be implemented with just one bytecode. 'If' generally needs both a conditional go-to and an unconditional go-to instruction.[83]

We make sure only to provide built-in functions that have O(1) or O(log n) worst-case running time.[84] Obviously this is possible because computer instruction sets contain conceptually constant-time actions. We offer higher-level builtins, however.[85] O(1) and O(log n)

[81] As in single-stepping a program in a debugger: executing one near-constant-time instruction on each step.

[82] Abstract syntax tree

[83] And other syntax we might create, like 'while' or 'for', has other similar complications. Bytecode is no worse than AST interpretation for this. In fact, our bytecode is isomorphic to each AST node containing a sequence of states ($\approx$ bytecodes) that it can be in, each along with a "next AST node state to evaluate" reference.

It is also possible that symbolic evaluation could work, given a mechanism to permit recursion, but bytecode seems to be adequate and effective. Symbolic evaluation has additional complications like maintaining correct lexical scope when substituting a function into another function.

[84] $\log_2 n$ (with n being memory usage) on present computers cannot be worse than a constant factor of 30 or 40, and less if we limit a robot-program's total memory usage. Thus, we can treat O(log n) as a constant without risk here. Furthermore, high memory usage alone tends to degrade performance due to the cache hierarchy, often at a rate similar or greater than O(log n). At high enough memory usage, swap space becomes part of the cache hierarchy, which is pretty much doom (assuming spinning hard disks). So we need to limit memory usage, but O(log n) time usage is fine.

[85] One reason is that implementing data structures in an interpreted language is slower than using compiled,

builtins are sufficient, with some cleverness, to implement any common operation. For example, a list-sorting builtin would be O(n log n), but we can provide an ordered-set data structure that has O(log n) insertion. With n insertions, user code can create an ordered set in O(n log n), then convert it in O(n) to a (sorted) list. This sorting method uses only O(log n) or faster builtins, and does not require implementing clever algorithms in the Lispy language.[86]

Currently, a single operation can take O(length of the program text) if the program consists of a function with a zillion arguments (calling it will take O(number of arguments)), or with a zillion variables in its closure (creating the closure will take O(number of variables)). This is too high. We could modify these operations to act one-item-at-a-time, or we could simply ban functions and closures with more than, say, 100 arguments or closed-over variables.

12.2 Side effects

The implemented Lispy language itself does currently not have mutation, but mutation will be easy to implement while keeping the Haskell purely functional. The interpreter already contains a concept of "pending values" to implement 'letrec'.[87] When a lambda expression is evaluated, a closure is created. Normally closures are plain old values that can be included in another function's closure at will. If, however, a new closure recursively references itself (or references another not-yet-created function), the interpreter generates a new serial number and stores `PendingValue <serial number>` in the new closure, rather

optimized C++ or Haskell or JavaScript data structures.

[86] Spending time reimplementing well-known structures and algorithms in an obscure-by-definition scripting language seems wasteful.

[87] This implementation of letrec using pending values is not strictly necessary. I believe it is possible to implement letrec purely in the compiler by translating letrec to a clever use of the Y combinator. An intriguing possibility.

than try to store itself in itself.[88] When the closure is finished being evaluated, it stores the closure's completed value in an interpretation-state-global map from serial numbers to values (named `lsPendingValues`[89]).

When implementing mutable state, PendingValue can be renamed to GlobalValue or ModifiableReference[90], and builtin functions can be added that have a side-effect of updating the global map.

(In Haskell-land, that 'side effect' actually means returning a new version of the state that has the new value in it. This is made possible by balanced trees that can have an updated copy created in O(log n). I use the Haskell standard library Data.Map, a purely functional self-balancing binary search tree.[91] It has the same asymptotic performance as a mutable self-balancing binary search tree.)

References to mutable state, when shown to the user, should look friendly. It is trickier to give a complete description of a reference-to-mutable-state than for an immutable value. Immutable structures need not distinguish between contained value and identity, but mutable structures must.[92] We should be able to show the user the complete state of the heap. (These

[88] Because of Haskell's lazy evaluation, it might be possible to literally store itself in itself. I didn't want to rely on lazy evaluation, however. The data loops this lazy evaluation would create are difficult to tell apart from an infinite data structure during e.g. serialization.

[89] The 'ls' prefix stands for LispyState. Such prefixes are a convention for Haskell record fields because Haskell field names are scoped as top-level functions.

[90] The idea is similar to a Haskell IORef, or an ordinary reference or pointer in most imperative languages.

[91] The Data.Map documentation cites "Stephen Adams, 'Efficient sets: a balancing act', Journal of Functional Programming 3(4):553-562, October 1993, `http://www.swiss.ai.mit.edu/~adams/BB/`" and "J. Nievergelt and E.M. Reingold, 'Binary search trees of bounded balance', SIAM journal of computing 2(1), March 1973." For further implementation techniques, see Okasaki's *Purely Functional Data Structures* [28].

Such structures are uncommon in non-functional languages, even though they have an ability (efficient creation/sharing of related versions) that the mutable versions do not. They do, however, exist. LLVM (C++ code) has implementations called ImmutableSet and ImmutableMap, described on `http://llvm.org/docs/ProgrammersManual.html#picking-the-right-data-structure-for-a-task`.

[92] Mutable references can be generated at runtime, bound into closures, put into maps, and so forth. Neither the code line that creates them, nor the value they currently contain, is enough to tell which box they reference. For example, in Python, `[2]` and `[2]` have the same value but not the same identity. You can change the second list's contained value to "3" without changing the first list.

In contrast, in Haskell, there is no way to tell whether two equal (immutable) lists are in the same location in memory (aside from an unreliable GHC primitive useful only for optimization).

are small robot programs, so this shouldn't be too overwhelming.) If we do not do something clever, we'll have to use the serial-number to identify which identity a mutable reference has. Instead, we can use random inoffensive words as the HTML+JavaScript code does, and give each serial-number a friendly random name that isn't used elsewhere in the code.

12.3 Limiting memory use

This code does not currently limit Lispy code's memory use, but it must, because unlimited memory use can cause people's RAM to start swapping to disk and effectively freeze their computer. One way to do this is, every $\frac{K}{B}$ bytecode instructions (for any memory limit K, and B equal to the maximum memory a single instruction can allocate), check whether the program is using more than K memory. Check this by traversing every piece of data in its LispyState. If it is using more than K memory, do not evaluate it further (this is a deterministic memory-overflow error). This method limits maximum memory to 2K at a mere constant-factor time cost.[93]

But is the precondition true? Is there a maximum amount of memory a single instruction can allocate? Each initialized byte that's allocated takes an amount of time to initialize, so in N time one can only allocate O(N) memory. However, we have a purely functional map data structure. Updating it takes O(log n), but if we keep the old and new version, a naïve traversal will take 2n time to traverse the shared data twice. In the worst case we face O(n^2 / log n) if code repeatedly modifies the same map while keeping the old copies around.

The traversal code could short-circuit and report too much memory usage once it has traversed K memory. Or it could detect and discount for shared memory usage, though I am dubious about the wisdom of doing that. A serialized version of the values is unlikely to maintain this sharing.

[93] $\frac{K}{B}$ instructions cause traversing K memory. The time overhead per instruction is $\frac{\mathrm{O(K)}}{\mathrm{O(K/B)}} = \mathrm{O(B)}$. O(1) was already spent on executing that instruction. B is a smallish constant.

It is also undesirable to enumerate the entire memory in one large-time-cost, uninterruptible operation. It can be made incremental. Perhaps, instead, we can track the current logical memory usage of the program by updating a number whenever we allocate or free anything. This is tricky because freeing is typically done by garbage-collection. Perhaps we can use an incremental garbage-collector that we instrument to keep track of total allocated.[94]

We may have to integrate with such a garbage-collector for other reasons. The Haskell code currently has an issue where pending values (`lsPendingValues`; see subsection 12.2) are never erased, even if all references to them are erased. This issue becomes more serious once the pending-values implementation is used for mutable state as well. This requires garbage-collection to resolve, because a pending value may refer to itself (in fact, it is very likely to be a recursive function). The code cannot easily piggyback on the garbage-collection of the host language (e.g. Haskell) for this.

12.4 Strings

We don't implement strings, and they are not much use for a small fictional-robot language. For object-oriented programming, we can have a 'symbol' type, useful in map keys. Entire robot-programs are compiled at once, so we can simply catalog every symbol used in the code, sort them, and represent each symbol string as an int.

If string manipulation proves important, we can represent them as a sequence of characters.

[94] For example, Lua has a simple incremental mark-sweep collector, as noted in the manual:
Lua 5.1: `http://www.lua.org/manual/5.1/manual.html#2.10`
Lua 5.2: `http://www.lua.org/manual/5.2/manual.html#2.5`

12.5 Numbers

Numeric operations must be constant-time. This rules out using a bignum library. Numeric operations must be deterministic across platforms. This rules out floating-point.[95] Currently we use a 64-bit signed two's complement integer. We can do moderately better at making a user-friendly numeric type, but it will never be perfect.

Because Lasercake is about science, it uses units like meters and seconds and watts. Robots are quite likely to use units: "go 4 meters", "wait 2 seconds", "set the laserbeam power to 7 watts". In Lasercake's C++ code, we use SI units in the type system[96] to detect errors at compile-time if we combine units in a buggy way (for example, adding watts and seconds doesn't make sense). The robot language is dynamically typed, so tracking of a number's units (if any) would happen at runtime. Units are enormously helpful, so we plan to implement this. For example, you'd write (wait 2s) rather than (wait 2). When the game displayed a value that had units it would display both the number and its units. This is complex because of compound units (m/s), nicknames for units ($\mathrm{W} = \mathrm{kg}\cdot\mathrm{m}^2/\mathrm{s}^3$), and SI prefixes (milli) and the imperfect precision of the number representation. If the user multiplies meters by kilometers, that is certainly a valid unit of area, but SI does not provide a way to write it. If ? were an SI prefix in ?m^2, ?'s value would have to be $\sqrt{1000}$, which is not even a rational number. Doing something imperfect here is still much better than not having units at all.

[95] IEEE754 is now prevalent enough that, with the right compile options, we can probably cause floating-point to be deterministic. But then what if someone compiles with -ffast-math, or (on x86 where we have to avoid x87 excess precision) disables our -mfpmath=sse or -ffloat-store or -fexcess-precision=standard? What if any of the implementation-defined behaviour affects us? (See e.g. `http://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/Floating-point-implementation.html`.) Floating point also has pitfalls such as equality often being a bad idea on it; it is no perfect thing to strive for.

[96] Implemented in units.hpp in Lasercake's code.

12.6 Serialization

The Haskell implementation could trivially save and restore the evaluation state as data.[97] It is just a non-cyclic, immutable bunch of data.

This is not entirely satisfactory. It would be nice to be able to save the evaluation state in a way that doesn't rely on the specific bytecode, to the extent possible. This implementation's interpreter is safe: even if a bytecode or stack are loaded that cannot be created by a source program, the interpreter will not crash or hang. Lua bytecode, which we may ultimately wish to use, is not safe. Untrusted bytecode is unsafe even when untrusted Lua source is fine to compile and run.[98]

Also, suppose we can save the stack state with reference to the AST rather than the bytecode. It is plausible that we can use tree-diffing on ASTs to allow resuming a computation on a slightly different AST than it was saved from. This could allow runtime modification of robot programs without having to start them over from the beginning.

Every bytecode instruction has a reference to the AST node that generated it. AST nodes are numbered and indexed in many ways: we can serialize whatever location/path information is best. Closures are easy: they point to the body or root of their lambda AST. Stack frames are trickier because they can point to any bytecode at all. They can be in the partway through evaluating an 'if'. The serialized format needs to specify, for each special form, where in the special form's evaluation it is. This is not hard, but needs specific consideration. I have only looked a little at Lua bytecode, but I suspect this would not be too hard to retrofit.

[97] It does not currently implement this, but it's as simple as (in Haskell) deriving `Show` and `Read` on every data type or (in any language) using serialization libraries.

[98] `http://lua.2524044.n2.nabble.com/Bytecode-Safe-or-not-luac-manual-td6946200.html`

13 The suitability of modifying Lua to meet our needs

Lua is a small piece of code, designed to be customized.

The number representation is easy to change: it is a typedef in the Lua source code.[99] The string handling parts of Lua can be removed, along with any other features we do not want (it is important to remove features to prevent any O(n) runtime behavior). The table implementation can be replaced; worst-case O(n) lookup is unsuitable. The comparison operators can be changed to compare mutable-object-references deterministically. The parser can be tweaked to allow dimensional units. The bytecodes can have AST references added to them. The garbage collector can be instrumented to keep track of allocated quantity and to react in a predictable way when the amount of allocated data is too high. The interpreter loop can count instructions and exit after a certain number.

This is a lot of tweaks, but in return we get a decent incremental garbage collector, a fairly fast interpreter, and a well-tested framework. If we keep the basic Lua syntax and document our differences from Lua itself, then we get a language that is familiar to many people (especially video-game people) and has high-quality documentation already in existence.

14 Related Work

Once upon a time I played RoboWar[100], a simulation where two robots in a 2D arena move and shoot and try to destroy the other one. Players program robots in a custom stack-based Reverse Polish notation language. In each frame, a robot can execute between 5 and 50 instructions (depending what hardware one chooses for the robot). In addition, robot programs can set 'interrupts' such as: when a missile is nearby, jump to handleMissile and

[99] LuaJIT is unsuitable because, among other reasons, it only supports 'double' as numeric type. (I can't find my source for this information, but it makes sense that it'd be hard to customize: LuaJIT works by creating machine code for every operation.)

[100] `http://robowar.sourceforge.net/`

put the previous instruction-pointer location on the stack.

StarLogo[36] simulates massively parallel worlds on a small 2D world. Stationary ‘patches’ exist in a grid; ‘turtles’ move around on top of that grid. It uses a programming language in the Logo family.

In Majesty: The Fantasy Kingdom Sim,[101] players can neither control their heroes directly nor write programs for them. Each hero has a personality: a small program built into the game. The personalities have states like “Exploring”, “Wandering”, “Fighting”, “Thinking”, “Buying a healing potion” and “Fleeing in terror”. The player can see this state by selecting the hero. This brings flavor to the game. I’d love if our robot programs can easily show such text as well. Majesty also has sound effects and animations for certain state-of-mind changes. This may be harder to emulate when we allow arbitrary programs. It may be worthwhile to allow players to explicitly specify audio/visual effects, because they are very cool. They play into a decoration-and-emotions aesthetic that the otherwise rather industrial nature of Lasercake discourages.[102]

Chromium[103] runs untrusted scripts from the Web, so it has multiple layers of defense. We should definitely use the OS-level sandboxing techniques it documents[104] for sandboxing the simulation code. On OS X, this uses BSD sandbox(); on Linux, seccomp filters;[105] on Windows, a combination of techniques. (These OS-level sandboxes are not suitable for causing a robot-program to fail after a deterministic number of robot-instructions or robot-memory-allocation. CPU limits almost certainly will not cut off the program at the exact

[101] `https://en.wikipedia.org/wiki/Majesty:_The_Fantasy_Kingdom_Sim`, homepage `http://www.majestyquest.com/`

[102] For example, [11] shows some ways that games are likely to engage people who don’t like male-dominated games. (This article is written about young girls specifically, but its notes are more broadly applicable.)

[103] Chromium is the open-source software that forms most of the Google Chrome web browser.

[104] `http://dev.chromium.org/developers/design-documents/sandbox`

[105] Seccomp filters are new as of Linux 3.5 and have been backported into Ubuntu 12.04. They re-purpose the Berkeley Packet Filter (BPF) interface to filter system calls. There is a helper library for seccomp filters at `http://sourceforge.net/projects/libseccomp/`, with an introduction at `https://lwn.net/Articles/494252/`. The original seccomp was awkward to use; we could support this older interface, but it might not be worth the effort.

same point cross-platform.[106] Memory limits might, if we choose our memory allocator carefully. In any case, one OS process for each of tens of thousands of robots might be too much.)

It would be neat to let players share their robot programs using GitHub; this may or may not be feasible.

My HTML prototype pays attention to the user's rule changes near-instantaneously.[107] This instantaneousness was partly inspired by Chris Granger's article "Connecting to your Creation".[108]

14.1 Visual and textual editing

There are many languages and environments aimed at visual rather than textual manipulation of programs.[109] Some of them use code blocks; these are visually arranged similar to textual languages but are drag-and-drop. These have holes to be filled in for e.g. the contents of an 'if' or the arguments of a function. Some mimic flowcharts[110], such as Pure Data[111] or ISADORA[112].

However, even flowchart-like applications like hardware description languages are often text-based, e.g. Verilog and VHDL. There is something effective about text.

One can paste text into forums. One can search for it on the Web. One can version-

[106] I suspect they're not even entirely deterministic for the same binary on the same OS, kernel version, and CPU. I haven't investigated; it's likely that some platform offers that kind of determinism. That would still require everyone using the same binary, not their version compiled with their own compiler flags. Such a requirement not to recompile is contrary to Lasercake's free/libre philosophy.

[107] Actually, it updates as soon as you de-focus the text field to show that you are done editing it. This should be improved somehow. Updating on every key-press would lead to a lot of broken intermediate states. One approach might be to visually make a field that's being edited be very much more a text-box. Testing with actual users is an important part of choosing a way.

[108] `http://www.chris-granger.com/2012/02/26/connecting-to-your-creation/`

[109] `https://en.wikipedia.org/wiki/Visual_programming_language`

[110] `https://en.wikipedia.org/wiki/Flow-based_programming`

[111] `https://en.wikipedia.org/wiki/Pure_Data`; homepage `http://puredata.info/` which was down at the time of this writing.

[112] `http://troikatronix.com/` – a media manipulation tool for live performances

control it. One can be confident that there's no hidden content making things work.[113]

Some languages can be edited visually or textually.[114] A visual editing might not preserve the whitespace layout in the text. At least one popular language, Go[115], contains a re-indenting tool `gofmt` that is socially intended to be used everywhere.[116] If preserving whitespace isn't expected, then it's easier to have a visual editor. Go achieves this without indentation being syntactically significant like Python or Haskell.

14.2 Control flow

Eli and I have had many discussions about robot languages. One discussion yielded a couple novel features we may use.

We aim for a learning curve that doesn't force the player to make any giant jumps in understanding. All robots will be manually controllable by the player; the player can play manually with just one robot if they like. The player can define robot plans (programs). For example, a program could do something like "pick up a tile of rubble, go to the rubble pile, dump it in the pile, go back to the starting point". The player could use this even with one robot. The program would be like a "macro" in some games. There'd be an easy way to invoke programs manually. The robot would be like a cyborg: part human, part machine control.

The player can also do something similar with multiple robots. A possible robot instruction is "Ask for the player's input", which gives the player a notification; the robot will sit and wait. This lets the player write simple programs that ask for help when something confusing happens. (Robots could also have non-blocking notifications just to let the player

[113] Except for trailing spaces, trailing newlines, non-ASCII Unicode characters, CRLF vs. LF, tabs vs. spaces, etc. Text isn't *actually* simple. It's probably simpler than data representations that contain text *in addition to* a larger non-textual structure, though.

[114] I believe; if this is not true, it clearly can be and should be true.

[115] `http://golang.org/`, created by Google.

[116] `http://golang.org/doc/effective_go.html#formatting`

know their progress.)

There may also be a block “until” syntax. It works somewhat like exceptions, except that the try/catch-ish ‘until’ block decides when the stack is unwound, rather than a ‘throw’ expression. The expression in the ‘until’ would be evaluated often enough to make this structure effective. As a silly pseudocode example,

```
until (in free fall) {
  loop {
    until (targeted rock turns to rubble) {
      # Continuous actions last forever unless interrupted by an 'until'.
      fire laser
    }
    while (not pointing at a rock) {
      aim in a random direction
    }
  }
}
# Do something about the fact that we're falling possibly to our doom.
# Grow wings, perhaps?
```

Pros:

- This has fewer concurrency pitfalls than multithreading.
- This gives a clear resolution to race conditions in interrupt-based event handling. If one bad thing happens, then another, which one wins? With this ’until’ model, it is clear: the most-outer-scoped one wins.
- Unlike an evented model[117], this allows long-running computations to be interrupted just as easily as anything else. So it’s something worth trying.
- A language with short-running action primitives and no event handling must run continuously, wasting more host-computer CPU time than necessary.

[117] https://en.wikipedia.org/wiki/Event-driven_programming

Cons:

- 'Until' in the above pseudocode looks like 'while not', but it does not have those semantics. We should make the syntax clearly not mean that.
- We should check if 'until' is a word in plenty of human languages. I recall a European commenter on the Haskell `unless` function (which is equivalent to `when` ∘ `not`) noting that their native language does not have a word specially for 'unless'.
- 'Until' requires interruption-safe code. If the code writes two pieces of data in a row that should stay in correspondence, and is interrupted between the two writes, this is bad. Haskell has similar interruptions as "asynchronous exceptions". In purely functional code, interruption is harmless. For code with side-effects (for Haskell, code in `IO`), there is syntax to make blocks of code uninterruptible (thus atomic).[118]

14.3 Types

Many modern games use Lua as their scripting language, a dynamically typed language. Second Life uses its own Linden Scripting Language that, unlike most scripting languages, has static types.[119]

I contemplated a hybrid static/dynamic approach. Static analysis could attempt to find errors. Code that it can prove type-incorrect would be marked in red; code that it can prove correct can be green or unmarked; code that it cannot prove correct or incorrect would be yellow. As I recall, Epigram, an experimental dependently-typed functional language with an Emacs-based development environment, uses a similar scheme.[120]

[118] These are named `block` or now `mask`. In Haskell, they are not special syntax but rather functions `IO` α → `IO` α mapping one computation to another. `block` and `unblock` are slightly easier to understand than `mask`, but deprecated because their API isn't quite as modular as it could be. Documentation:
`http://www.haskell.org/ghc/docs/7.6.2/html/libraries/base/Control-Exception.html#v:mask`

[119] `http://wiki.secondlife.com/wiki/Category:LSL_Types`

[120] `https://en.wikipedia.org/wiki/Epigram_%28programming_language%29`. Its website `http://www.e-pig.org/` used to work.

C++-style type inference goes some distance: the type of an expression is determined by the types of its subexpressions. (In C++11 you can take advantage of this by listing the type of initialized local variables as 'auto'.) Hindley-Milner-style type inference [25] goes further by letting the algorithm infer argument types from result types when the result is used in a way that constrains its type.

There's a possible type-soundness pitfall with applying Hindley-Milner inference to languages with mutation. Part II of [43] addresses this.

Andrew Kennedy has done work on dimensional units in the type systems of the functional, Hindley-Milner-type-inference-based languages ML [18] and F♯ [19]. Dimensions appear to integrate into the Hindley-Milner regime rather smoothly.

14.4 Speed

It is desirable that the same language used for robots can be used for defining the randomly generated worlds that Lasercake has. Currently world-generators are defined in C++ and well-optimized and still a speed issue. We are working on better algorithms, but it is likely that world generation will be more speed-critical than robot programs. It also has a slightly different security profile. A malicious worldgen function can obviously freeze the simulation anytime it likes. The simulation absolutely depends on being able to find out about the contents of anywhere in the world at any time. The worldgen could just get into an infinite (or extremely long duration) loop when asked about a location. Therefore, it is pointless to make the worldgen interruptible or CPU/instruction-limited. It might still be desirable to limit its memory usage to protect the player's system, but OS memory limits are fine for that; the limit can be extremely high. Worldgens need not be as easy to create as robot-programs (though it would be awesome if they were that easy).

For speed, we could use LLVM as a backend to compile robot languages to. LLVM is in C++, just like Lasercake, but it is a large library. It would require care to avoid letting user scripts run code that gains control over the Lasercake process. It is not as easy/efficient to compile dynamically-typed languages to LLVM bitcode as it is for statically typed languages, but it's worth something.

Division CPU instructions are slow; [26] shows how to transform divisions by a constant to multiplications. Libdivide is a free/libre/open-source library implementation of this.[121]

[121] http://libdivide.com/

Part III

Campus Infrastructure and People

15 A disaster story

On December 11 to 12, 2008, an ice storm ravaged New England.[122] It was the last week of the Fall term at Marlboro College. I was a student. The storm hit. Trees fell, roads were blocked and power was out across much of New England. A couple days passed. At Marlboro, power was still nowhere in sight, the road to the outside world was barely passable, and winter loomed. Ellen McCulloch-Lovell, the college president, helped evacuate the campus and helped students get to their families for winter break.[123]

I was able to evacuate on the same day that Ellen asked us to get home if we possibly could. I called my family and they drove from eastern Mass. to Marlboro, VT on the most major highways (even many of the minor highways were impassable, two days after the storm). But how did I call my family?

It was moderately hard to find a way to call my family. My dorm's landline was dead. Cell phones, which barely get reception at the best of times in Marlboro, Vermont, mostly did not work, and the ones that did were in high demand. The college's Internet had been down ever since the storm. Ben Lieberson tried the pay phone in the Campus Center. It had a dial tone. Paying the pay phone with quarters did not work, because it was physically full of quarters. Ben tried a phone card, and it worked. Ben graciously let me use his card to quickly call my family as well (the card was low on funds). And that is how I was able to contact my family.

[122] Some details of this story may be off, because I am relying on four-year-old memories. Julie Powers-Boyle's Plan of Concentration [33] documents meteorological and ecological effects of the storm.

[123] Exams were not a priority, but they worked out alright. Students took them at home in the coming weeks, as their and professors' electricity, internet and well-being permitted.

Thankfully, I indeed had a family. Professors and other community members sheltered students in Brattleboro for a few days. The college community's social vulnerability was much less than many folks in the area. We know and care about each other. For the most part, we have durable houses. Many of us are not desperately poor.[124]

16 Social geography

Marlboro College is a small, 300-student undergraduate institution located near the top of a mountain in Marlboro, VT, USA. Marlboro, VT is home to about 1000 people. The nearest small city is Brattleboro to our east, with about 12,000 residents. Marlboro College, in addition to its main undergraduate campus, has a small graduate school located in downtown Brattleboro which will play a minor role in this study.

As an institution, we have connections with some of our peer institutions. We play soccer with Bennington, SIT and Landmark (all nearby institutions with less than 1000 students each). Our college library offers van trips to the much larger UMass Amherst library twice a semester. Our World Studies Program has a formal relationship with SIT; occasionally a Marlboro student can take an SIT class. The Marlboro College Graduate Center operates mostly independently from the main college, but sometimes undergrad Marlboro students take a class there. Bennington and Hampshire are the colleges that Marlboro students also applied to and therefore make fun of.

Most Marlboro students live on campus. Most of the rest rent in Marlboro or Brattleboro. Most Marlboro staff and faculty live in Marlboro, Brattleboro, or neighboring towns. There is free public transit from downtown Brattleboro to campus, paid for by the college: college-run vans and stops by the MOOver bus.[125] Some commuters take this transit, some drive, and some drive sometimes and take transit sometimes.

[124] Student loans make students desperately poor after college but not during.

[125] The MOOver is mostly aimed at tourism, but the college is near its route and pays for it to stop here.

17 My research

I interviewed several Marlboro College staff and a student involved in emergency response. I asked what it felt like to be amidst Hurricane Irene, the tropical storm that flooded Vermont and neighboring states on Monday, August 29, 2011. I asked about Marlboro College–related infrastructure and disaster response. Exact questions varied by the interviewee's knowledge area and often began open-ended. I interviewed Elias Zeidan, student and (at the time) college Fire Chief; Jeremy, Director of Academic Support Services; KP, Master Electrician for the college; Ken Schneck, Dean of Students, host of *This Show Is So Gay*,[126] and Brattleboro Selectboard member; John Baker, Network Administrator; Dan Cotter, Director of Plant & Operations, who used to be the Electrician; and Philip Johansson, Senior Writer & Editor for college publications. I've also corresponded with Allison Turner, Science Laboratory Coordinator and volunteer firefighter.

KP and Dan Cotter told me about electrical power for campus. Allison Turner told me about emergency-response and communication methods and electrical power, while warning that she might be wrong about any of it. Dan Cotter and John Baker told me about college Internet networks. Elias Zeidan, Ken Schneck, Dan Cotter, and KP told me about emergency-response, communication and coordination. Philip Johansson and Jeremy told me about their experiences as staff whose positions don't directly touch on any of these issues.

I was away from Marlboro College at that time. During Hurricane Irene and the following Fall, I was living in Massachusetts. I lost power for six days. I heard what was going in in Vermont through friends on Facebook.[127] Three years earlier, however, I was at Marlboro College during the Ice Storm of 2008, when the college was evacuated early for winter break. Multiple people I interviewed recalled that storm when discussing Irene.

[126] `http://www.thisshowissogay.com/`

[127] A social website. If you read this in 2030 and haven't heard of Facebook, hello there!

18 Infrastructure

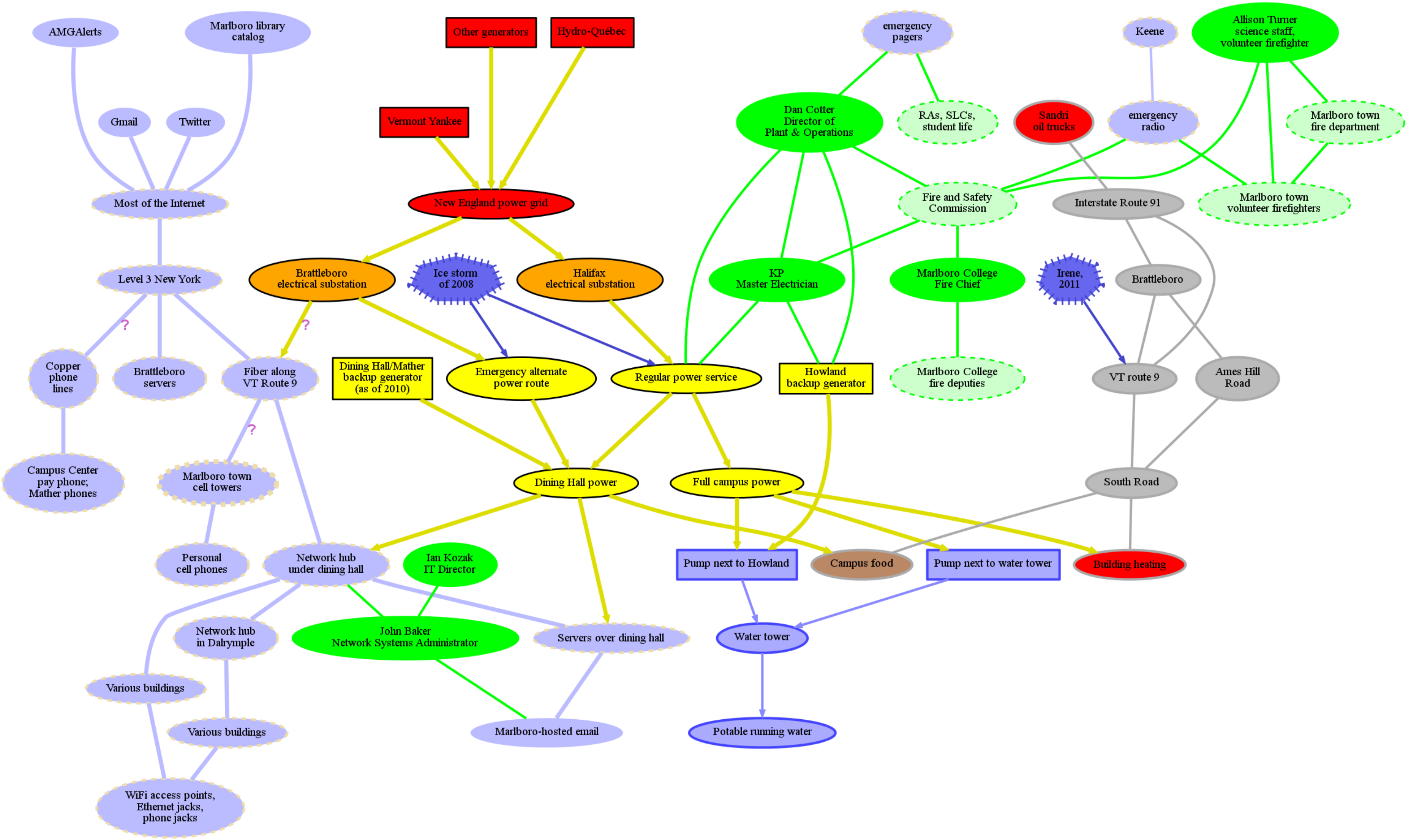
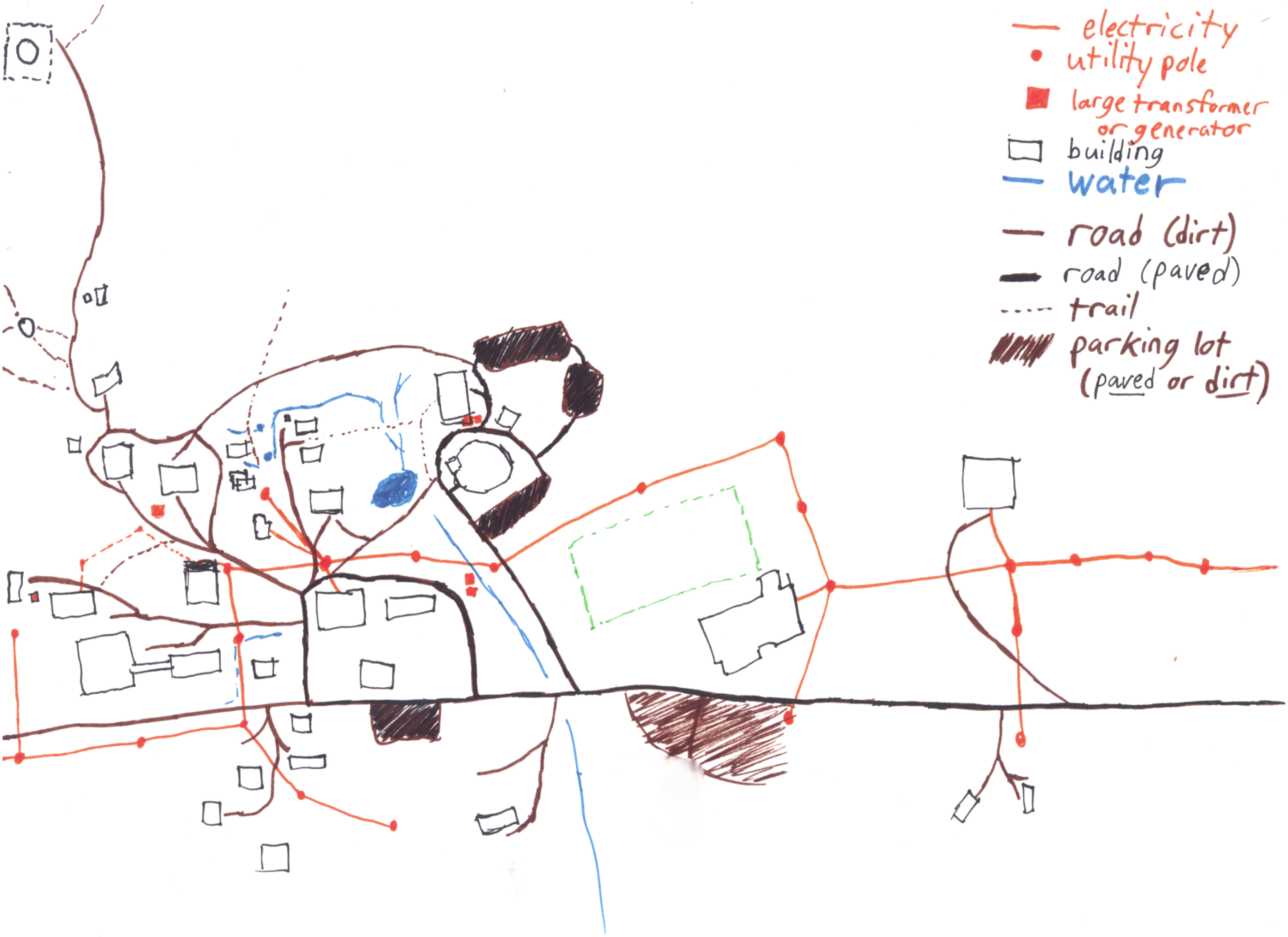


Figure 1: Symbolic sketch of infrastructure dependencies, as of 2011 and 2012.
Green: people. Light blue: communication. Blue: water. Spiky blue: disaster impacts.
Red/orange/yellow: electricity. Red: heating. Grey: roads. Brown: food. “?”: connections I’m not sure are accurate.

18.1 Electrical infrastructure

In May 2012, I surveyed the utility poles on campus and took photos of many pieces of utility infrastructure on campus. Some of the electricity distribution is underground. Hints at the underground infrastructure, such as transformers, are visible above-ground. Dan Cotter knows about the underground pipes through which power and/or networking cables are distributed, because he was involved in putting them in, most of them decades ago. A campus utility map is on the following page; the photos are in section 21.

Marlboro College power lines



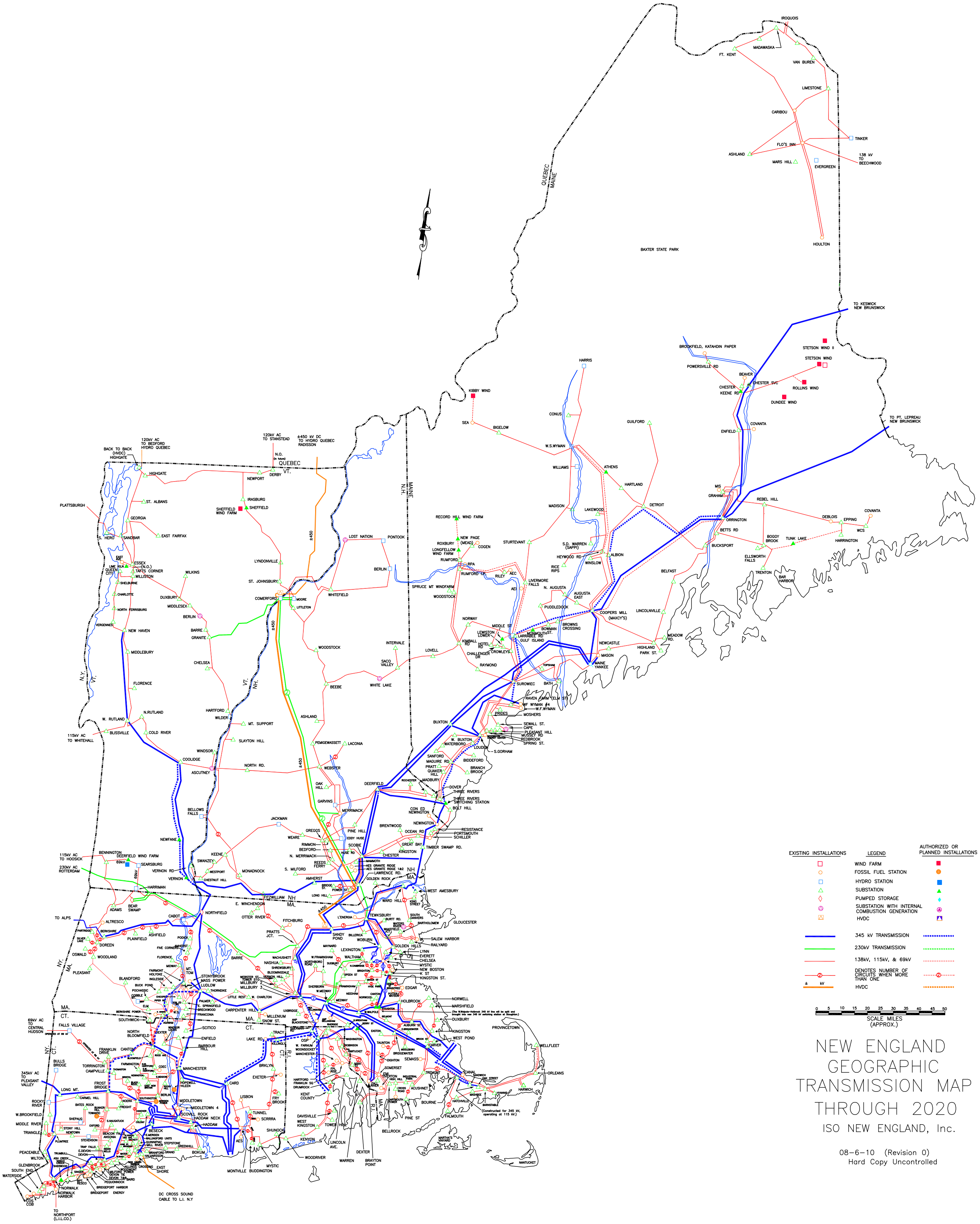


Figure 2: Major transmission lines in the New England power grid
http://www.iso-ne.com/nwsiss/grid_mkts/key_facts/2010_ne_trans_map_noncceii.pdf

When the power grid is intact, the college gets electrical power along power lines leading to a substation to our south in Halifax. Figure 2 shows the major electrical transmission lines in New England.[128] It is rare to have a year at Marlboro without several power outages from wind or winter weather.

When there is a power outage on the Halifax path, it is possible to switch the power connection at the junction of South Road and Lucier Road ("Freshman Curve") to connect northwards along South Road then east Route 9 to Brattleboro. This is a manual action and the transmission capacity in this direction is smaller than on the Halifax path, so it's only done if the south path isn't going to be fixed for a while and the north path is good. The power company is CVPS.

The college has several fixed generators. It has one by Howland that is activated manually. Since 2010, it has one that powers the Dining Hall and Mather that activates and deactivates automatically, and one powering Serkin. It has a couple portable generators. The generators are fuelled by either oil or propane.

The **New England power grid** is a network of power transmission lines that allow power plants to work together at powering the load. See Figure 2 for a map. It occasionally has enough redundancy to keep a high-voltage line failing in a storm from taking out anyone's power. These high-voltage lines are also more durable than typical roadside power poles; there are fewer of them, so it's economically practical to make them tougher.

Hydro-Québec is a collection of large hydroelectric plants in mid Québec. It provides power to Québec and New England. It has high-voltage AC and DC transmission lines from the plants south to the population centers. The 1998 North American ice storm, which did not affect Marlboro directly, knocked down these transmission lines, causing power shortages in New England for up to a month.[129]

[128] I haven't found exactly where the Halifax substation connects to these lines.

[129] [33], citing "McCready, J. 2004. Ice storm 1998: lessons learned. *Proceedings of the 6th Canadian Urban Forest Conference*, October 19-23, 2004. Kelowna, B.C., Canada."

Vermont Yankee is a notorious nuclear power plant in Vernon, VT, 12 miles as the crow flies from Marlboro College. We are within the radius designating places with a plan in case of a nuclear emergency.[130] Meanwhile, it is one of the many power sources for the New England power grid.

Other generators: The New England power grid has many generators, including coal, wind, and many others. If just a few fail, there is generally still enough power to sustain the grid; this allows plants to shut down for routine or unexpected maintenance.

18.2 Internet infrastructure

The on-campus communication network topology is chiefly underground. The primary network hub is in a locked room in the basement of the Dining Hall.[131] Many buildings are (network-wise) connected directly to this hub. One such building is Dalrymple, which contains another hub which connects to many more buildings. The connections between hubs are high capacity fiber-optic cables. Within each building is a smaller hub (switch) that connects by Ethernet cable to the building's WiFi access points and wall Ethernet jacks. In some buildings the WiFi access points are powered through Ethernet;[132] John Baker plans to use powered ethernet throughout more buildings as equipment gets upgraded. Most of the phone jacks on campus are also connected to the world through this Internet (IP) network. Non-powered landline phones will work with these phone jacks whenever the Internet does, but ones that need to be plugged into power won't.

For a map of the Hydro-Québec transmission infrastructure, see `https://commons.wikimedia.org/wiki/File:Quebec_Map_with_Hydro-Qu%C3%A9bec_infrastructures-fr.svg`

[130] The plan involves keeping a supply of iodine to be ingested in case of radiation; staying in certain buildings like the dining hall and using tape to make them to be as airtight as possible until the radiation decreases; evacuating to a shelter in Greenfield. One can only guess how well these plans will work. Would Greenfield even be far enough away? Hopefully we'll never need to find out. Vermont Yankee has a bad safety record, but it donates money to nonprofits in Brattleboro, keeping itself popular.

[131] Technology-wise, these are network switches.

[132] This is called Power over Ethernet. The network switch is connected to mains power and offers lower-voltage power through Ethernet cables.

[133] `http://maps.level3.com/default/`

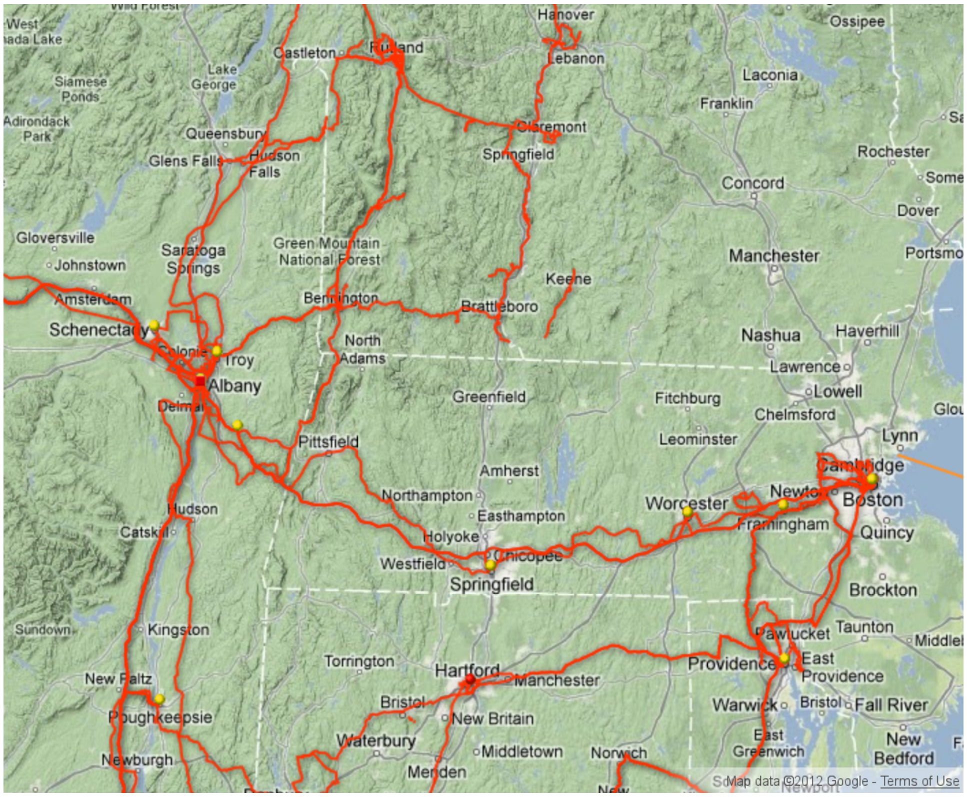


Figure 3: Level 3's fiber network[133]

The fiber-optic cables that connect the Dining Hall network hub to the rest of the Internet belong to Level 3 Communications, a multinational network operator.[134] These cables are strung along utility poles up South Road, where they connect to cables along Vermont Route 9. Westward they connect to a junction in Bennington, from where it leads in several directions. Eastward from Marlboro, it connects to Brattleboro, then to north of Brattleboro. See Figure 3 for a map. Network operators have peering agreements with each other so that messages can travel freely between cables owned by different operators; thus we are connected to the entire Internet.[135]

[134] Dan Cotter tells me; also `traceroute`'s reverse DNS, outgoing from Marlboro, names nodes on *.NewYork1.Level3.net.

[135] I could not find a comprehensive geographic map of the Internet, but other Internet service providers have similar maps in this region to Level 3's.

The Internet backbone is a redundant, fault tolerant network. It is built and operated by many providers who have agreements between each other to allow Internet traffic from each one into the other's network. BGP, an Internet routing protocol, helps data packets find a working pathway through the Internet backbone

The per-building network equipment has about 15 minutes of battery backup. The Dining Hall hub has about 2 hours of battery backup. Level 3's network tends not to be affected by power failures.[136] Laptops have batteries. Thus, when the power goes out, people have at least 15 minutes to post on social networking websites "The power's out!", and often do so.[137] Additionally, since mid-Fall 2010 (I believe)[138], the Dining Hall has a generator that automatically turns on when the power goes out. Now students often congregate in the Dining Hall to do homework when the power goes out on campus.[139]

Marlboro College has a Graduate Center in downtown Brattleboro. Its primary Internet connection is, administratively speaking, through a WAN with the main campus, because Level 3 charges us less for this than it would for separate connections.[140] The Grad Center also has a lower-bandwidth DSL connection through Sovernet that its staff use, useful when Marlboro College is disconnected. Brattleboro loses connectivity less often than Marlboro. The weather is more severe in Marlboro's mountains than Brattleboro's plain, and fewer people live in Marlboro.

Marlboro College also has a website[141] and several Internet services for community mem-

even when some parts of it are broken. The U.S. has a high enough density of Internet backbone nodes that this rarely if ever fails.

This architecture stems in part from the U.S. military, which needed a highly redundant, fault-tolerant network. Thus, the military provided some of the Internet's funding in the early days of the Internet.

Long-distance telephone calls are, nowadays, routed through the Internet as well.

[136] Fiber optic cable requires both ends of the cable to be powered but not the length of cable in-between. I am not aware of exactly where the cable from the college terminates and what backup that end has. It seems to work at least as well as the college's backup.

[137] I witness this on Facebook. I think people who frequent Twitter, Tumblr or other sites post there too. Such posting does not happen every outage, as people often post only if they happened to be using their laptop when the power went out. It is a surprise to many people that the Internet can work when the power's out. I feel surprise about this even now that I know how it works.

[138] If I recollect correctly when it happened. This generator was purchased as part of disaster preparedness planning prompted by the 2008 ice storm. It also powers Mather, the administration building, which is adjacent to the Dining Hall. Ellen McCulloch-Lovell, the college president, cares about disaster resiliency and helped lead these efforts.

[139] One time this happened was during Hurricane Sandy in 2012. Hurricane Sandy did not turn out to be a huge disaster for Vermont, but was a big disaster in other regions.

[140] The main campus has more people than the grad center, which is why the main connection to Level 3 is in Marlboro rather than Brattleboro.

[141] `http://www.marlboro.edu/`

bers. Most of them are hosted on campus on servers on the second level of the Dining Hall.[142] Since 2009, some Marlboro email accounts are hosted by Google rather than us (individual users have the option to upgrade to using Google). Since 2010, the library database is hosted off-campus far from here by ByWater Solutions.[143] These changes were not made for disaster resiliency; we liked the services better and they happened to be hosted off-campus.

When Marlboro loses connection, Marlboro's technical staff switch the public DNS servers to point to the backup DNS and website in Brattleboro (hosted by a friendly local company) where they can place a message to let outside folks, such as students' parents, know what's going on. Staff also switch the Grad Center's public Internet access (that's usually routed through the WAN to Marlboro) to point to the Sovernet DSL connection.

Senior staff have contemplated getting a satellite Internet or phone connection for times of disaster, but have not chosen to incur this expense.

18.3 Phone and radio

Most on-campus landline phones use the Internet infrastructure. Some phones have copper wire connections out, such as the pay phone in the Campus Center, and phones in Mather (the administration building) and Maintenance (the plant & operations building). The copper wire connections do not require any power on campus (unless they are used with a phone that requires mains power).

Cell phone towers have a few days' worth of backup power. Cell phones and pagers have batteries. Some cell phone batteries last much longer than others. During emergencies like Hurricane Irene that last a long time and sometimes knock down cell towers, mobile cell stations that connect via satellite may drive around. Sometimes these cellular networks do

[142] This level is not connected by foot indoors to the main hall; it has a separate exterior door and stairway. Some of the technical staff's offices are on this level.

[143] `https://catalog.marlboro.edu/`. ByWater Solutions use the open-source library-database software Koha, which was one of the reasons we switched. I was on the Library Committee at the time, though the staff did all the work.

not have sufficient bandwidth; cell phone use tends to increase during disasters.

Radio stations generally work if the transmitter is working. Bad weather can interfere with reception, however.

Marlboro College's buildings have fire alarms. These alarms have a control panel in the west entrance to Mather. When set off, as long as electricity and such are working, the fire alarms auto-dial 911. This goes to a dispatcher who pages the town of Marlboro's entire volunteer fire company. Thus, false alarms are a bad thing. False alarms are sometimes caused by cooking mistakes or other shenanigans.

There is also a regional dispatcher. There is a radio tower in Keene, NH that coordinates across the whole region.[144]

Pagers and emergency radios work on dedicated airwave frequencies that work over long distances. In Vermont, this is particularly important because many areas do not have cell phone reception. Some radios only receive; others can both transmit and receive messages on their frequency.

At Marlboro College, RAs (Resident Assistants) have a pager rotation: one of them has the RA pager at any given time. There are about a dozen RAs. This is for urgent residential life issues. SLCs (Student Life Coordinators) also have a pager rotation; there are three SLCs. This is for urgent medical or building issues that an RA couldn't handle. There is a pager rotation between the Dean of Students, the Director of Housing and Residential Life, and the Director of Psychological Services. The college fire chief has a pager radio rotation with their deputies which is paged by fire alarms. During business hours, for emergency purposes, dialing *611 on an on-campus landline dials the offices of senior staff. Marlboro town volunteer firefighters each have a pager radio. I believe Plant & Operations and possibly other college departments have pagers as well.

[144] Elias Zeidan told me about these emergency response arrangements; I may be getting some details wrong.

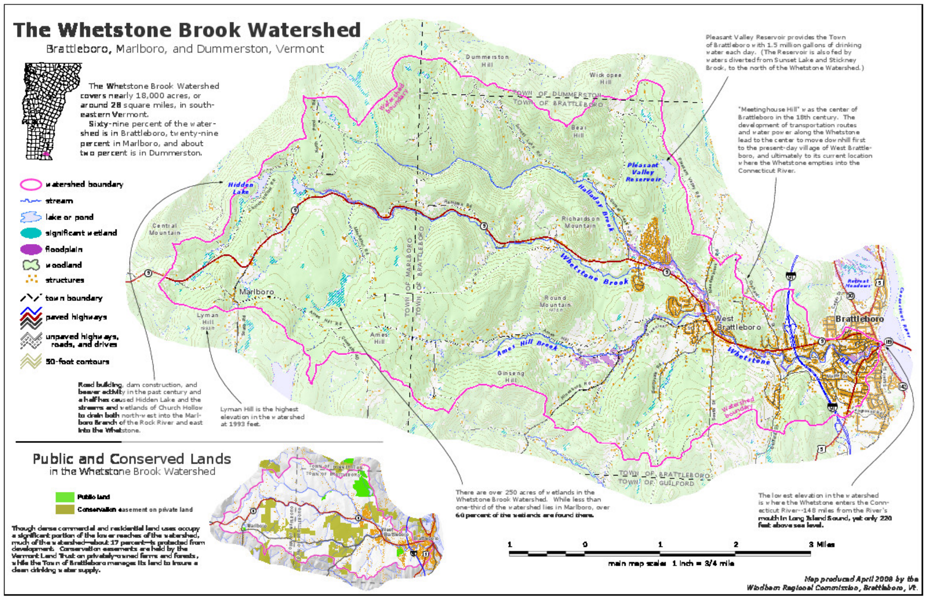

The Whetstone Brook Watershed
Brattleboro, Marlboro, and Dummerston, Vermont
The Whetstone Brook Watershed covers nearly 18,000 acres, or around 28 square miles, in south-eastern Vermont.
Sixty-nine percent of the watershed is in Brattleboro, twenty-nine percent in Marlboro, and about two percent is in Dummerston.
watershed boundary
stream
lake or pond
significant wetland
floodplain
woodland
structures
town boundary
paved highways
unpaved highways, roads, and drives
50-foot contours
Road building, dam construction, and beaver activity in the past century and a half has caused Hidden Lake and the streams and wetlands of Church Hollow to drain both north-west into the Marlboro Branch of the Rock River and east into the Whetstone.
Lyman Hill is the highest elevation in the watershed at 1993 feet.
Pleasant Valley Reservoir provides the Town of Brattleboro with 1.5 million gallons of drinking water each day. (The Reservoir is also fed by waters diverted from Sunset Lake and Stickney Brook, to the north of the Whetstone Watershed.)
"Meetinghouse Hill" was the center of Brattleboro in the 18th century. The development of transportation routes and water power along the Whetstone lead to the center to move downhill first to the present-day village of West Brattleboro, and ultimately to its current location where the Whetstone empties into the Connecticut River.
There are over 250 acres of wetlands in the Whetstone Brook Watershed. While less than one-third of the watershed lies in Marlboro, over 60 percent of the wetlands are found there.
The lowest elevation in the watershed is where the Whetstone enters the Connecticut River--148 miles from the River's mouth in Long Island Sound, yet only 220 feet above sea level.
Hidden Lake
Central Mountain
Marlboro
Lyman Hill
Ames Hill
Dummerston Hill
Wickopee Hill
Bear Hill
Pleasant Valley Reservoir
Richardson Mountain
Whetstone Brook
Round Mountain
West Brattleboro
Brattleboro
Ginseng Hill
Watershed boundary
TOWN OF MARLBORO
TOWN OF BRATTLEBORO
TOWN OF DUMMERSTON
TOWN OF BRATTLEBORO
TOWN OF BRATTLEBORO
TOWN OF GUILFORD
Public and Conserved Lands
in the Whetstone Brook Watershed
Public land
Conservation easement on private land
Though dense commercial and residential land uses occupy a significant portion of the lower reaches of the watershed, much of the watershed—about 17 percent—is protected from development. Conservation easements are held by the Vermont Land Trust on privately-owned farms and forests, while the Town of Brattleboro manages its land to insure a clean drinking water supply.
1 0 1 2 3 Miles
main map scale: 1 inch = 3/4 mile
Map produced April 2008 by the Windham Regional Commission, Brattleboro, Vt.

Figure 4: The Whetstone Brook watershed

`http://www.windhamregional.org/gis/watershed-maps`

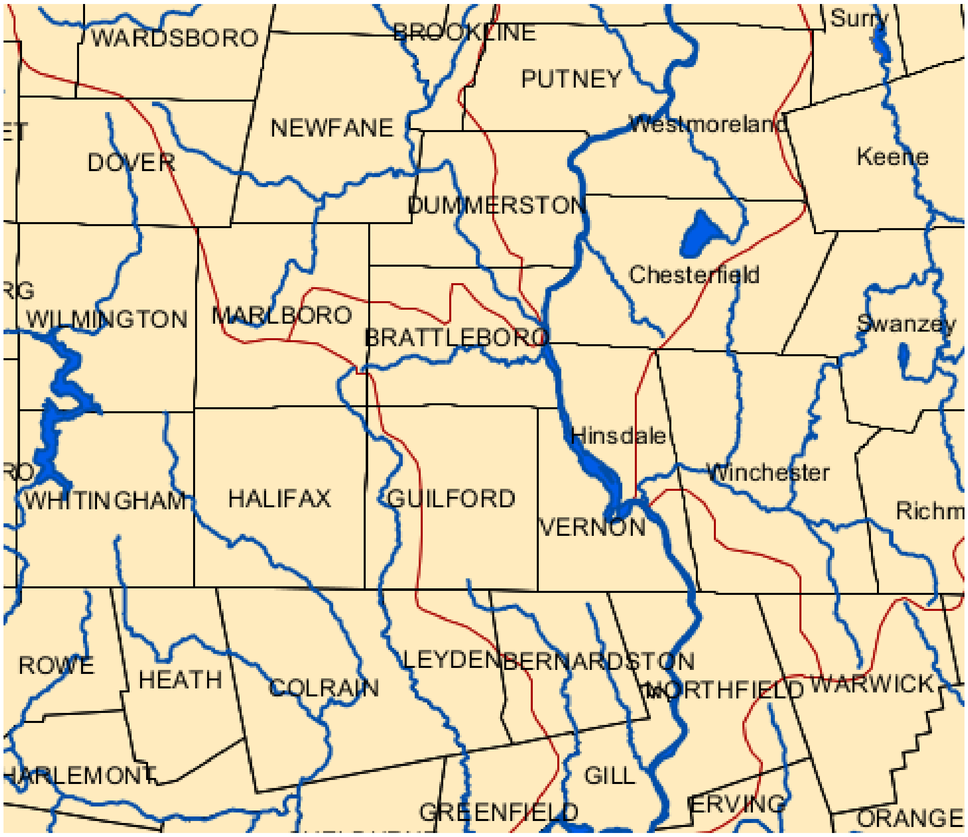


Figure 5: Marlboro region towns and rivers
(brown lines divide watersheds)

`http://ctriver.org/our_region_and_rivers/maps/watershedsmap.html`

18.4 Roads and watersheds

Brattleboro adjoins the Connecticut River, interstate highway 91, and a passenger rail line. These all run roughly parallel for dozens of miles, south through Massachusetts and north along the eastern border of Vermont.

There are two roads that connect Marlboro College to Brattleboro: VT Route 9, a paved, heavily-trafficked (for rural Vermont) two-lane road, and Ames Hill Road, an unpaved, often muddy road a mile to its south. The college is on South Road, a paved road that leads from the college, past Ames Hill Road and the center of town[145] to Route 9. A few more dirt roads lead south and north from Marlboro. Route 9 continues west to Wilmington, Bennington and New York State, and east through Brattleboro and Keene (NH) to Maine.

Marlboro College is near the edges of several watersheds. Rain that falls on the college flows into the Green River, which merges with the Connecticut further south in Greenfield, MA (see Figure 5). Rain that falls just a few miles northeast of us flows down the Whetstone Brook, through Brattleboro, into the Connecticut River, as shown in Figure 4.

The Whetstone Brook parallels Vermont Route 9 for most of the distance between Marlboro and Brattleboro. Ames Hill Road is a mile away but still in the same watershed. When Hurricane Irene rained on the area, water flowed into the Whetstone Brook and washed away large portions of Route 9 between Marlboro and Brattleboro. This water then hit Brattleboro and damaged many homes and businesses.[146] Ames Hill Road was not as destroyed, but it is unpaved and not designed for much traffic. After some restoration work, it became passable long before Route 9 did and served as Marlboro's road connection to the world for a month.

When the roads were impassable after Irene, Vermont Governor Peter Shumlin visited

[145] The center of town consists of a church, a post office, and rural houses denser than in the rest of the town.

[146] *Good Night Irene* by Craig Brandon et al. describes the devastation in Brattleboro, as well as the even worse flood in Wilmington to Marlboro's west, and a half-dozen other towns. The *Brattleboro Reformer*, a local newspaper, published many articles about the storm's effects as well.

the town of Marlboro by landing in a helicopter on Zimmerman Field, which is the college's soccer field and one of the only places in the town of Marlboro large, flat and open enough to land a helicopter.

18.5 Drinking water

Marlboro College has a water tower half a mile up the hill from the college. It provides pressure and water. It holds several weeks' supply of water. It is supplied by a well next to the water tower and a well next to Howland.[147] To my knowledge, drought has never impacted this water supply.

I hear tales, but I have never seen us without water in my four years here. Joy Auciello remembers a time in 2004 when bacteria got in the water supply and "drinking water had to be trucked in".[148] The Dining Hall's drink machine doesn't work when the power goes out. Hot water is stored in heaters that don't work without power; its heat quickly runs out. But sinks have supplied drinking water even during the major disasters in the last several years, the 2008 ice storm and 2011 storm Irene.

Our sewage flows into a series of septic tanks that begins on the hill above Maintenance and continues at points further out in the woods. This is lower than the main part of campus; gravity sends waste where it needs to go.

18.6 Heat

Buildings are heated by heating oil[149] in boilers that require electrical power. Heating oil is delivered by oil truck. The buildings store at least a few days' worth of heating oil. In the 2008 ice storm, the only generators we had were a stationary generator for Howland and a

[147] interview with Dan Cotter

[148] `https://www.facebook.com/groups/189481367744500/permalink/563478643678102/` (a thread in the Marlboro College Facebook group that I started to ask for information for this Plan paper).

[149] Except for Married Student Housing, which is heated by electricity.

portable generator. Power was out for days; KP heroically moved the portable generator from building to building to periodically run the boilers, making sure that people and pipes wouldn't freeze. Hurricane Irene occurred in August so freezing was not an issue.

18.7 Food

The Dining Hall feeds most people on campus. Its refrigeration, cooking and dish-washing do not work well without electricity. The generator is enough to power refrigeration and cooking, but not dish-washing, so disposable plates and utensils are used when the power is out. Before this generator was installed, cooking was affected much more by outages.

Food is usually delivered by large trucks. For the months that Ames Hill Road was the only access to the college, these trucks could not navigate it; Richie, the head of the kitchen, regularly drove Ames Hill Road to get food onto campus.

18.8 Smoking

No place on campus sells cigarettes, yet a lot of students are addicted. At times when it is difficult to get off campus, this leads to a shortage and smokers helping each other out.[150]

18.9 Light

Buildings have battery-backed emergency lighting that lasts at least several hours. Many students have flashlights or headlamps. The campus outdoors is not well lit even when all is well.

[150] Alas, I know less about this than I might because I'm sensitive to cigarette smoke and tend to stay away from people who are smoking.

19 Emotions

During disasters, people's emotions vary from excited to indifferent to scared to any combination. For example, each of the people I interviewed had different responses. Some were used to dealing with emergencies; some were kept busy by dealing with them and only processed it emotionally later; some were almost bemused; some appreciated the chance to spend time outside. Some people's families were affected; others' weren't.

Staff varied in the electricity, phone, water, and so forth that they had access to at home. Some have solar panels, or wells. Some have cell phones; some have powered or non-powered landline phones; some never get cellular service at home. Vermont is rural and has a fair bit of diversity in these things.

19.1 "Marlboro College No Power No Water"

Marlboro College's motto, if we have one, is "Marlboro College, no power, no water, $40,000 a year." I've heard this reference dozens of times in my time here. It used to go "$30,000 a year". A web search turned up some history. A Marlboro student, after the 2008 ice storm, used the phrase in her blog.[151] The phrase is still around today. Before Hurricane Sandy hit in 2012, a student blogging about the hurricane tagged their post "marlboro college", "no water no power", "40 thousand dollars a year".[152] Students often say it whenever the power flickers or goes out. In 2012 I asked about the phrase's history on the Marlboro College Facebook group.[153] A professor quipped popularly "Unless I've misinterpreted the question. Marlboro College goes back all the way to 1946 and has been offering a lack of power and lack of water for a modest sum for all that time.". It appears that the phrase was created between 2002 and 2004, according to Allison Turner quoting Franklin Crump for the

[151] http://rachelknight.blogspot.com/2008/12/is-internet-addiction-hitting-ludicrous.html

[152] http://rosiebeeloved.tumblr.com/post/34451872275/so-this-is-happening

[153] https://www.facebook.com/groups/189481367744500/permalink/563478643678102/ (not public).

Facebook thread. It was improvised in song by the student band Big Dragon Truckstop, consisting of Franklin Crump, Forrest Gardner, and Chris Jones. That song was digitally recorded, and the recording still lives.[154] The phrase gained a life of its own, bringing it to where it is today.

The phrase is often used in a silly way. The "$40,000 a year", however, is often omitted, as students often cringe to think about the price.

20 Knowledge dynamics

20.1 Complex systems

There are many complex systems that we rely on. I have shown some of them above: communication and power and water and so forth. Some of them give helpful redundancy, such as cell phones and emergency radio. Some are *tightly coupled*: a failure in one leads to a failure in another. A lack of power leads to a lack of building heat. Damaged phone lines and cellular towers couple with high telephone usage during disasters to overload a system that normally works fine. Toxic materials, such as gasoline, are caught in floods; the floods then do more damage to farmland and buildings they hit. We're unlikely to know what all these interactions will be before they happen. Charles Perrow describes effects like this in *Normal Accidents: Living With High-Risk Technologies* [30].

Perrow is concerned with particularly risky systems like nuclear power plants and aircraft, but he opens with a human-scale example. A person plans to arrive early to an important job interview. They have plenty of ways to get to the interview. But they wake up and their roommate has accidentally broken the coffee-pot, so they dig up a slower coffeemaker. After having coffee, they rush out the door, forgetting their car keys and locking the door behind

[154] `http://akbar.marlboro.edu/~johnny/nopowernowater.mp3`; also included with the digital version of this Plan of Concentration.

them. They usually leave a key in a flowerpot for this, but they've lent it to a friend. Their neighbor's car happens to be broken. The bus drivers are on strike, and because there are no buses, the city's taxis are too busy to give them a ride. No one of these problems would have stopped them from getting to the interview. But a few small unlucky things were enough to stop them, and a few unlucky things will always happen eventually. For some apparently redundant transportation (buses and taxis), it turned out that one's failure causes the other to fail as a method for our poor example person.[155]

20.2 Familiarity and planning

People were able to communicate during disasters because they were already familiar with communicating with each other. For example, on the day of Hurricane Irene, KP called the electric company to make sure they knew about the power outage, as she has a habit of doing. They knew already, but, as she said, it can't hurt to make sure. The Marlboro staff and faculty work together more closely as a group at Marlboro than happens at most colleges; Marlboro is small and democratic by temperament.

Marlboro has made a point to think ahead for disasters. I have a friend at another college which was hit hard by Hurricane Irene in 2011 and Hurricane Sandy in 2012; their college hadn't learned from its disorganization after Irene at all.

At the same time, ingenuity during disasters saves the day. After the 2008 ice storm, Ben and I called our families using a pay phone that most likely was not installed for the sake of emergencies. The kitchen cooked creatively in the face of failing power and refrigeration. Professors contacted each other to check in with each other and to see whether they could help. In fact, wanting to help others during disasters is a common response; panicking or looting is not.[156]

[155] Charles Perrow, *Normal Accidents: Living With High-Risk Technologies* [30], p. 6.

[156] See e.g. Kathleen Tierney, Christine Bevc and Erica Kuligowski, "Metaphors Matter: Disaster Myths, Media Frames, and Their Consequences in Hurricane Katrina" in The Annals of the American Academy of

Political and Social Science 2006, 604: 57.

Town of Marlboro, Vermont
Emergency Preparedness Register 2011

Name____

Phone(h)______________________________ Phone(cell)__________________________

911(Street) address__________________________ email address____________________

Mailing address____

□ Year-round resident or □ Part-time resident (list months) _______________________
□ I would like to meet with someone to talk about my emergency plan.
□ I wish to be informed about ways of communicating during an emergency.

Needs:
□ I would need transportation to leave my home in an emergency.
□ My driveway is long or otherwise difficult for emergency access.
□ during a power outage I **do not** have heat
□ during an outage I **do not** have water
□ I do not have a battery-operated radio.
□ I have an electricity generator but will need assistance with it.

I have medical devices that require electricity:
□ Monitoring devices (heart rate, blood pressure, blood oxygen, etc)
____
□ Respirator or breathing devices;
□ Oxygen concentrator;
□ Other____________________
□ I have medications or special dietary needs that require refrigeration.

□ I am deaf. □ I'm sight impaired.
□ I use a walker or cane. □ I am wheelchair-bound. □ I am bedridden.

Resources:
□ I could provide transportation for others in an emergency.
□ I have a snowmobile □ I have an ATV
□ I can provide water for others during a power outage.
□ I have an "in-line" generator (it comes on automatically when power is out).
□ I have a portable generator.
□ I have a source of heat not requiring electricity (describe):
____
□ I could shelter others in case of need.
How many? ____________________
□ I have a landline phone that works in an outage (it does not plug into an electrical outlet).

Do you have other special circumstances or needs in emergency situations? Please describe:

____

____

****

Relatives or Friends (not living with you) to be notified in the event of emergency or evacuation:

Name (Local) __________________________Phone_______________Cell_______________
Address____

Name (Out of Town)_____________________Phone_______________Cell_______________
Address____

Privacy Notice: The above information will be held confidential and will only be used by Town Emergency personnel for emergency planning and to assist residents in the event of an emergency or evacuation.

Please return completed form in the enclosed envelope. Thank you.

Solutions that are out of mind tend to be forgotten; in the aftermath of Irene, everyone forgot that Marlboro College had subscribed to a web service AMGAlerts[157] that's for sending emergency notices to everyone subscribed (such as students and parents) by phone, text, email, and so forth. In the town of Marlboro, there are many people who don't see each other on a daily basis. Town community members discussed ways to make sure everyone's okay during a disaster. In the past, neighbors have often visited each other on foot. This is not ideal in disaster conditions. One idea was for neighbors to have walkie-talkies for each other; but unless neighbors use them for something on a daily basis, they're likely to forget all about them. Someone suggested walkie-talkies could be a friendly way to chat, to keep them in memory.

Making announcements at places that are naturally community hubs has worked well. At Marlboro College, the Dining Hall is such a place. During disasters, people congregate there, or go there to find out what's going on. Folks have also remembered Twitter.[158] Marlboro College maintains a Twitter account @MarlboroCollege. This account is used regularly, and is also part of disaster planning as one way to spread news during severe weather events.

20.3 Vulnerability and inequality

Humans are often resilient, but social inequality puts people on the edge of disaster in their everyday life. Large events that we call disasters, then, reveal the disasters of vulnerability and inequality that already exist in our communities. For example, where do folks living in trailer parks that were flooded by Irene go now? Often, hazards from nature or industry become disasters when our infrastructural and social fabrics are weak.[159]

One approach for addressing inequality and lack of community is named Community

[157] `http://amgalerts.com/`. "100% web-based": `http://amgalerts.com/faq.asp`. The college subscribed as part of disaster preparedness work undertaken after the 2008 ice storm.

[158] Twitter is a currently popular web and SMS social network for short messages.

[159] For further reading on this perspective, see Brenda D. Phillips et al., *Social Vulnerability to Disasters* [31].

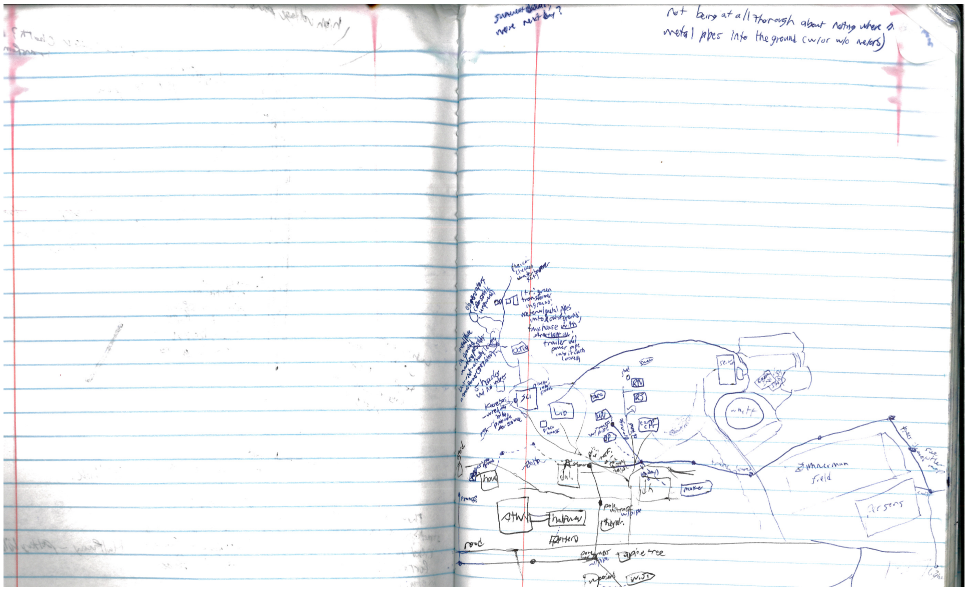
Asset Mapping. People in a community who care get together and list all the positive community spaces, energies or people that already exist in a community. The community builds from these strengths, rather than stare depressed at what they don't have.[160]

[160] I have a few handbooks that show an arts-focused element of such strategies. These include:
Keith Knight, Mat Schwarzman et al., *Beginner's Guide to Community-Based Arts* [17].
Michael Rohd, *Theatre for Community, Conflict and Dialogue* [37].

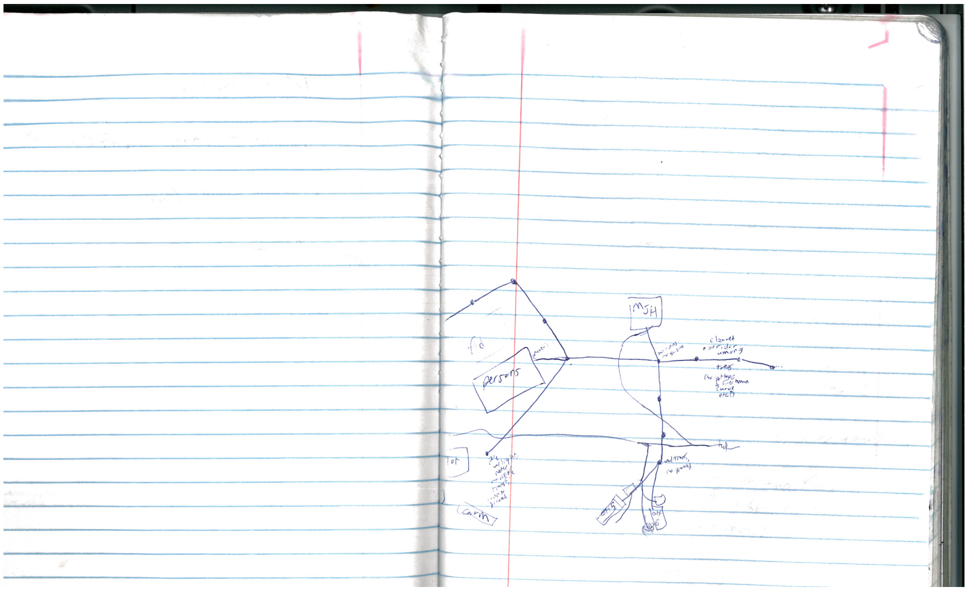
21 Photos

This section contains chiefly photos of on-campus electrical distribution infrastructure, plus a few others relevant to this paper.

The original maps I drew in the field:

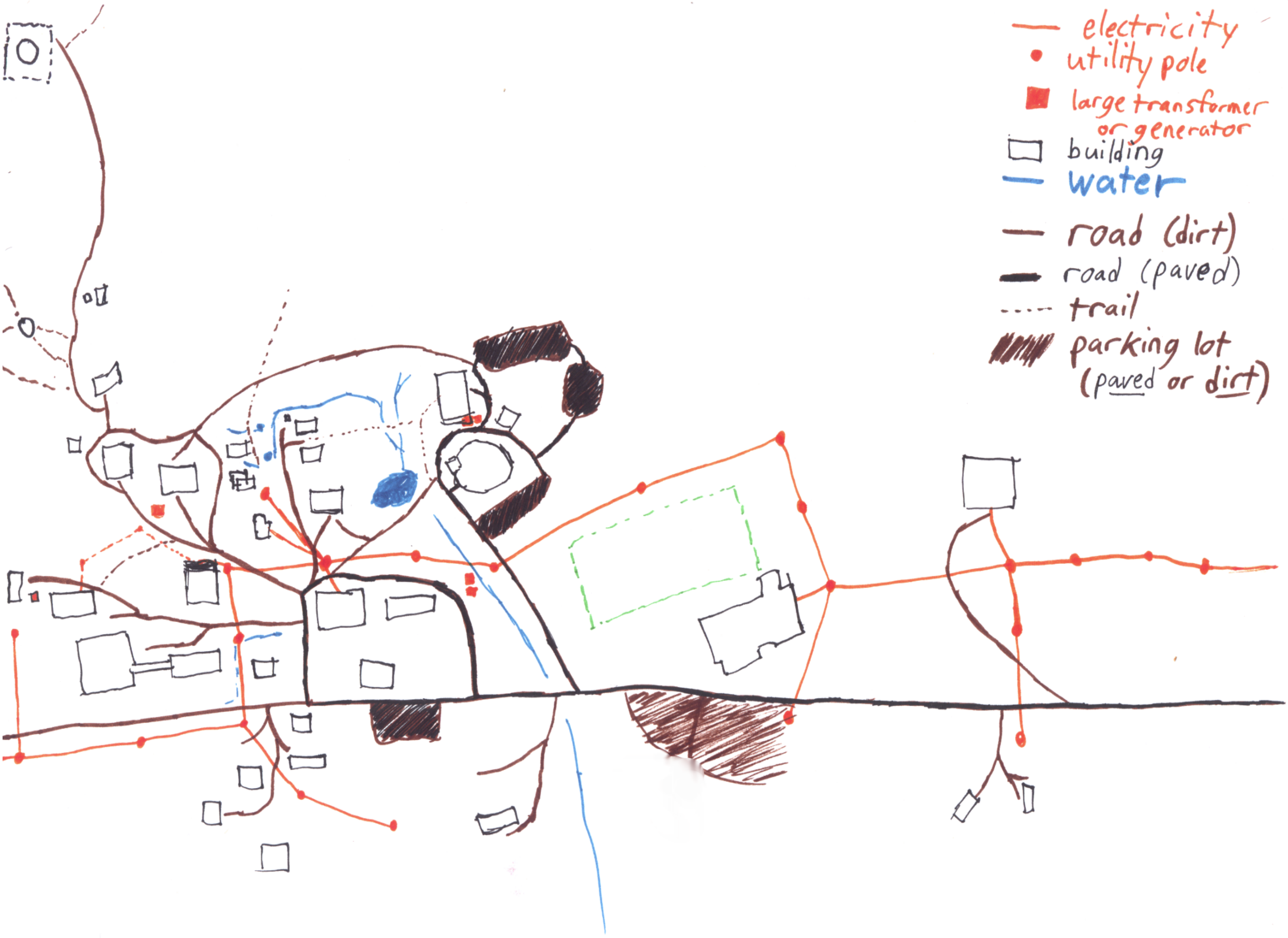


were next day?
Not being at all thorough about noting where metal pipes into the ground (w/or w/o meters)
green transformer in ground
meter w/ metal pipes into ground
trailer w/ power pipe into it
Sci
Lib
RM
RJ
camp ctr
Sem
white
drainage
path
Jul.
road
ATW
halfway
apple tree
Zimmerman field
Persons
see other map

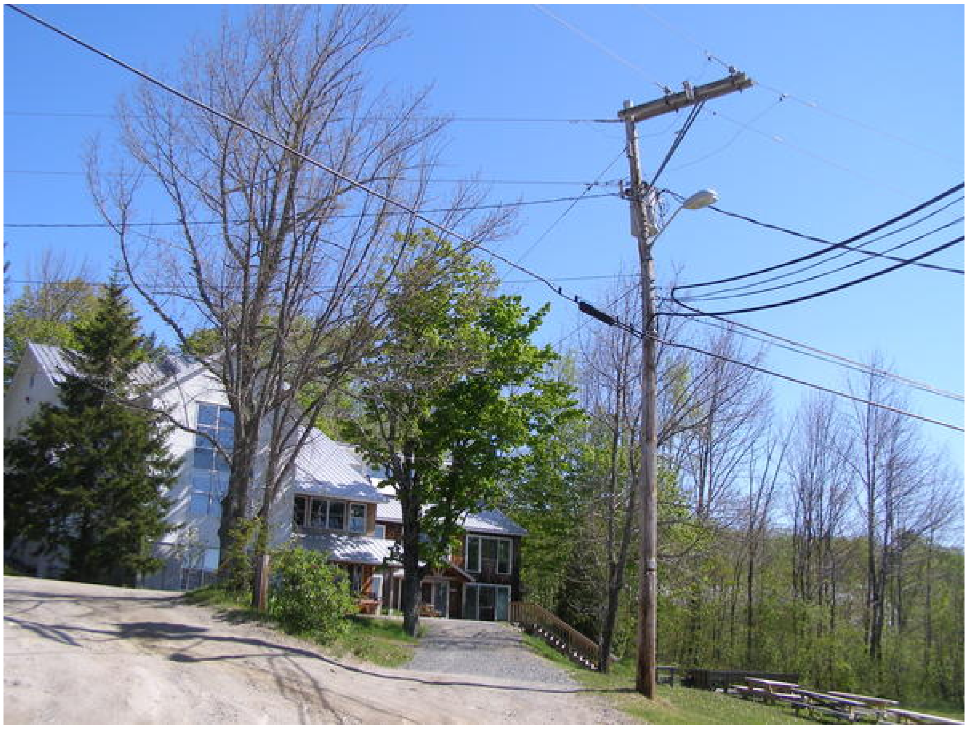


MJH
fld
Persons
cleared corridor among trees
(to cottages & freshman curve etc?)
w/ trans, no ground
pole w/ light
lot
CHS

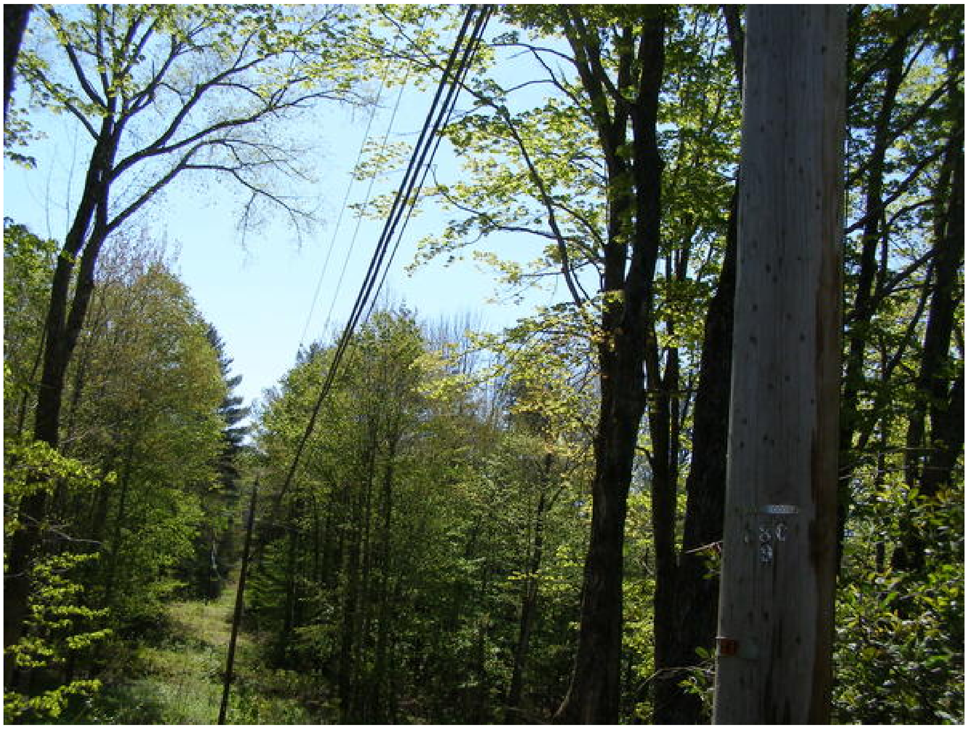
Marlboro College power lines, traced from the original fieldwork



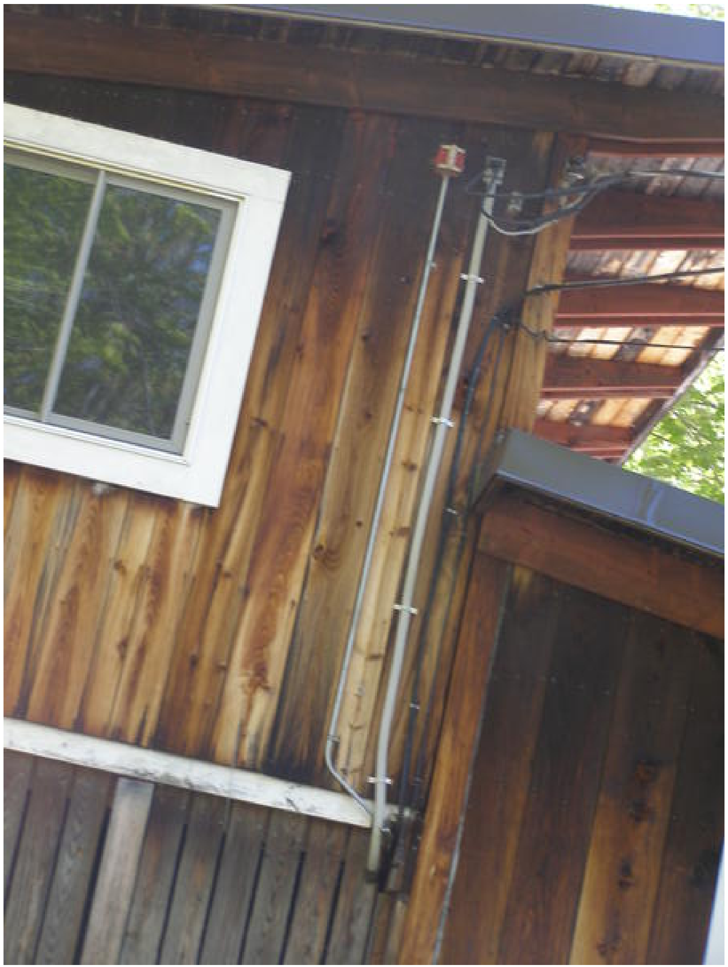
These pictures were taken May 2012



The power pole between the Dining Hall, Campus Center (which the picture faces), and OP (Outdoor Program building), in the center of campus.



Facing east from near Married Student Housing off the east side of campus. Power arrives here from South Road/Lucier Road to the east.



Power lines connect to the Married Student Housing building.

Base of a power pole (treated to prevent rot, perhaps using creosote). The wire down the side of the pole connects to the ground in order to be an electrical ground.

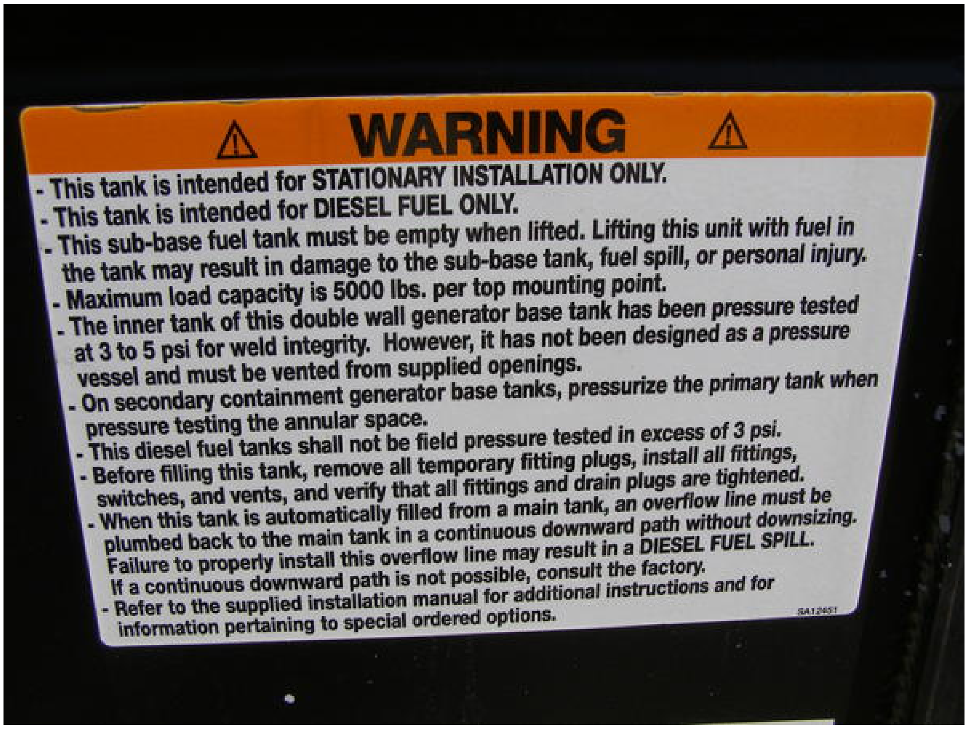
Power pole west of Married Student Housing and east of Persons Auditorium (right). View faces south. Power lines continue north then west around the soccer field to campus.

Parking lot street lamps (Upper Theater lot). I believe they are grid-connected. Some of the street lamps have motion sensors, light sensors or timers to save energy when the light is not needed.

Recycling dumpster next to the Theater's costume shed (northeast of Whittemore Theater). The recycling dumpster used to be behind the Dining Hall. Cliff Inman collects our recycling, and that of many other organizations in Windham county, and takes it to the recycling center in Brattleboro, Windham Solid Waste Management District `http://www.windhamsolidwaste.org/`.

Speed limit 10 mph on the way to parking lots keeps people safe. This wasn't always paved. It was controversial when it was paved. Beside the road, a ditch with rocks offers drainage.

Transformer and generator beside Serkin Center for the Arts.



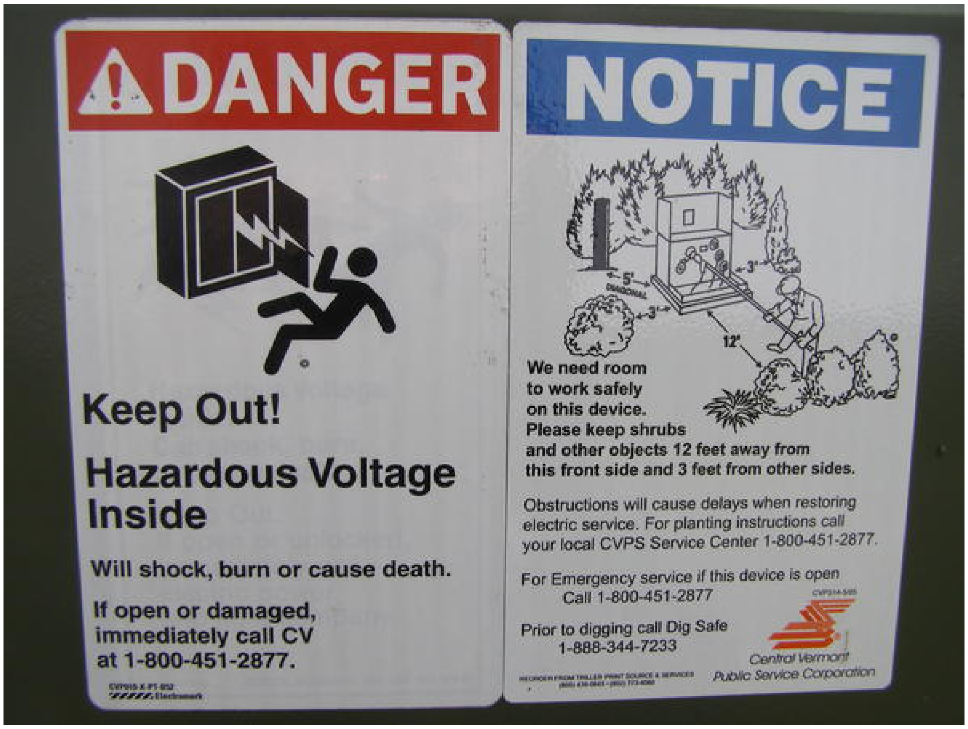
⚠ WARNING ⚠

- This tank is intended for STATIONARY INSTALLATION ONLY.
- This tank is intended for DIESEL FUEL ONLY.
- This sub-base fuel tank must be empty when lifted. Lifting this unit with fuel in the tank may result in damage to the sub-base tank, fuel spill, or personal injury.
- Maximum load capacity is 5000 lbs. per top mounting point.
- The inner tank of this double wall generator base tank has been pressure tested at 3 to 5 psi for weld integrity. However, it has not been designed as a pressure vessel and must be vented from supplied openings.
- On secondary containment generator base tanks, pressurize the primary tank when pressure testing the annular space.
- This diesel fuel tanks shall not be field pressure tested in excess of 3 psi.
- Before filling this tank, remove all temporary fitting plugs, install all fittings, switches, and vents, and verify that all fittings and drain plugs are tightened.
- When this tank is automatically filled from a main tank, an overflow line must be plumbed back to the main tank in a continuous downward path without downsizing. Failure to properly install this overflow line may result in a DIESEL FUEL SPILL. If a continuous downward path is not possible, consult the factory.
- Refer to the supplied installation manual for additional instructions and for information pertaining to special ordered options.

SA12451

Warnings on the fuel tank beside Serkin.

Transformer beside Serkin.


DANGER
Keep Out!
Hazardous Voltage Inside
Will shock, burn or cause death.
If open or damaged, immediately call CV at 1-800-451-2877.
NOTICE
5'
DIAGONAL
3'
3'
12'
We need room to work safely on this device. Please keep shrubs and other objects 12 feet away from this front side and 3 feet from other sides.
Obstructions will cause delays when restoring electric service. For planting instructions call your local CVPS Service Center 1-800-451-2877.
For Emergency service if this device is open Call 1-800-451-2877
Prior to digging call Dig Safe 1-888-344-7233
Central Vermont Public Service Corporation

Warnings on the transformer beside Serkin.

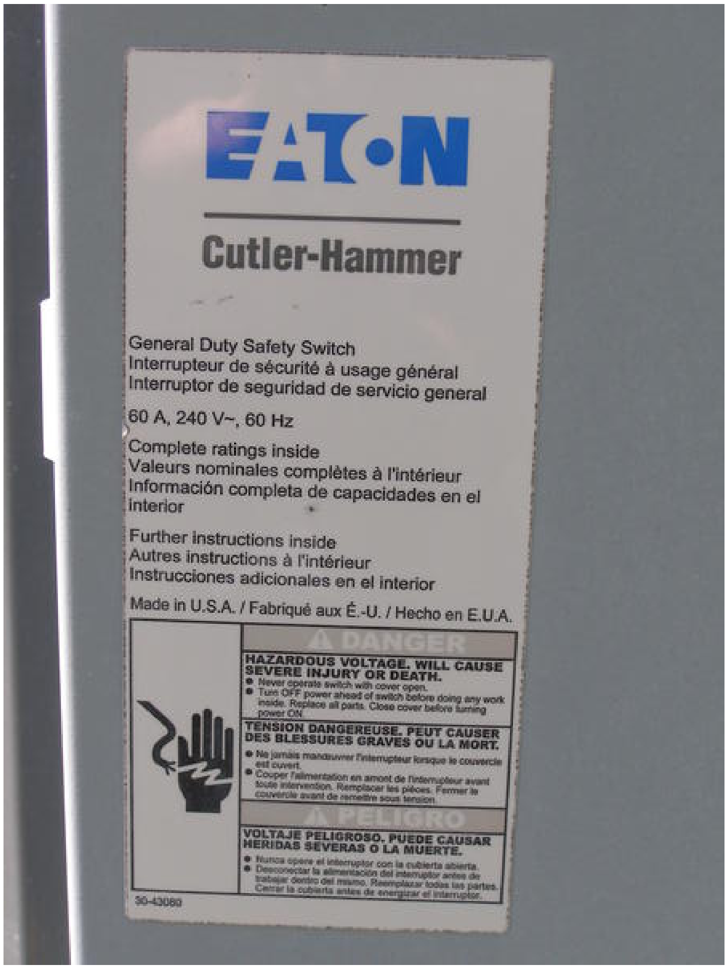
Information on the transformer beside Serkin. “NO PCBs (LESS THAN 2 P.P.M.)” refers to polychlorinated biphenyls, carcinogenic chemicals formerly allowed in electrical devices.

Handicapped parking sign behind Serkin. This helps include people with handicaps, who are an important part of area community. This was helpful for a Windham County disaster planning meeting held in Serkin in Jan or Feb 2012.[161]

Pipes, probably electrical conduits, between the building and the ground.

[161] I was there. Jon Mack, Newfane Selectboard member and former Marlboro College professor, was there. Several dozen people were present. I've lost any other information about it.

An HVAC system of some sort for Serkin (cont'd).



EATON

Cutler-Hammer

General Duty Safety Switch
Interrupteur de sécurité à usage général
Interruptor de seguridad de servicio general

60 A, 240 V~, 60 Hz

Complete ratings inside
Valeurs nominales complètes à l'intérieur
Información completa de capacidades en el interior

Further instructions inside
Autres instructions à l'intérieur
Instrucciones adicionales en el interior

Made in U.S.A. / Fabriqué aux É.-U. / Hecho en E.U.A.

⚠ DANGER

HAZARDOUS VOLTAGE. WILL CAUSE SEVERE INJURY OR DEATH.

- Never operate switch with cover open.
- Turn OFF power ahead of switch before doing any work inside. Replace all parts. Close cover before turning power ON.

TENSION DANGEREUSE. PEUT CAUSER DES BLESSURES GRAVES OU LA MORT.

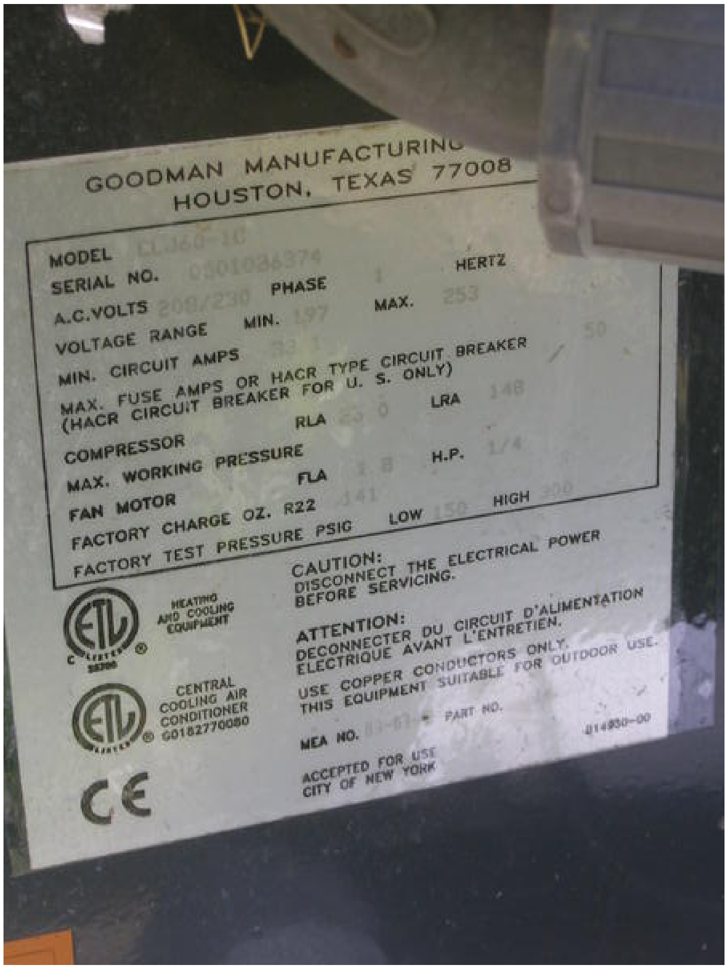
- Ne jamais manœuvrer l'interrupteur lorsque le couvercle est ouvert.
- Couper l'alimentation en amont de l'interrupteur avant toute intervention. Remplacer les pièces. Fermer le couvercle avant de remettre sous tension.

⚠ PELIGRO

VOLTAJE PELIGROSO. PUEDE CAUSAR HERIDAS SEVERAS O LA MUERTE.

- Nunca opere el interruptor con la cubierta abierta.
- Desconectar la alimentación del interruptor antes de trabajar dentro del mismo. Reemplazar todas las partes. Cerrar la cubierta antes de energizar el interruptor.

30-43080



GOODMAN MANUFACTURING
HOUSTON, TEXAS 77008

MODEL CL360-1C
SERIAL NO. 0501086374
A.C.VOLTS 208/230 PHASE 1 HERTZ
VOLTAGE RANGE MIN. 197 MAX. 253
MIN. CIRCUIT AMPS 33.1
MAX. FUSE AMPS OR HACR TYPE CIRCUIT BREAKER 50
(HACR CIRCUIT BREAKER FOR U. S. ONLY)
COMPRESSOR RLA 25.0 LRA 148
MAX. WORKING PRESSURE
FAN MOTOR FLA 1.8 H.P. 1/4
FACTORY CHARGE OZ. R22 141
FACTORY TEST PRESSURE PSIG LOW 150 HIGH 300

HEATING AND COOLING EQUIPMENT

CENTRAL COOLING AIR CONDITIONER G0182770080

CE

CAUTION:
DISCONNECT THE ELECTRICAL POWER BEFORE SERVICING.

ATTENTION:
DECONNECTER DU CIRCUIT D'ALIMENTATION ELECTRIQUE AVANT L'ENTRETIEN.

USE COPPER CONDUCTORS ONLY.
THIS EQUIPMENT SUITABLE FOR OUTDOOR USE.

MEA NO. 83-63-E PART NO.

014930-00

ACCEPTED FOR USE
CITY OF NEW YORK

(cont'd) More pictures of the HVAC system.

Town Trail sign along the dirt road around the back of campus between Serkin and the Science Building. It has a blaze and a reminder to avoid stepping in ski tracks during the winter (skiers like to have intact ski tracks to ski in).

Between Dalrymple (in the distance) and Happy Valley (behind and to the right) is a parked car. It is forbidden to park on campus unless one is a volunteer firefighter or a person with a relevant disability. Students involved with Fire & Safety write tickets for illegally parked cars, but that doesn't stop a lot of students from doing it. I am taking these pictures at the end of the school year, so cars may also be parked on campus to help students move out.

A fire hydrant is in the distance near Dalrymple.

Power pole connecting the central campus pole to Happy Valley (left). This view faces east with the Campus Center in the background.

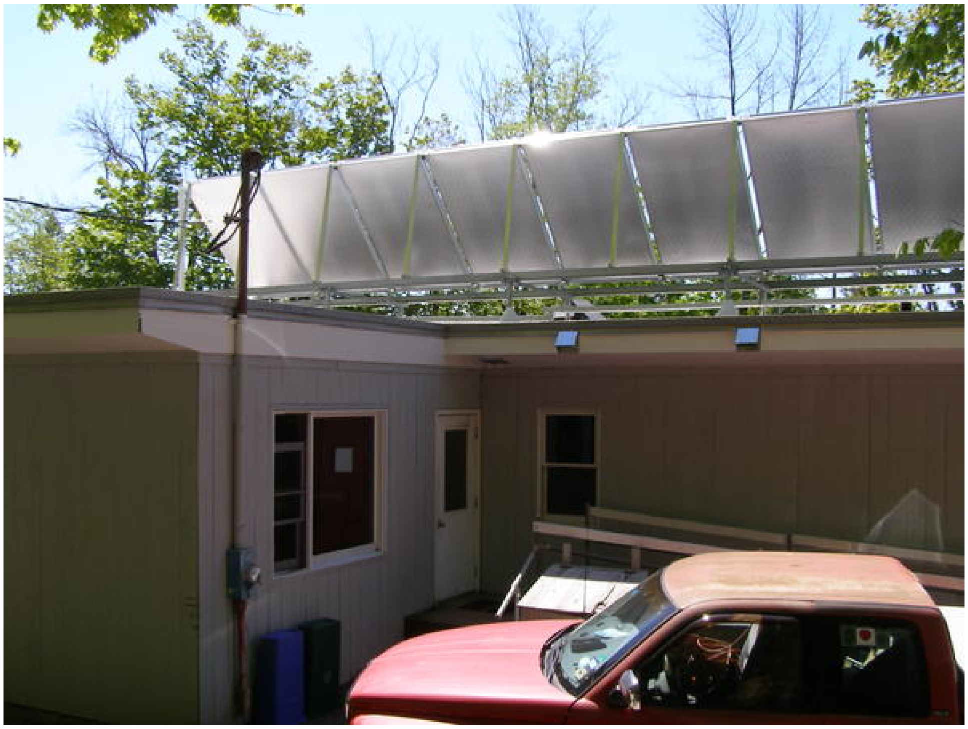
Power pole just east of Dalrymple, with main wires south and east and a lower-voltage wire to Dalrymple.

The cable to Dalrymple also continues on through the woods to Howland.

(continued)

The path from Dalrymple to Howland. The satellite TV dish is visible here. It is not connected most of the time. It has been used for watching special events such as Presidential debates or inaugurations.)

The cable reaches Howland.



Howland's solar hot water panels. Dan Cotter is deservedly proud of them.

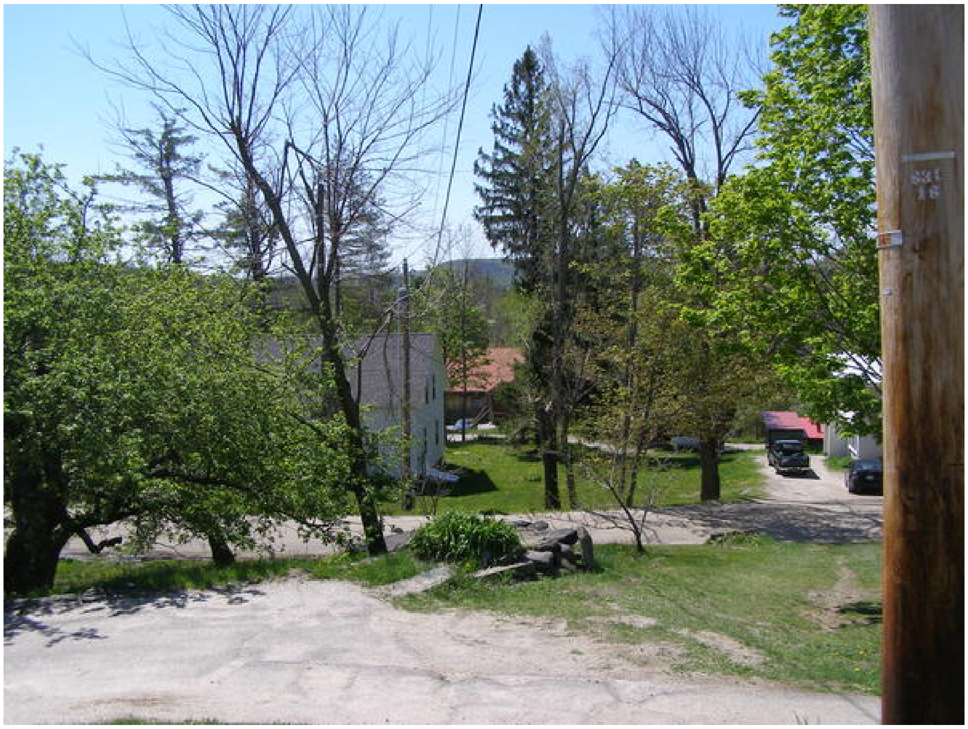
At end of year, dumpsters appear to give room for students to throw away everything they're not keeping when they move out. At Marlboro, students do not keep the same room from year to year.

It is good weather for motorcycling.

Generator/pump shed between Howland and Marlboro Gardens.

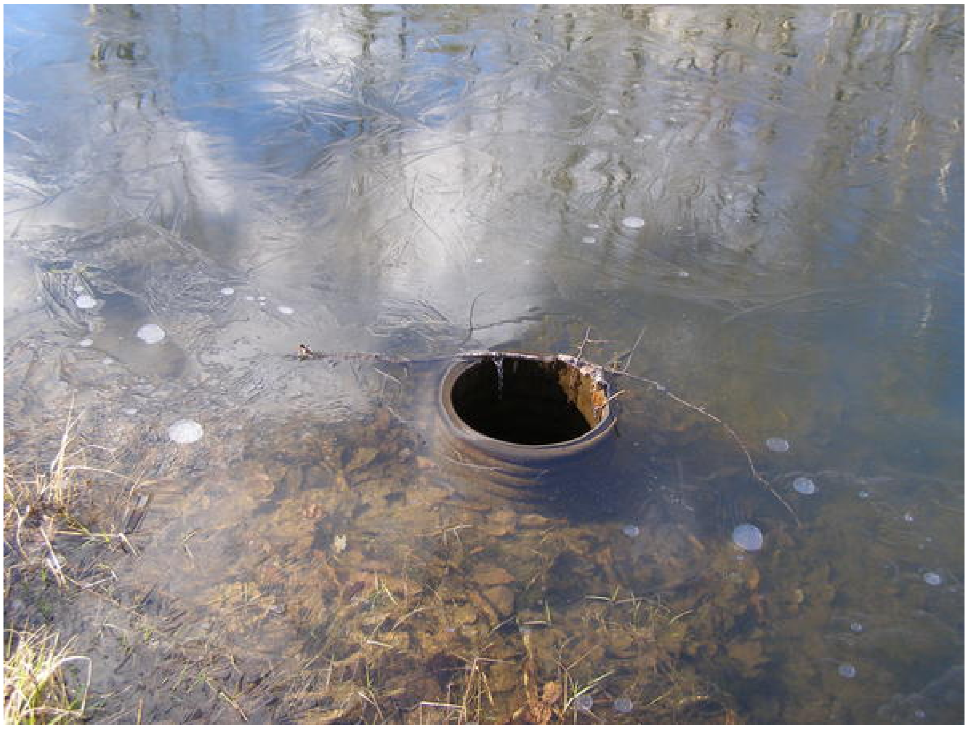
Propane tank nearby.

South of Marlboro Gardens, facing south, power poles lead from Moss Hollow Road (the unpaved continuation of South Road west of the college).



Under the pole east of Dalrymple; the wires continue south between Hendricks (mid-left) and Halfway (right) towards Woodard (rear).

These pictures were taken December 2012



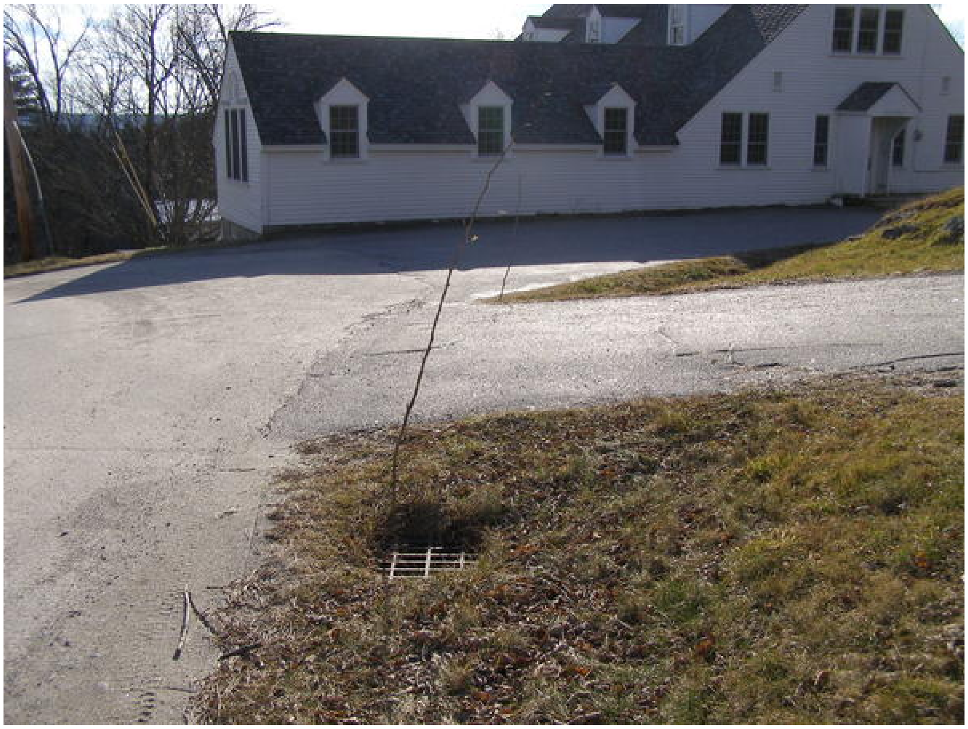
The “Fire Pond” has a drain that goes under the walking-path beside the pond. Occasionally in high rain the pond overflows over the path.

Facing downhill (south) from the Fire Pond, watching its effluence.

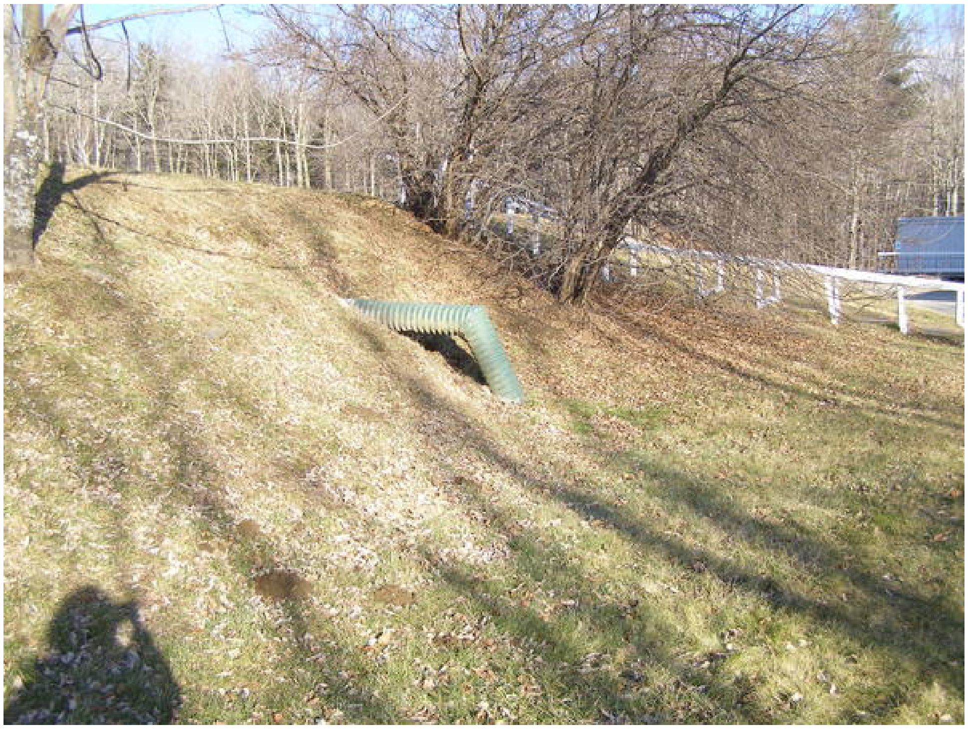
Facing the inlet to the Fire Pond (facing north).

Facing the inlet to the Fire Pond.

Facing the inlet to the Fire Pond.



Facing south towards Dalrymple, a grate helps water flow under the road. This makes the road erode less.



The south side of the soccer field (Zimmerman Field) has a visible drainage pipe. (Facing east-north-east.)

The north side of the soccer field (Zimmerman Field), where the drainage pipe begins. (Facing north.)

Part IV

Computer Science Exams

Each of these exams was a one-week long take-home exam. The exams permitted using books or the Web, but any sources looked at must be cited and it is more impressive to answer without needing external sources. The exams forbid discussing with other people.

As usual, after I submitted them, I had realizations and regrets.

When comparing algorithm efficiency, I could have put more weight or analysis on real-world cache and branch-prediction efficiency; it is for those reasons, after all, that my arguments against hash tables are not universally adhered to.

After submitting the Linux exam, I realized a scarier way to abuse having control of CS, which I decline to write here because it wouldn't count towards my exam, only towards attackers who read Plans. People who are in Jim's CS classes and read man-pages are likely to rediscover it anyway.

All in all, I'm pretty satisfied with these exams.

The original textual submission is in an addendum; for this printed copy I added section numbers, syntax hilighting, LaTeX-formatting, and the like, and added the questions as presented by Jim Mahoney, but did not change any words or substantive symbols. I wrapped code lines only where necessary for them to fit on the printed page.

22 Algorithms exam

```
computer science plan exam : algorithms
============================================

  * for Isaac Dupree from Jim Mahoney
  * out: Wed Oct 10 2012
  * due: Wed Oct 17 2012 by midnight

This is open take-home exam: books or web sources are OK as long as
the problem doesn't set other constraints, you cite them explicitly,
and that they aren't a drop-in solution to the problem. However, the
more your answer is a summary of someone else's article, the less we
will be impressed. Don't ask other people for help. Don't just give a
numerical result, give an explanation.  Your job is to convince us you
understand this stuff.

So in terms of a grading rubric, what you should think about is
 * technical merit - correctness & thoroughness of response
 * clarity of expression (including docs & tests for code)
 * demonstration of understanding (vs summarizing other's work)

As always with my exams, if you think there's a mistake in one of the
questions or it doesn't make sense, you can (a) ask for clarification,
and/or (b) make and state an explicit interpretation and do the
problem that way.  (Again: the point is to show your mastery,
not to get the "right answer" per se.)

Good luck.
```

22.1 implementation and complexity

question 1 (35 of 100 points)

Pick any two of the following three problems to analyze:

(i) sorting a list of numbers

(ii) the partition problem: given a multiset (a set-like construction that allows multiple copies of the same thing) of integers, is there a way to partition it into two parts such that the sum of integers in each part is the same?

(iii) The convex hull problem: given a set of points (x[i], y[i]), find that list of points that's on the extreme outer "edge", which form a convex polygon containing all the other points.

For each of the two problems that you choose, pick an algorithm to examine and answer the following questions. (In order of decreasing impressiveness, you may choose a good one that you already know and understand, invent one, or search the literate. In any case, be explicit about where your algorithm came from, and explain it clearly.)

(a) Describe what you expect the O() run time to be for the algorithms, and explain why and and on explicitly what type of problems (e.g. worse case, average case, etc).

(b) Implement and test the algorithms in a language of your choice. (Do *not* use built-in or library routines for e.g. sorting.)

(c) Run your code on various size data sets that you generate, recording the number of steps (however you choose to define that) the algorithms take to run. Show explicitly with a plot of the data from this "numerical experiment" that the O() behavior is as expected.

22.1.i sorting a list of numbers (Python)

```
#!/usr/bin/env python

# Everything in this file is from my memory, aside from the
# info about Python's built-in sorted() which I researched after
# writing everything else here.


# Worst case, everything is on the same side of the pivot every time.
# Then we compare this many times: (n-1) + (n-2) + (n-3) + ... + 0
# which has n terms and is O(n^2).
#
# Best case, we split the list in half each time.  Then we compare
# this many times: (n-1) + ((n-1)/2 - 1)*2 + ... =
#                  (n-1) + (n-3) + ((n-3)/4 - 1)*4 + ... =
#                  (n-1) + (n-3) + (n-7) + ... =
#                  (n - (2^1 - 1)) + (n - (2^2 - 1)) + (n - (2^3 - 1)) ...
# This series has O(log n) terms and is O(n log n).
#
# Average case, if we assume* the pivot is equally likely to be anywhere
# in the ordering, we split the list in a 1/4--3/4 or more-balanced split
# half of the time.  Such a split reduces log_2 n by at least 1/2 (compare
# to the best case 1+delta and worst case 0+delta, with delta the effect
# of removing one pivot element, which is small for large n).  So on average
# it is at least half the time at least half as good as the best case:
# that's only a constant factor of 4 worse than the best case, so it's
# O(n log n) average case.
# (The constant factor is smaller than 4 because many of the better splits
# are better than 1/4--3/4 and many of the worse splits are better than 0--1,
# but I didn't do the math to find out exactly what the factor is.)
#
# * or if we were to select which item is the pivot using a true random
# number, which would make our assumption guaranteed true (but add a large
# enough constant factor that we'd rather use another sort algorithm
# that is worst-case O(n log n) anyway).


# Data collected:
# % python --version
# Python 3.2.3
# (the program also works in python 2 unmodified).
# time python 1i.py
# python 1i.py  0.07s user 0.01s system 96% cpu 0.080 total
# I'll look at the user time.  So there's a 0.07 s constant overhead.
# I'll subtract that from each of these measured times before noting it.
#
# `time python 1i.py 10000`, for example.  I'm running each three times
# and picking the middle time value (not the most perfect but not terrible
# for an exercise).
#  10000: 0.17 s
#  20000: 0.37 s
#  40000: 0.76 s
#  80000: 1.73 s
# 160000: 3.51 s
# 320000: 7.38 s
#
# With each doubling of input size (with randomly ordered inputs),
# the runtime slightly more than doubles.  That's consistent with n log n.
# (Cache sizes and other things can screw up apparent trends, but that
# seems not to have happened too dramatically in this sample, for better or
# for worse.)
#
# Let's try the worst-case!
# `time python 1i.py 200 bad`, for example.  I'll subtract the same 0.07s
# from each.
# 200: 0.02 s
# 400: 0.06 s
# 800: 0.22 s
# As expected, each doubling is roughly multiplying the runtime by four
# (quadratic increase).  Python crashes for higher numbers than 1000 because
# of recursion depth (I think this is easy to adjust, but these numbers are
# decent enough for exposition).


def non_inplace_quicksort(items):
    """
    This sorts lists.  Use sorted(list) instead if you
    are actually programming Python, because it has the exact
    same effect and is part of the language.  Like this
    sort implementation, it is a stable sort (as of Python 2.2
    http://wiki.python.org/moin/HowTo/Sorting#Sort_Stability_and_Complex_Sorts
    ).  Unlike this sort implementation, it is faster because
    it is written in C and had a lot of work put into its performance (
    http://weblog.hotales.org/cgi-bin/weblog/nb.cgi/view/python/2002/08/03/0
    https://en.wikipedia.org/wiki/Timsort
    ).

    This version is not in-place, but it is not too much harder
    to write an in-place version (one which does not allocate
    memory, aside from the stack).  It would involve swapping
    the order of elements rather than creating new lists
    to put them in.

    >>> non_inplace_quicksort([1, 2, 3])
    [1, 2, 3]
    >>> non_inplace_quicksort([2, 3, 2, 1])
    [1, 2, 2, 3]
    >>> non_inplace_quicksort([2, 3, 2, 1, 7, -3242])
    [-3242, 1, 2, 2, 3, 7]
    >>> non_inplace_quicksort([])
    []
    >>> non_inplace_quicksort([’God’])
    [’God’]
    """
    # ’if’ conditions and trivial returns are all O(1).
    if len(items) == 0:
        return []
    elif len(items) == 1:
        # lists are mutable in Python so we create a new list
        # to return, rather than return ’items’.
        return [items[0]]
    else:
        # choose an arbitrary element as the pivot, O(1).
        pivot = items[0]

        # Filtering/partitioning is O(n).
        # This could be more efficient by a constant factor by
        # iterating items once, not multiple times.
        prevs = list(filter(lambda x: x < pivot, items))
        nexts = list(filter(lambda x: x > pivot, items))
        # It’s not necessary to separate the equal ones into their
        # own category, but it makes this a stable sort
        # (elements that are equivalent by the equivalence relation
        # are not changed in their order in the list).
        # Other things being equal, stable sorts are nice,
        # (occasionally code requires a stable sort or produces
        # prettier results with one).  Sort functions are sometimes
        # unstable in order to be faster by a constant factor.
        # (Writing ’sort’ in python is probably the worst constant
        # factor here, though, assuming we’re using the CPython
        # interpreter!)
        eqs = list(filter(lambda x: x == pivot, items))

        # prevs and nexts each have at most one less item than items,
        # so this algorithm is guaranteed to terminate.
        # We don’t need to write ’pivot’ here explicitly because
        # it will be a part of ’eqs’.
        # list append is O(n).
        return non_inplace_quicksort(prevs) \
            + eqs \
            + non_inplace_quicksort(nexts)

if __name__ == "__main__":
    import doctest
    doctest.testmod()

    print("(Usage: python 1i.py [test-case-size-for-timing [bad]]\n\
  ’bad’ means sorted data, which is worst for quicksort.)\n")
    import sys
    if len(sys.argv) > 1:
        testSize = int(sys.argv[1])
        bad = False
        testList = list(range(testSize))
        if not (len(sys.argv) > 2 and sys.argv[2] == "bad"):
            import random
            random.shuffle(testList)
        result = non_inplace_quicksort(testList)
        if testSize < 150:
            print(testList)
            print(result)
```

22.1.ii the partition problem (not answered)

22.1.iii the convex hull problem (Haskell)

```
{-

I invented this algorithm using my head.

O(n * v) where n is the the number of points provided and v is
the number of vertices in the convex polygon.

This could be made average-case O(n log n) (n = provided points)
by using a quadtree instead of linear search in nextPoint.

To test and run:
* have a GHC/Haskell-platform environment
* % cabal install blank-canvas
* % cabal install doctest
* if necessary, % cabal install random
* % ~/.cabal/bin/doctest 1iii.hs
* % runghc 1iii.hs
* after it starts running,
  it will print a debug message saying the polygon vertices.
  Then open a browser and go to localhost:3000 to look at the polygon.
Because I was lazy, it can only be tweaked by the CONFIGURATION PARAMETERS
below in this file.  Also, it generates points randomly distributed in a
rectangle, which isn't optimal for being able to tweak the number of
vertices of the resulting convex polygon (distribution within a circular
disk, for example, would be better for that particular purpose).  It was
amusing to see the runtime increasing quadratically for a little while as
I'd increased the number of points, and then decreasing because I'd had
testWidth small compared to the number of points, and when you fill up a
square area the polygon is just a quadrilateral!

I decided to not risk anything to inexactness and used Haskell's
arbitrary-precision rational arithmetic (which is implemented with
the GMP library) and avoided trigonometry functions (which produce
nonrational results).  I've done spatial algorithms like this before
and I know it's quite practical to do with fixed-size ints and doing
the algebra to get rid of actual rational arithmetic, I just didn't
do that here - it shouldn't affect the asymptotic performance since
the rationals are not growing to unbounded height (rational height
```

```haskell
refers to max(log(numerator), log(denominator))).

Basically it starts with the bottom left point, which must be on the
polygon, then goes in a circle starting from the angle of the previous
edge (or off to the middle of nowhere to start off), and finds the
point (by brute force) that has the least angle-rotation from the
current point and previous edge angle, and draws (concludes) an edge
between those, then moves on to that point, till it gets back to the
point where it started.

-}

module Main(main) where

import Graphics.Blank
import Control.Monad
import Data.List
import System.Random
import qualified Data.Set as Set

import Debug.Trace

-- CONFIGURATION PARAMETERS
staticTest = False
smallTest = False
testWidth = if smallTest then 99 else 9999
extraPoints = if smallTest then 0 else 1000


testPoints :: [Point]
testPoints = [(3,6), (7,7), (4,4), (2,8), (6,6), (8,2), (3,4), (2.5, 6)]

-- TODO we could avoid duplicating points (duplicates happen not
-- to hurt though).
generateTestPoints :: (RandomGen g) => Int -> g -> ([Point], g)
generateTestPoints numPoints gen = let
    (gen1, gen2) = split gen
    coords = map toRational (randomRs (1,testWidth :: Integer) gen1)
    (coords1, coords2) = splitAt numPoints coords
    points = zip coords1 coords2
    in (points, gen2)
```

```
getPoints :: IO ()
getPoints
    | staticTest = return testPoints
    | otherwise = do
        numPoints <- randomRIO (3,30)
        ourTestPoints <- getStdRandom (generateTestPoints (numPoints+extraPoints))
        return ourTestPoints

main :: IO ()
main = do
    ourTestPoints <- getPoints
    let poly = findConvexPolygon ourTestPoints
    print poly
    blankCanvas 3000 $ \context -> do
        print "hi"
        send context $ do
            drawPolygonFrom ourTestPoints poly
{-
        moveTo (50,50)
        lineTo (200,100)
        lineWidth 10
        strokeStyle "red"
        stroke ()-}



drawPolygonFrom :: [Point] -> [Point] -> Canvas ()
drawPolygonFrom points poly = do
    let polySet = Set.fromList poly
    let edges = zip poly (tail (cycle poly))
    let minx = minimum (map fst poly)
    let miny = minimum (map snd poly)
    let maxx = maximum (map fst poly)
    let maxy = maximum (map snd poly)
    -- Negate y because Canvas thinks that +y is downward and math thinks
    -- +y is upward.  Displaying in the math way makes things clearer.
    let pointToDisplay (x,y) =
        (realToFrac ((x - minx) / (maxx-minx) * 200),
         realToFrac ((miny - y) / (miny-maxy) * 200))
    forM_ edges $ \ (a, b) -> do
        beginPath ()
        moveTo (pointToDisplay a)
        lineTo (pointToDisplay b)
        lineWidth 5
        strokeStyle "blue"
        stroke ()
    forM_ points $ \p -> do
        let (x,y) = pointToDisplay p
        let ptsize = if Set.member p polySet then 6 else 3
        beginPath ()
        moveTo (x, y + ptsize)
        lineTo (x + ptsize, y)
        lineTo (x, y - ptsize)
        lineTo (x - ptsize, y)
        fillStyle "red"
        fill ()


-- Haskell's Rational does arbitrary-precision rational arithmetic
type Coord = Rational
type Point = (Coord, Coord)
type Offset = (Coord, Coord)
--data Slope = Slope
--    Integer --numerator
--    Integer --denominator (zero for vertical lines)
--data DirectedPoint = DirectedPoint Point Offset
--data QuadTreeOfPoints = None | Point Point | Quad Q Q Q Q

-- lexicographic, lowest first
--sortedPoints = sort points
--pointSet = Set.fromList points
--firstPoint = head sortedPoints


{-
--we arbitrarily always go counterclockwise on the
--mathematical coordinate plane
--So we need a way to compare two offsets given a third base offset angle
--(farther winning ties in this case, for whichever meaning is winning)
nextPoint :: DirectedPoint -> Point
-- the next point's direction will be (dest - src)
nextPoint =
-}
```

```
-- avoid trigonometry so that we never give the wrong answer from
-- rounding error
data Angle
    = XPlus Rational  --slope
    | YPlus
    | XMinus Rational --slope
    | YMinus
    deriving (Eq, Ord, Show)
type Ray = (Point, Angle)

-- | converts an offset to the angle (represented using line slopes)
-- that it can represent.
--
-- >>> offsetAngle (2,3)
-- XPlus (3 % 2)
-- >>> offsetAngle (2, -3)
-- XPlus ((-3) % 2)
-- >>> offsetAngle (-2, -3)
-- XMinus (3 % 2)
-- >>> offsetAngle (-2, 3)
-- XMinus ((-3) % 2)
-- >>> offsetAngle (0, -3)
-- YMinus
-- >>> offsetAngle (0, 3)
-- YPlus
-- >>> offsetAngle (0, 0)
-- *** Exception: offset is not an angle
offsetAngle :: Offset -> Angle
offsetAngle (x, y) = case compare x 0 of
    LT -> XMinus (y / x)
    GT -> XPlus (y / x)
    EQ -> case compare y 0 of
        LT -> YMinus
        GT -> YPlus
        EQ -> error "offset is not an angle"

pointOffset :: Point -> Point -> Offset
pointOffset (srcx, srcy) (dstx, dsty) = (dstx - srcx, dsty - srcy)

-- | cyclicLessThan changes the least (e.g. negative-infinity) point
-- of an ordered type.  For example, 'cyclicLessThan 0' makes negative
-- numbers greater than positive numbers and otherwise preserves
-- the ordering.
--
-- https://en.wikipedia.org/wiki/Cyclic_order (exam note: I invented
-- this function without the Internet and then decided it would be better
-- documented if I found out something mathematics would call it).
--
-- >>> cyclicLessThan 1 2 3
-- True
-- >>> cyclicLessThan 1 3 0
-- True
-- >>> cyclicLessThan 1 0 3
-- False
cyclicLessThan :: (Ord a) => a -> a -> a -> Bool
cyclicLessThan lowest a b = if (a < lowest) == (b < lowest)
    then a < b
    else (b < lowest)

-- | squared to avoid rounding error from sqrt
--
-- >>> offsetDistanceSquared (2, -3)
-- 13 % 1
offsetDistanceSquared :: Offset -> Coord
offsetDistanceSquared (x, y) = x*x + y*y

pointDistanceSquared :: Point -> Point -> Coord
pointDistanceSquared a b = offsetDistanceSquared (pointOffset a b)

-- |
--
-- >>> comparePointsInCounterclockwisenessFromRay ((1,1), XPlus (3/4)) (1,2) (2,100)
-- GT
comparePointsInCounterclockwisenessFromRay :: Ray -> Point -> Point -> Ordering
comparePointsInCounterclockwisenessFromRay (base, angle) point1 point2 = let
    offset1 = pointOffset base point1
    offset2 = pointOffset base point2
    angle1 = offsetAngle offset1
    angle2 = offsetAngle offset2
    in if angle1 == angle2
        then EQ
        else case cyclicLessThan angle angle1 angle2 of
            True -> LT
            False -> GT
```

```haskell
-- | less is better
--
-- >>> betterNextPolygonPoint ((1,1), XPlus (3/4)) (1,2) (2,100)
-- GT
-- >>> betterNextPolygonPoint ((1,1), XPlus (3/4)) (1,2) (1,100)
-- GT
-- >>> betterNextPolygonPoint ((1,1), XPlus (3/4)) (1,2) (0,100)
-- LT
betterNextPolygonPoint :: Ray -> Point -> Point -> Ordering
betterNextPolygonPoint ray@(base, _angle) point1 point2 =
    case comparePointsInCounterclockwisenessFromRay ray point1 point2 of
    LT -> LT
    GT -> GT
    EQ -> -- greater distance is better here, so that our polygon doesn't
        -- have two consecutive collinear edges, so compare in the
        -- opposite order from usual.
        compare
            (pointDistanceSquared base point2)
            (pointDistanceSquared base point1)


-- for ties, picks the furthest away one
-- goes (counter?)clockwise.
-- obvs never picks the same point?
nextPoint :: [Point] -> Point -> Angle -> Point
nextPoint allPoints currentPoint priorDirection =
    minimumBy
        (betterNextPolygonPoint (currentPoint, priorDirection))
        (filter (/= currentPoint) allPoints)

-- Lowest x, with lowest y for ties, will always be one of the
-- polygon vertices.
--
-- O(n)
firstPoint :: [Point] -> Point
firstPoint points = minimum points

-- Starting from the furthest left (and bottom) point,
-- the exact direction doesn't matter too much;
-- it can be anything not pointing into the field of points.
firstDirection :: Angle
firstDirection = XMinus 0

-- TODO What about non-polygons? (e.g. < 3 sides)
findConvexPolygon :: [Point] -> [Point]
findConvexPolygon allPoints = let
    firstVertex = firstPoint allPoints
    go verticesSoFar currentVertex priorDirection =
        --trace (show ("yo!", currentVertex, priorDirection)) $
        let
        nextVertex = nextPoint allPoints currentVertex priorDirection
        in if nextVertex == firstVertex
        then verticesSoFar
        else go (nextVertex : verticesSoFar) nextVertex
            (offsetAngle (pointOffset currentVertex nextVertex))
    in go [firstVertex] firstVertex firstDirection
```

22.2 trees

```
question 2 (20 of 100 points)
-----------------------------

Now let's look at trees. In a language of your choice :

(a) Generate and store in a data structure a binary tree of coin flips
to a given depth N, so that for example (Heads Tails Heads Heads Tails)
is a node at depth 5.

(b) Implement both a depth-first and breadth-first tree search
of this tree for a given sequence of heads and tails. Is either of
these types of tree searches better than the other for this problem?

(c) What is the O(n) runtime of (a) and (b)? Give an explanation
and show that your code has this behavioir.

(You don't need to write code for problems 3 through 5,
though you can if you wish to.)
```

22.2.1 response (C++)

```cpp
// To compile: g++ 2.cpp -o 2
// To run: ./2 [depth]
// (it just runs the test that's in main() with a tree of depth [depth] or 5,
// and wastes potentially a lot of time (which you can measure if you use
// 'time' or such), and finishes.)
//
// See the comment above breadthOrDepthFirstSearch() for complexity analysis.
// There is a main() function, but I did not do enough testing on this code
// (you should grade it less for that if good code is part of this grade).
//
// I first tried writing it in SML but I didn't know that language well enough
// to do that efficiently or its online docs weren't good enough (see
// rationale and attempt in 2.sml).


#include <stdlib.h>
```

```
#include <list>
#include <vector>
#include <algorithm>

// This is a horrid mash of C and C++.  I do not even bother to have code freeing
// the memory allocated for the tree.  I use comments to note C++ implicit
// behaviors.

enum CoinFlip { Heads, Tails };

// If we don't store any data in this tree, then the entire large
// tree might as well just be represented by one int (its depth)
// and if it were an abstract data type, no one could tell!
// Or by something similar to TreeWithHistory below.
struct CoinFlipTree {
    // These pointers are both null if at a leaf of the tree,
    // or both nonnull otherwise.
    CoinFlipTree* ifHeads;
    CoinFlipTree* ifTails;
};

CoinFlipTree* generateCoinFlipTree(unsigned int depth) {
    if(depth == 0) {
        CoinFlipTree* tree = (CoinFlipTree*)malloc(sizeof(CoinFlipTree));
        tree->ifHeads = NULL;
        tree->ifTails = NULL;
        return tree;
    }
    else {
        CoinFlipTree* tree = (CoinFlipTree*)malloc(sizeof(CoinFlipTree));
        tree->ifHeads = generateCoinFlipTree(depth - 1);
        tree->ifTails = generateCoinFlipTree(depth - 1);
        return tree;
    }
}

// This is actually a problem that we can search without thinking about
// a frontier at all, because there is only one correct path that we can
// find out rather trivially.
//
// There is no reason this uses recursion; if I was thinking like
// an imperative programmer, or was optimizing C/C++ performance, or was
// worried about large arrays causing stack overflow, I would make it
// a loop instead.
CoinFlipTree* obviousSearch(CoinFlipTree* tree,
        unsigned int numCoinFlips, CoinFlip* coinFlipsArray) {
    if(numCoinFlips == 0) {
        return tree;
    }
    else if(tree == NULL) {
        return NULL;
    }
    else {
        CoinFlipTree* subtree;
        if(coinFlipsArray[0] == Heads) {
            subtree = tree->ifHeads;
        }
        else {
            subtree = tree->ifTails;
        }
        return obviousSearch(subtree, numCoinFlips - 1, coinFlipsArray + 1);
    }
}

// It's easier to think about all the state implied by these algorithms if
// we eschew recursion.  I'm using C++ STL libraries for this internal state
// because I know them and explicit memory management would obfuscate this
// code.

//
struct TreeWithHistory {
    CoinFlipTree* tree;
    std::vector<CoinFlip> history; //implicitly inited to []
};

// Because C++ is too sad to be able to have even vector deep equality
// built in.
bool history_equal(std::vector<CoinFlip> const& history,
        unsigned int numCoinFlips, CoinFlip* coinFlipsArray) {
    if(numCoinFlips != history.size()) {
        return false;
    }
    return std::equal(history.begin(), history.end(), coinFlipsArray);
}
```

```
bool history_is_a_prefix(std::vector<CoinFlip> const& history,
        unsigned int numCoinFlips, CoinFlip* coinFlipsArray) {
    if(numCoinFlips < history.size()) {
        return false;
    }
    return std::equal(history.begin(), history.end(), coinFlipsArray);
}

enum BreadthOrDepthFirst {
    BreadthFirst, DepthFirst
};

// stupid == true: a pure breadth first or depth first search.
// If n is the number of coin flips, both breadth and depth first will
// search O(2^n) nodes on average, although depth-first has an unlikely
// best-case time of O(n).  Because of how this search checks whether it's
// found the right node, average case time is O(n*2^n).
// (If n represents the number of items in the tree, then the average case
// time is O(n) or rather O(n log n).)
//
// stupid == false:
// We prune nodes from being searched farther if we can tell they're
// wrong by the time we see them.
// If n is the number of coin flips, both breadth and depth first will
// search O(n) nodes before finding the right answer (and then check that
// it is the right answer at mere O(n) cost).  If n is the number
// of items in the tree, they will search O(log n) nodes.
CoinFlipTree* breadthOrDepthFirstSearch(
        CoinFlipTree* tree,
        BreadthOrDepthFirst searchOrder,
        bool stupid,
        unsigned int numCoinFlips, CoinFlip* coinFlipsArray) {
           // implicitly inited to [];
    // used as a FIFO (breadth-first) or LIFO (depth-first) queue.
    std::list<TreeWithHistory> frontier;
    TreeWithHistory initial;
    initial.tree = tree;
    frontier.push_back(initial);

    TreeWithHistory current = initial;
    while(!history_equal(current.history, numCoinFlips, coinFlipsArray)) {
        if(stupid || history_is_a_prefix(current.history,
                                        numCoinFlips, coinFlipsArray)) {
            if(current.tree != NULL) {
                TreeWithHistory next1 = current; //deep copies ’history’
                next1.tree = current.tree->ifHeads;
                next1.history.push_back(Heads);
                frontier.push_back(next1);

                TreeWithHistory next2 = current; //deep copies ’history’
                next2.tree = current.tree->ifTails;
                next2.history.push_back(Tails);
                frontier.push_back(next2);
            }
        }
        if(frontier.empty()) {
            return NULL;
        }
        if(searchOrder == BreadthFirst) {
            current = frontier.front();
            frontier.pop_front();
        }
        else {
            current = frontier.back();
            frontier.pop_back();
        }
    }
    return current.tree;
}


int main(int argc, char* argv[]) {
    long depth = 5;
    if(argc >= 2) {
        depth = strtol(argv[1], NULL, 10);
    }
    CoinFlipTree* tree = generateCoinFlipTree(depth);
    std::vector<CoinFlip> coinFlipsArrayToSearchFor;
    for(int i = 0; i != depth; ++i) {
        // check some arbitrary bit of rand()
        if(rand()&(1<<13)) {
            coinFlipsArrayToSearchFor.push_back(Heads);
        }
        else {
            coinFlipsArrayToSearchFor.push_back(Tails);
        }
    }
    BreadthOrDepthFirst searchOrder = BreadthFirst;
    bool stupid = true;
    breadthOrDepthFirstSearch(tree, searchOrder, stupid, depth,
                              &coinFlipsArrayToSearchFor[0]);
    return 0;
}
```

22.2.2 (incomplete SML version)

```
(*

(See 2.cpp - I gave up on using ML)


I chose ML for this even though I don't know the language well because it is:

(A) statically typed with algebraic data types (discriminated unions and
      aggregates).  It's easier to talk about types (which is the purpose of a
      data structures exam question) when the language provides a precise way
      to talk about them!
(B) strictly evaluated.  Discussing and measuring asymptotic complexity
      is a bit more complicated in Haskell because something might be
      evaluated partially but not entirely.

......alright I should probably use C/C++ because I am struggling with this
language too much.  Or Haskell
*)


print "Hello world!\n";

datatype CoinFlip = Heads | Tails ;;

datatype CoinFlipTree
  = Leaf
  (* left is heads, right is tails *)
  | Node of CoinFlipTree * CoinFlipTree
  ;;
```

```
fun generateCoinFlipTree 0 = Leaf
  | generateCoinFlipTree depth =
    let val subtree = generateCoinFlipTree (depth-1) in
      Node (subtree, subtree)
    end
  ;;

(* obviousSearch : CoinFlip list * CoinFlipTree -> CoinFlipTree option *)
fun obviousSearch (nil, Leaf) = SOME Leaf
  | obviousSearch ((Heads::xs), (Node (tree, _))) = obviousSearch (xs, tree)
  | obviousSearch ((Tails::xs), (Node (_, tree))) = obviousSearch (xs, tree)
  | obviousSearch (_, _) = NONE
  ;;

print (case (generateCoinFlipTree 3) of
            Leaf => "Leaf!\n" | Node (_, _) => "Node!\n");

print (case (obviousSearch (generateCoinFlipTree 3) of
            Leaf => "Leaf!\n" | Node (_, _) => "Node!\n");

(* but to be properly breadth-first or depth-first we need to
* consider and possibly re*)

  (* of 'a ?  The problem didn't ask for any data in the tree... *)
    (* If *)

print "Bye :)\n";
```

22.3 paradigms

```
question 3 (15 of 100 points)
-----------------------------

The task of finding a given string from an alphabetized list
can be solved by algorithms that fit into several
paradigms, including brute force and divide-and-conquer.

 * Given an example of an algorithm that fits each of these
 descriptions, each of these, and describe their O() behavior.

 * Can you describe a third algorithm design technique that can be
 applied to this problem, a representative algorithm, and its O()
 behavior?

 (For full credit, do this problem on your own. If you do need outside
 sources, as always be clear on which ones you used, and for which part
 of the problem.)
```

Response

Task:

$$\text{isMember} : (\text{String},\ \text{collection of Strings}) \to \text{Bool}$$

Suppose we start with the collection of strings as an unsorted linked list or similarly not-particularly-useful-for-this-purpose structure.

Brute force means trying every possibility (or every one until we find an answer). For example, iterate through the linked list until we find the string we're looking for or until we get to the end of the list. This is $O(n)$ in the size of the collection of strings, with the primitive operation that there is n of being string-equality. If the string is in the list (assuming no duplicates) it takes average $n/2$ time, and if it's not then it takes n time - both are $O(n)$ average and worst case. If it's present, there is $O(1)$ best case, but best case isn't very important except for things with special requirements on compute time like cryptography.

Divide-and-conquer (judging by reading Algorithm Design (1st ed., Jon Kleinberg, Éva Tardos)'s first paragraph about divide and conquer) is about dividing the problem into smaller parts (unless the part is trivially small), recursing into each part, and then combining the sub-results into a greater result. In this problem, one way we could apply divide-and-conquer is by preprocessing the collection of strings into a sorted array or binary search tree (an $O(n \log n)$ one-time cost). Then, suppose it's an array. Examine the middle element of the array. A stupid application of divide-and-conquer would be $O(n)$:

- if it's a 1-element array, return whether it equals the queried string.
- if it's larger, split it into the sub-array preceding and the sub-array at and after that middle element. Recurse into each. Combine the two boolean results with inclusive-or.

A smarter application of divide-and-conquer would use the value of the middle element. Is the string we're searching for lexicographically less than this middle element? We can short circuit one of the two recursive calls as false, because it is guaranteed not to have the string anywhere within it. This is $O(\log n)$ best, average, and worst case, with the basic operation being string lexicographic compare.

We can try to exploit the internal structure of strings. They are sequences of characters. We can put the collection of strings into a trie at $O(n * t)$ cost with n being the total number of characters in the collection of strings and t depending on the structure of these trie nodes (Array with length matching the size of the alphabet: $t = 1$. Hash table: $t = 1$ average case, $\min(\text{size of alphabet}, n)$ worst case. Binary search tree: $t = \log(\min(\text{size of alphabet}, n))$ worst case, possibly better average case in this particular usage. Etc.). Let's assume t is 1, because, say, taking the strings as bytes and using a 256-ary array-based trie is not the worst thing we could do. Then a lookup is $O(c*t)$ — with t=1, so $O(c)$ — where c is the number of characters in the string we're searching for, and there are no underlying operations to worry about (we know that e.g. byte comparison is constant time). Is that better than $O(\log n)$ with the fundamental operation being comparisons? Well, how long does a comparison take in terms of c? Best case $O(1)$, worst case $O(c)$. So $O(\log n)$ to $O(c \log n)$. (Actually, c should be $\min(c, \text{the worst length in the collection})$ for both the string comparison and the trie cases - minor detail). If a string is found, we must compare the whole thing at least once, so actually $O(c + \log n)$ to $O(c \log n)$. The former is worse iff $\log n$ is much larger than c; e.g. a large collection of short words. That iff can't happen. Suppose the alphabet is binary [0,1] and all words are 5 bits long. Then there are a maximum of 32 possible words. log 32 (in base 2, not that it matters for asymptotics) is 5. 5 is the number of characters (c). So $\log n$ can't really exceed c, so $O(c + \log n)$ is the same as $O(c)$ - there is at worst a constant factor involved. $O(c \log n)$, however, is by this reasoning as much as $O(c^2)$. Is the worst case also the average case? I suspect it depends how you define a probability distribution of all possible collections of words.

Hashing (hash tables) also exploit the internal structure of strings in their hash function. Average case $O(c)$, worst case $O(c * n)$.

We can use a heuristic approach. This is probably pretty stupid for string search, but here's one way it might go: As preparation, arrange the words into an undirected graph with edges drawn between ones with small Levenshtein distance between them. Ensure the graph is connected (relax the sense of "small" above to do this). To search, start somewhere in the graph. See if we've found the word here. If so, we're done. If not, look at all this node's neighbors. Which one has the least Levenshtein distance to the word we're looking for? Go to that node and repeat. That isn't a very good heuristic algorithm. It won't terminate if the string we're looking for isn't in the collection. I also haven't proved (and rather doubt) that Levenshtein distance has the right properties to make this algorithm terminate even in many cases where the word is somewhere in the collection. A better algorithm would remember where it's been and not go there. What if it gets stuck? It should probably use closed and open nodes like A* does (although it shouldn't use A* proper unless there is a good heuristic with the required property satisfied. Also we don't care about the path, only the goal, so we don't even need what A* is trying to achieve. It should probably keep an open set along with each of their Levenshtein distances from the search string in a heap and pick the closest one each time — a simple best-first search.). These better algorithms would terminate and have worst-case no worse than $O(n)$ and would always reach the worst case for strings that aren't in the collection, and probably have some better average-case time. I'm sure there are ways to improve the heuristic more. It still isn't a good choice for this problem.

[Academic integrity note: I skimmed some of the concepts I was using on Wikipedia to double-check whether I was using the words correctly. I was (or maybe I am still using them wrong – in any case, I didn't change my mind about anything here because of Wikipedia or any other source). Wikipedia also helped remind me of the name of Levenshtein distance, a concept I often think about but can never quite remember the name of. As noted above, I used an algorithms book to define “divide and conquer” for me.]

22.4 P and NP

```
question 4 (15 of 100 points)
-----------------------------

Explain in your own words what exactly is meant by "P" and "NP" as
complexity classes in computer science, why this is such an important
question, and what is and isn’t known about them. Give explicit
examples of problems that are in each of these classes, and explain
why the known algorithms have behaviors consistent with your P and NP
descriptions. You may use external sources if you need to - and if so
be clear which ones you used, of course - but the point here is to
convey to us your understanding, not to just summarize a wikipedia
article.
```

Response[162]

So, there are some problems[163] whose answers can be checked in polynomial time.

Let’s consider an easy problem which can be solved and checked in polynomial time, but checked more easily than it’s solved: “is this list sorted?”. This is easy to check in linear time, since comparison functions are (hopefully!) transitive. On the other hand, sorting a list tends to take $O(n \log n)$ time. Given only a less-than function, sorting a randomized list can’t do better than that on average, because it needs to do that many comparisons to even find out what order the list is in! There are $n!$ possible orderings. Each comparison only gives 1 bit of information. So there need to be about $\log_2(n!)$ comparisons, which is $O(n \log n)$.

In any case, sorting is in P, because P includes all the problems that can be solved in polynomial time. Polynomial time includes a lot of things. It includes anything asymptotically less than or equal to $O(n^C)$ with C constant. $O(n^{10000})$ may not be a practical-to-use

[162] This question’s response was submitted 40 minutes late.

[163] Technically, P and NP refer to “decision problems” (boolean result), but it more or less works fine to pretend they work for “function problems” (arbitrary data result) too. I’m a computer programmer and I usually want more-than-boolean function results so I am (perhaps foolishly) pretending that.

algorithm, but it's in P. On the other hand, $O(2^n)$ is not in P. There are expressions that are superpolynomial but subexponential. Here are some $O()$s from small to large:

- $O(n^{100000})$
- $O(2^{\log n})$
- $O(2^{(n^{1/2})})$ (that's two to the square root of n)
- $O(2^n)$
- $O(n!)$
- $O(2^{(n^2)})$

The first one is in P, the rest aren't. [Those examples besides n^{100000}, 2^n, and $n!$ were indirectly inspired by Wikipedia.]

The rest can be, but are not necessarily, in NP. NP ("nondeterministic polynomial time") is the class of problems whose answers can be checked in polynomial time. Obviously, any problem which can be solved in polynomial time can also be checked that fast (by solving it): NP includes P. NP does not include all problems.[164] NP probably is not equal to P, but we haven't proved this.

(Side note that I'm not sure is correct: Cryptography's current foundations would crumble if P = NP. It depends on problems that are much easier to verify than to find an answer that you'd verify. Anyway, quantum computing has different abilities because the universe

[164] I don't know of any problems that are not in NP, though. Wikipedia `https://en.wikipedia.org/wiki/PSPACE-complete` lists some, unless it turns out that PSPACE = P or some such unexpected conclusion. According to `https://en.wikipedia.org/wiki/EXPSPACE` deciding whether two regexps are equivalent is EXPSPACE-complete, and according to `https://en.wikipedia.org/wiki/EXPTIME` EXPSPACE is a strict superset of PSPACE which is a not-proven-to-be-strict superset of NP, and EXPSPACE-complete is the hardest problems in EXPSPACE, so that problem isn't in NP, and it is something that sounds useful to a computer programmer. In any case, problems not in NP are likely to be hard enough that we wouldn't want to try solving them with a computer, except perhaps on really small examples.

is weird. If it and its Shor's algorithm comes into force then cryptography will have to find foundations other than factoring integers anyway.)

There is a class “NP-complete” inside NP of problems, all equivalently hard (by a specific sense of equivalent that allows different problems to have polynomial differences in the $O()$ exponent or multiplying the exponentiation), which are the hardest problems in NP. Assuming P != NP, they cannot be solved in polynomial time (but can be checked in polynomial time; imagine 'sort' but on a much grander scale). Contrapositively, if an NP-complete problem is solved in polynomial time, we know that P = NP... and a lot of people and money have probably been thrown at this and this hasn't happened yet, as far as we know.

A famous NP-complete problem is the traveling salesman problem. Imagine you're a salesperson and you want to visit all of a given set of cities while doing the least traveling possible. It turns out that finding that answer is NP-complete. Also, the best known asymptotic for this problem is $O(2^n)$. Luckily for actual salespeople, finding an answer that's verifiably within 1of the best is in P [citation needed. I can't find where I got that idea. But anyway, solving a similar problem is often all you need to do for an actual real-life purpose].

NP-complete problems aren't completely intractable; small enough examples can be solved on computers, and a lot of work has been put into making them solve moderately large problems. For example, the traveling salesman problem has been solved perfectly for some examples in the 10000s of cities.[165] Also, real life data sets are sometimes more predictable than random counterparts of the same size in a way that lets heuristics solve them faster.

An NP-complete problem closer to my field is dependency resolution in Linux package management. Packages can have requirements like “depends on these other packages”, “depends on this or that – either will do”, “conflicts with this package – both cannot be installed at the same time”, etc. When the user asks to install a package, what other packages must

[165] `https://en.wikipedia.org/wiki/Travelling_salesman_problem#Exact_algorithm`

we install (and possibly uninstall!) to allow them to install this? Sometimes a solution does not exist. For an obvious example, if the user asked to install two conflicting packages at once. It gets complicated when it's the packages they depend on that might conflict. This is complicated in practice – Debian's apt-get, for example, has many heuristics for it. Recently Fedora integrated a SAT solver library into their package manager. SAT solvers attack an NP-complete problem (`https://en.wikipedia.org/wiki/Boolean_satisfiability_problem`). Since all NP-complete problems are equivalent, and SAT solvers have a lot of optimization and heuristic work already put into them, and SAT and dependency resolution are related enough that it's practical to translate one into the other, this change was good. It improved package-manager speed and improved its suggestions in response to complicated user installation demands.

Some problems are not polynomial and not NP-complete (assuming P != NP). Factoring integers into primes seems likely to be one of them. This would mean it is "harder" than every problem in P and "easier" than every NP-complete problem. I think this does not necessarily mean its asymptotic complexity is below every NP-complete problem, because of the way NP-complete equivalence does its thing. (For reference, integer factorization's current best time is $O(e^{n^{1/3} * (\log n)^{2/3}})$ with $n =$ the number of bits in the integer.) [Information from `https://en.wikipedia.org/wiki/Integer_factorization` - I already suspected this problem might be interesting but did not have the details till I checked Wikipedia.]

Knowing about NP-complete problems was useful while programming a graph algorithm last semester. I realized that one of the things we were trying to do could be used to find Hamiltonian cycles. Finding Hamiltonian cycles is proven NP-complete. Once I realized this, we decided not to try and find a better-than-exponential algorithm to do it, and instead spent our time on things more likely to be productive. (It turns out we were doing small enough problems that our naive factorial-time algorithm was plenty fast enough.)

[Academic integrity note: I read a lot of wikipedia articles mid-way through writing this

- P, NP, NP-complete, PSPACE, NP-hard, co-NP, integer factorization, traveling salesman problem, boolean satisfiability problem, decision problem, function problem, etc. Then I read most of chapter 8 of Algorithm Design (1st Edition) by Jon Kleinberg and Éva Tardos, “NP and Computational Intractibility”, and the new knowledge I gained from it was (1) a way to define a decision problem with checkable answers and (2) the way in which different NP-complete problems can have polynomial time differences between their answers.]

22.5 algorithm tradeoffs

```
question 5 (15 of 100 points)
-----------------------------

You're given a list of N names and told that you'll need to search
the list for a given name M times.  The following are suggested:

  * using a hash table
  * using a heap
  * sequential search
  * sorting followed by binary search

Discuss the time efficiency of these approaches as the size of N
and M vary.  Which one would you suggest and why?  Would your answer
change if you needed to change the list of names by adding and
deleting names (say, P times) between searches?
```

Response

We need to do this M times on the same or similar collections:

$$\text{isMember} : (\text{String},\ \text{collection of Strings}) \to \text{Bool}$$

with size of collection = N.

(Falsely) assuming constant-time ==, <, and hashing of strings, we have:

A hash table has average case insert/delete/lookup $O(1)$. So there is $O(N)$ setup time and $O(M)$ lookup time in all – $O(M + N)$. We can't do any better than that in average asymptotic time. Worst case it is $O(M * N + N * N)$. Unfortunately, this has been used maliciously in e.g. computational complexity based denial-of-service attacks against web servers that put HTTP headers in a hash table. This year most languages finally mitigated this well-known attack by putting some randomness into their hash function, but it's just a mitigation. Hash table hash functions demand a small constant-factor time cost, which distinguishes them from cryptographic hash functions. An attacker can still interact with a web server to infer what their random seed is, and then DOS it.

Heaps are array-based structures that can implement priority queues. They do fewer things than a balanced binary tree, but typically have better constant factors in time and space for the things they do do. They have a mix of $O(\log n)$ and $O(1)$ costs. I haven't come up with a way they're useful here.

Sequential search requires no setup but is $O(M * N)$ for lookups. If M is very small (e.g., one or maybe two lookups) then it would be a fine choice. If N is very small (e.g. 1-4 possible words) then sequential search is fine and may have a better constant factor than fancier approaches (simple code fits in instruction cache easily, has few instructions, the collection can be in a memory-contiguous array, and the search is branch-predicted successfully all but one times). But its asymptotic factors quickly catch up to the constant factors and make it a bad choice for even moderately large M-and-N. It easily has $O(1)$ addition to the collection (if we're not worried about the possibility of wasteful duplicates) and can have $O(1)$ or $O(N)$ removal depending how you define the problem and arrange your sequence.

Sorting followed by binary search is $O(N \log N)$ setup and $O(\log N)$ lookup. As a computer programmer, it's often pragmatic to look at $\log n$ as a (potentially somewhat large) constant factor and just measure how bad it is for the sorts of data sets we're using. This approach's worst case equals its average case, so it should probably be used more often for DOSable programs (anything that can read data from an outside source, basically) in place of hash tables.

Balanced binary search trees have $O(\log N)$ insert (so $O(N \log N)$ setup), and $O(\log N)$ lookup. They have worse constant factors in time and memory than sorting followed by binary search (due to 2 to 3 pointers per node), but they allow $O(\log N)$ insert and removal whereas a sorted array has $O(N)$ insertion and removal.

So far: Sequential search best for very small M or very small N. Hash tables best for non-malicious data used for non-life-critical operations; balanced binary search trees (or sorting followed by binary search if you don't need to modify it any more often than once every N lookups) best for data actively being attacked by smart people, or realtime systems (e.g. medical device or industry) where reliability is more important than average speed. That's as far as (farther than) most programmers look.

As discussed in the answer to question 3, tries may be better (possibly one of their many variations – Patricia tries, array-mapped tries, etc.). They are $O(c)$ average and worst case; compare to hash tables' $O(c)$ average case and $O(c * N)$ worst case. Asymptotically, we should obviously use them. Practically, they can be hard to get good memory usage and constant factor speed at the same time – but then, it's also easy to write a bad hash function if creating a hash table, or fail at a tree's self-balancing algorithm. The biggest practical reason they're used less is that they're not in standard libraries of popular languages, I think. (Incidentally, the same effect makes hash tables be used more than balanced binary search trees for associative collections. Many languages have hash tables built in to the language. Java and C++ support both equally well, but Perl/Python/Ruby/PHP/Lua/Javascript have built-in associative collections that are hash-table based.)

So far: tries win the theoretical battle on every count (besides small N or M).

What if we want to create multiple related collections? This is natural in purely functional programming, where you'd want "inserting" into a collection to give you a new collection with the new element while leaving any references to the original collection unmodified. Sometimes it is useful to have this ability for a particular programming task even in an imperative language (although you're unlikely to have a library in an imperative language that makes this practical, and it is most practical when in the presence of garbage collection). Hash tables can't do this. Balanced binary search trees can; Haskell's standard associative collection (Map from Data.Map) uses one and has all the same asymptotic complexities you'd expect from an imperative one without its limitation to destructive updates. Tries can too.

So, asymptotically anyway, tries still win, with balanced binary search trees coming in second except under more-particular circumstances.

[Academic integrity note: I checked Wikipedia for heap and trie to double-check that I meant the same things by them as they mean, and I did. Also, I wrote this after writing most of the answer to question 3.]

23 Languages exam

```
computer science plan exam : languages
============================================

  * for Isaac Dupree from Jim Mahoney
  * out: end of day  Thu Oct 25 2012 by midnight (barely)
  * due: end of day  Thu Nov  4 2012 by midnight[166]

This is open take-home exam: books or web sources are OK as long as
the problem doesn't set other constraints, you cite them explicitly,
and that they aren't a drop-in solution to the problem. However, the
more your answer is a summary of someone else's article, the less we
will be impressed. Don't ask other people for help. Don't just give a
numerical result, give an explanation.  Your job is to convince us you
understand this stuff.

So in terms of a grading rubric, what you should think about is
 * technical merit - correctness & thoroughness of response
 * clarity of expression (including docs & tests for code)
 * demonstration of understanding (vs summarizing other's work)

Be very explicit about which sources you used for each problem.

As always with my exams, if you think there's a mistake in one of the
questions or it doesn't make sense, you can (a) ask for clarification,
and/or (b) make and state an explicit interpretation and do the
problem that way.  (Again: the point is to show your mastery,
not to get the "right answer" per se.)

Good luck.
```

Response

I didn't refer to external sources for this exam except for the normal infinity of looking at documentation while coding. I think I didn't look at any documentation while writing 2A.hs. But it doesn't really make a difference (besides how fast I am from having memorized the

[166] Adjusted to midday Friday per email exchange regarding when the exam came out.

names and packages of functions in each language). I ran into some people on the Internet writing about things about languages this week, just like I do most weeks because I spend too much time reading tech stuff. That didn't substantially affect my essays.

23.1 describing buzzwords

```
question 1 (40%)
----------------

As a way to demonstrate your understanding of programming ideas,
discuss the concepts behind following programming buzz words across
languages you're comfortable with, at least (Haskell, C, C++, Javascript).

(a) First organize the terms into groups of concepts, showing which
    are variations of the same concept or idea across or within
    languages, or closely related concepts, or opposites.

(b) Then for each of these concept group, discuss the ideas behind
    that group, and give concrete code snips across these languages
    to illustrate them.

Be clear that I *don't* want you to just define each of these words;
instead, I want you to use them as the starting point for a
conversation with examples about some of the fundamential notions with
programming languages, and how those notions vary across languages.

In alphabetic order, the words are

      API
      argument
      bind
      callback
      class
      closure
      collection
      comment
      concurrent
      compiled
      data structure
      dynamic
      exception
      fork
      function
      functional
      global
```

```
hash
immutable
imperative
inheritance
interface
interpreted
iterate
lexical
link
list
lazy
macro
method
name
namespace
object
overload
scope
package
pattern
pass by reference
pass by value
recursion
side effect
stack overflow
static
symbol
syntactic sugar
thread
test
throw
type
variable
vector
```

Response

23.1.1 callback, closure, function, functional, lexical, scope, symbol, bind, name, variable, argument

The core concept of modern programming languages is the function.

Scheme:

named function:

```
(define (f x) (+ x 7))
```

unnamed:

```
(lambda (x) (+ x 7))
```

Haskell:

named:

```
f x = x + 7
```

unnamed:

```
\x -> x + 7
```

Lambda calculus:

unnamed (assuming “plus” is addition on, say, Church numerals):

```
λx.  plus x 7
```

named (sort of):

```
(λplus_seven.  [...])  (λx.  plus x 7)
```

C:

named:

```
int f(int x) {
   return x + 7;
}
```

unnamed:

well, it still has to have a name, but it *can* be used in an expression:

```
&f
```

C++:

named: same as C, or polymorphic:

```
template<typename Int>
Int f(Int x) {
   return x + 7;
}
```

unnamed (C++11):

```
[](int x) { return x + 7; }
```

(which can be emulated with structs...?)

Javascript:

named:

```
function f(x) { return x + 7; }
```

unnamed:

```
function(x) { return x + 7; }
```

In all of these, the variable “x” is in scope in the lexical (textual) body of the function. The addition operator, if it were written as a function with its own code which might use “x” to mean something else, would have no idea that “f” used the name “x”. If several threads called “f” at the same time, there would be several different “x”es at once (functions can be thread-safe), and if “f” called itself sometimes, then there would also be multiple “x”es at once (functions can be reentrant).

23.1.2 dynamic vs. lexical scope

Early implementations of Lisp used dynamic scope, in which the implementation of "+" could (according to the language, not necessarily according to good style) use the name "x" to refer to "f"'s "x". (Emacs Lisp is the last language to stick to dynamic scope). This behavior was easier to implement at the time, but it is universally recognized as broken (except for a few languages that allow one to explicitly have dynamically scoped variables in addition to the default of lexical scope).

The above behavior is weird. It breaks abstraction. There is often no good reason that a called function should have any access to the calling functions' arguments. The name chosen for an argument also shouldn't have any effect outside the text right there that defines the function.

But that's not the critical problem; that doesn't change the behavior of well-written programs. The critical problem pertains to closures. Consider:

```
function addSomethingToList(list, m) {
   return map(function(n) { return n + m }, list);
}
```

Here, the variable "m" is in the anonymous function's "closure". This is, most of the time, an implementation detail that implements how inner functions can refer to outer functions' variables.

(Functions that have closures can be given names, too; it's merely common that they're not. They, and all functions, can be returned from functions, stored in variables, passed to other functions, etc.)

Consider that map was:

```
function map(f, l) { ... }
```

Alright. But what if it happened to be:

```
function map(f, m) { ... }
```

In dynamic scoping, addSomethingToList calls map, which calls f (the anonymous function), which refers to m, and the nearest thing in the call stack named 'm' is the list bound to map's argument... which is the wrong thing! It's not even a number, it's a list! The answer we want, addSomethingToList's argument, is farther away in the dynamic call chain.

Worse, consider:

```
function addSomething(m) {
   return function(n) { return n + m };
}
```

to be used like addSomething(3)(4) or

```
addThree = addSomething(3);
...addThree(4)...
```

After addSomething returns, "m" won't be anywhere in the dynamic call stack, and the program cannot continue even in a wrong way!

Lexical scoping means that naming a variable always references the textually enclosing blocks of code and nothing else.

Some languages have wibbles to lexical scoping.

Python's wibble

In Python, global and nonlocal (closure) variables are fully lexical in a read-only way, but changing their value can require an explicit declaration inside the innermost function. This is because Python implicitly declares local variables:

```
def plusSeven(x):
   temp = x + 3
   return temp + 4
```

(What if someone adds a global variable named “temp” in a far away piece of code! We don’t want that to change plusSeven’s meaning.)

whereas many languages require an indication that a new variable is being bound, e.g. Lua uses “local”:

```
function plusSeven(x)
   local temp = x + 3
   return temp + 4
end
```

CoffeeScript has the same implicit variable declaration as Python, but throws caution to the winds and says that pure lexical scope is more important than the risk of a global variable or function named “temp”.

C++’s wibble

In some languages (e.g. C++), a method definition’s code can also refer to object member variables without a lexical indication of that variable’s name. In C++ this is mitigated by the fact that

- local lexically scoped variables will be chosen first if member and local variables have the same name (as is typical)
- method definitions state the name of the type they are a method of, or are defined lexically within the class declaration (which is not linguistically true in, say, Javascript or Lua)

- methods cannot be defined within methods (unlike Javascript or Lua)
- C++ programmers typically define a coding convention of how to name member variables (such as a prefix or suffix, like “m_” in “m_popCount”) for the sake of human readers of the code.

This is not to bash Javascript; the fact that its functions all implicitly bind “this” (unlike Lua) is a problem, but at least references to “this” require mentioning the word “this”, so Javascript is also tolerable.

But closures are not always an implementation detail!

```
function counter() {
   var x = 0;
   return function() {
     x = x + 1;
     return x;
   };
}

someCounter = counter();
anotherCounter = counter();
// Now someCounter and anotherCounter are the same... exactly the
// same... except for the fact that they're not the same thing.
someCounter(); // = 1
someCounter(); // = 2
someCounter(); // = 3
anotherCounter(); // = 1
```

When variables in closures are mutated, it is a way of giving each instance of a function its own private state. Therefore, you as the code-reader need to distinguish between different instances of otherwise identical functions. An easy way to do this is to think about closures explicitly. In Haskell, which does not have implicit mutation, one need never think about what a closure is (aside perhaps from performance optimization). Haskell is "purely functional". Some languages, like SML and Scheme, are typically written in a functional (non-mutating) style but natively support mutation. Some languages, sometimes called "imperative", are very difficult to use functionally: C does not support closures except by explicitly passing data around, and requires explicit memory management of that data. Python and Ruby are both equally capable of being used functionally and/or imperatively, but in my experience the Ruby libraries make Ruby a bit more natural to use functionally than the Python libraries are for Python.

23.1.3 recursion, iterate, imperative, immutable, side-effect, global

Recursion is when a function calls itself (or calls another function that calls the first one).

#1, C:

```
int factorial(int i) {
   if(i == 0) return 1;
   else return i * factorial(i - 1);
}
```

Iteration is when a function repeats something using a structure other than recursion.

#2, C:

```
int factorial(int i) {
   int result = 1;
   while(i > 0) {
     result *= i;
     --i;
   }
   return result;
}
```

The latter is typically described as imperative. Assigning to 'result' and 'i' repeatedly are examples of imperative actions. Programs that avoid assignment are written in a "functional" style; languages that forbid assignment are "purely functional". An assignment is a "side-effect"; so is printing a string to the screen or reading microphone input. (Yes, side-effect is sometimes used to refer to reading some part of the outside world without changing it. Sometimes.)

Some languages allow variables to be "immutable". Immutable values are nice because you know they can't change, but annoying because you can't change them. In Haskell, all values are immutable; in JavaScript, none are; in C++, individual values can be declared 'const' (immutable) and the compiler will check that they are not changed; in SML, values are immutable unless explicitly declared mutable.

Global variables are in the top-level scope that all functions can refer to. Functions and global constants are alright. Global variables can be modified, and are usually a bad idea.

There exists purely functional iteration: **#3, Haskell:**

```
factorial i = foldr (*) 1 [1..i]
```

(foldr is implemented using recursion, as is common for iteration functions in functional languages.)

Tail-recursion is when a function's recursive calls are in "tail position" - they immediately return the called function's result. The above C recursive factorial #1 is not tail-recursive. This is one way to write it tail-recursively:

```
int factorial_impl(int i, int accum) {
   if(i == 0) return accum;
   else return factorial_impl(i - 1, accum * i);
}
int factorial(int i) {
   return factorial_impl(i, 1);
}
```

23.1.4 stack overflow, thread, concurrent, fork

Tail recursion is interesting because function calls form a stack, with the bottom of the stack being the function the program starts with (e.g. main()), the next being the function currently being called by main(), the next being the function currently being called by that, etc. (At the top of the stack, the function is typically executing a machine or VM instruction, or syscall, or foreign call into a different computer language.) Some (most) languages restrict their stack size in memory to be significantly smaller than their heap (heap: the place where most dynamic memory allocations come from). Reaching the limit of stack space is called "stack overflow" and causes a program to crash or throw an exception (depending on the language).

Tail recursion can be implemented by, on the stack, *replacing* the calling function by the called function, rather than by pushing a new frame onto the stack. This is called the "tail-call optimization". This makes an asymptotic difference in memory usage, even if it weren't for artifically constrained stack sizes; #1 above is O(N) in memory usage (assuming the optimizer isn't clever enough to realize it doesn't need to be), and #3 is... also possibly O(N), because the C spec does not guarantee tail-calls to replace the current function's stack frame. But in a good language it would be guaranteed to be O(1), and a good optimizing C compiler (e.g. GCC) will in fact tail-call-optimize #3. C's ABI makes it difficult or impossible (I forget which) for a compiler to tail-call-optimize all cases, though.

Because the "tail-call optimization"'s main effect is to determine whether your code stack-overflows on large inputs or not, it isn't really an optimization; you can only rely on it if your language or implementation guarantees it. The Scheme standard guarantees it. Haskell has tail-recursion although it is of different significance in a lazily evaluated language. C, C++, Java, Javascript, and many other modern languages do not guarantee tail-call optimization (although most of the modern scripting languages have no fundamental barrier to it, so some of their implementations may choose to guarantee it).

Each execution stack is a "thread". There is the initial thread that runs "main()", and that initial thread, in languages that support this (e.g. C and Haskell, but not CPython [the main implementation of Python]), is free to create new threads in the same program. These new threads do things at the same time (and/or alternating back and forth unpredictably, if there aren't enough CPUs to run all running threads of all processes at the same time). This is useful to exploit multiple CPUs ("parallelism"), or to semantically do things at the same time such as Apache responding to multiple Web requests while avoiding a single one's slowness locking up the whole web server. This is hazardous because it's unpredictable when the threads run relative to each other.

"Thread safety" is hard to get right, especially if the threads operate on the same data in memory. Failed attempts at thread safety lead to "race conditions", where threads alternate operations in the wrong order and do nonsensical buggy things, or "deadlocks", where each thread is waiting on another thread simultaneously so the program freezes.

There are various alternative models to threads. The process can fork a whole separate process and have them communicate via e.g. pipes. Erlang is built around a concurrency model where there are pretend mini processes communicating inside the same OS process. Some languages have "green threads" which have semantics similar to OS threads, but do not use an OS thread for each thread; some run in just one OS thread (concurrency but no parallelism) and advanced ones (e.g. GHC Haskell's) can use N OS threads where N is the number of CPUs or a user-set number.

"Coroutines" or "evented programming" involve only changing between different tasks at specific points the programmer specifies; this makes thread safety pretty easy, but risks one task blocking another task for a long time.

23.1.5 interpreted, compiled, link

Languages are implemented in a few different ways.

They can be "compiled". A program called a "compiler" transforms source code into machine code. Often, each source file is separately compiled and then a "linker" is used to compile those pieces ("object files") into an executable.

They can be "interpreted". A program called an "interpreter" takes the source code, processes it slightly in memory (usually to something called a "byte-code"), and then runs it via code that looks at each operation the program does, does it, and then moves on to the next operation.

They can be "JITted". This is like interpreting but (usually) faster. The interpreting program (sometimes or always) transforms pieces of the code in-memory into machine code, but only when the pieces are called. This means potentially less initial time taken than compiling the whole thing, while keeping some of the performance of machine code.

C and C++ are usually compiled. Java and JavaScript are usually JITted. Haskell can be either compiled, interpreted, or a mix. Lua and Python can be interpreted (by their respective main implementations) or JITted (by LuaJIT and e.g. PyPy, respectively). There is usually no fundamental thing about a language that requires it to be in one category; for example, GCC people implemented a compiler for Java.

There are also a few bytecode formats that are standardized and written to disk, e.g. JVM and CLR. JVM started out as Java's bytecode, but became popular enough that many people implemented ways to compile other languages to JVM code.

23.1.6 dynamic, static, type

Some languages have a static type system, where

```
[1,2,3] / "foobar"
```

is a serious enough error (division on non-numbers!) that the language will report an error rather than compile or interpret the program at all. On the other hand, they also reject programs, like: pseudocode:

```
v = (an int or a string from somewhere);
b = (whether it is an int);
if(b) {
  (use v as an int);
}
else {
  (use v as a string);
}
```

, that would run without error in a dynamically typed language.

People online have flamewars about whether dynamic or static typing is better. They are wrong.

In dynamic languages, a value is typically represented at runtime as a tag indicating what type it is, plus the actual data. C (a statically typed language) doesn't need to do this, which helps it be faster. On the other hand, dynamic languages with optimized implementations (e.g. JavaScript) can avoid this some of the time, and static languages sometimes do this anyway (e.g. Java, because it has introspection and because it allows methods to be overriden by subclasses, and every Haskell type that is a union, which is common in Haskell).

Some static languages (e.g. C, Java) require the programmer to specify the type of everything explicitly. Dynamic language advocates hate writing those types. Other languages (e.g. ML, Haskell) have type inference that mean the programmer needs to write few or no type annotations. Dynamic language advocates hate deciphering the confusing error messages that Hindley-Milner type inference commonly leads to (or, more often, have never tried a language with type inference...). In Haskell it is still allowed to write types, which are checked by the compiler and also serve as documentation (and can also be used to make a function less polymorphic than it would naturally be, for API reasons or to increase performance).

Static language advocates hate writing a program in a dynamic language and having to wait until the program actually runs some particular piece of code before finding out it has a type error. Static language advocates may like the fact that their languages provide a way to name all types and thus think about them. Dynamic languages where everything is an object, like Ruby, allow that too, although they don't necessarily have as formal a way to describe compound types like "a list of integers".

Dynamic language advocates may like the way functions can be defined such that they can take all sorts of different argument combinations to mean different variations on the same function. This is common in JavaScript libraries. C++ (a static language) allows function overloading which achieves some of this effect.

Both sides argue about whether their preference is better for rapid prototyping. I personally find Haskell and JavaScript both better for rapid prototyping than C++. Which is better varies based on the problem – e.g. Haskell has pattern-matching, and JavaScript easily has HTML interfaces, neither of which are fundamentally related to the language's type system. (Erlang is dynamically typed with pattern matching; JavaScript just happens to be the language of the Web.)

23.1.7 lazy, pass-by-reference, pass-by-value

In Haskell, you can define "if" as a function:

```
my_if cond t f = case cond of
                   True -> t
                   False -> f
```

and the "false" branch will not be evaluated unless the condition is false, and the "true" branch will not be evaluated unless the condition is true. This is called lazy evaluation. As it happens, the compiler can detect that this function always uses 'cond' and thus it can note (as an optimization) that my_if is strict in 'cond'. Most languages evaluate all arguments fully before calling a function.

In C,

```
void foo(int i) {
  [...]
}
```

passes the int by value. If the function changes it, the caller will be unaffected. If 'int' was instead a complicated data structure, it will take a relatively long time to copy it.

```
void foo(int * i) {
  [...]
}
```

passes the argument by reference, or the C++

```
void foo(int & i) {
  [...]
}
```

which avoids syntactic overhead of passing by reference (for better or for worse).

In Haskell, every value is passed by reference (unless the optimizer concludes it can pass something by value).

In Python, JavaScript, Scheme, etc, some types are passed by value (ints, cons cells though not the things they refer to, etc) and some are passed by reference (lists, dictionaries, etc). This is the right decision for efficiency's sake, and maybe (I have no opinion) for semantics. Until you realize that certain types are passed by-value and learn which ones they are, though, it is super confusing if you're trying to modify a passed-in value. (Actually, the ones I said are passed by value might be implemented differently, e.g. by a reference to an immutable object, which could have the same semantic effect. I am not in the know.)

23.1.8 pattern, syntactic sugar

Scheme has almost no syntax; it is parenthesized sequences, the first item of which acts upon the rest.

```
(define (f x y) (+ x y))
```

Some languages have a lot of syntax. Some of it is "syntactic sugar", which means its effect is equivalent to some other part of the language but it looks different or better for what it does. For example, in Haskell,

```
if x then y else z
```

is syntactic sugar for

```
case x of
  True -> y
  False -> z
```

.

Haskell (as shown above) has pattern-matching, which you may or may not see as syntactic sugar. Haskell has

```
data Bool = True | False
```

(booleans don't even need to be defined as a built-in type!)

That case statement is implemented ultimately by a test and branch, rather like an imperative language's "if" statement applied to a tag that says which alternative the thing is, but there is no way to say that in Haskell. A more interesting type is

```
data Maybe a = Nothing | Just a
```

and then pattern-matching can reveal the variables within:

```
f :: Maybe Int -> Int
f (Just x) = x
f Nothing = 42
```

(That isn't a very good example, but it will serve.)

23.1.9 class, inheritance, method, object

Some languages are "object-oriented", which is a buzzword usually meaning "good" by the people who say it. What it actually means is that there are data types called "objects" that (for better or for worse) combine a lot of programming features into one place. C is not object-oriented (although there are C macro libraries that let you use C in a more object-oriented way, used by e.g. GTK+; their horrors might not be worth the benefit). Java and Ruby are object-oriented. Haskell is only object-oriented if you believe "object-oriented" means "good".

An object is usually of a class; the object is an “instance” of the class, and the class is written with a name and description in the source code. The class specifies a collection of data fields that the object has, and methods, which are functions that take the object as a parameter (usually with special syntax).

Classes can inherit from other classes. For example, in Lasercake (a game with robots that can move around), “autonomous_object” (representing physical things that do something every frame) is a subclass of “object” (representing physical things in the world), and “laser_emitter” is a subclass of “autonomous_object”. “object” and “autonomous_object” are, in this case, abstract classes because you can’t make an object from that class. laser_emitter is concrete; you can make a laser emitter, which, by its nature, is also an autonomous_object and an object. It has any data fields and methods defined by its superclasses, and can be used any place a superclass is expected. This lets us have a collection of autonomous_objects and call all their update() methods every frame. Objects are not necessary for this type of polymorphism; Python and Go have duck-typing (where a common superclass is not necessary) and Haskell has type-classes.

(A few languages, like JavaScript and Lua, have prototypal inheritance, where an object inherits from another object, rather than classes, where an object is of a class and the class inherits.)

23.1.10 API, interface, namespace, comment, package

Classes also, in many languages, have data-hiding; the data fields can be forbidden from access outside of the class’s methods. This is useful to make clean boundaries between different parts of code, to allow the implementation to be changed later, to present a clean API, etc. Haskell accomplishes this via modules. C accomplishes this via using many files and separate interface (header, .h) files and source (.c) files (and using the ‘static’ keyword for definitions local to a single .c file, etc).

An interface is a formal specification (or informal concept) of how a piece of code can be used by other pieces of code. It often consists of things like function signatures or the public parts of class declarations. For example, the methods of JavaScript's arrays form an interface, even if you think it isn't a very good interface. You can use it. You can't change how it works without re-writing the array code. An API ("application programming interface") is a fancy name for an interface or collection of interfaces. Interfaces can be hard to change because a lot of code may rely on the old version of the interface (thus the reason we have a not-so-great JavaScript array interface... wait, I don't actually think it's terrible, but anyway). Luckily, interfaces mean that the implementation code can be changed at will, to make it faster or less buggy or do more things, and we'll only have to think about the interface's guarantees to think about what we'll break. The worst case "complexity" of code interacting with other code is n^2 (every piece with every other piece), and interfaces make this less bad.

Files/modules are often combined into "library packages" (e.g. a priority queue or an HTTP implementation could be a library). These require some metadata that makes it easier for other people to install and use the library.

Comments are things in code that the compiler ignores but humans might look at. They are useful to document interfaces, and to document algorithms that are too complicated for the code to explain itself. They are also used in lots of other ways that may be good or bad depending on what you think. Since they are not checked by the compiler, it's possible that someone changes the code without changing a comment about the code, and then you have a wrong comment, which is often worse than no comment.

23.1.11 hash, list, data-structure, collection, vector

Data structures are classes in languages that are object-based. Otherwise they are just data structures. They combine groups of objects in various ways.

A vector is a sequence of objects that is laid out sequentially in memory. If you have the numeric index of an object, you can access it in constant time (according to the sketchy idea that memory access is constant time; considering caches and swapping, it's really a hierarchy with different properties, but everyone ignores that when talking about asymptotic performance). Vectors are memory-efficient but it takes a while to insert an element at the beginning or middle.

Linked lists are also sequences; each element, in addition to its contained value, contains a reference to the next element (and possibly the prior element). These allow insertion in constant time anywhere (once you have a reference to that location), but linear time lookup by index.

Hashes are associations from "keys" to "values". They allow, given a key, to look up the corresponding value in average-case constant time.

23.1.12 macro

Macros are a textual substitution mechanism in C and C++, which makes them fragile and to be avoided where possible.

Macros are powerful program transformers and syntax creators in Lisp.

Macros don't exist in most languages.

23.1.13 throw, exception

Exceptions are things that code can "throw" in cases of errors, like trying to read a file that doesn't exist, or (controversially) a bug in the program like accessing the first element of an empty list. They do something similar to instantly returning from the current function, the function that called it, etc, until they reach a "try/catch" block that surrounds the current execution context, and jump to the "catch" block of it. The catch block contains code designed to handle the exception at a place in the code that knows how to handle it.

If there is no surrounding try/catch block, the program exits.

The existence of exceptions requires code to be exception-safe; for example:

```
void f() {
    FILE* file = fopen(...);
    // do some stuff
    fclose(file);
}
```

If “do some stuff” (including functions it calls) can throw an exception, then the file will never be closed, which is a bug. C++ has an idiom called “RAII” to deal with this in an easier way than writing try/catch/finally blocks everywhere. Python has “with”. Haskell has most of its code being non-side-effecting and thus only has to deal with this occasionally.

23.1.14 test

Untested programs have bugs. If they are statically typed, they have fewer bugs, but they still have bugs. Tests are good. Test cases are easier to write for algorithm-level code than user-interface-level code. Each language has its own different conventions about how to write tests. Tests do not catch all bugs, but if they cover most or all of the program code then they catch a lot of them. Tests made it possible for Sam, Elias and I to write our Python graph-theory code, because we made so many mistakes that it would be impossible to pinpoint bugs by looking at what the whole program’s effect is. It was practical writing a test case where we knew the correct answer that some function should return, then seeing it return a different thing. This way we could fix the lowest-level functions first and work our way up, each layer relying on fairly debugged lower layers. Test cases are also useful to keep around after that debugging, because people will change the code later on, and these tests can tell them whether they broke anything obvious by their changes.

23.2 coding in dissimilar languages

```
question 2 (40%)
----------------

Write six programs implementing solutions to the following two
problems across the three languages you're comfortable with, namely
(Haskell, C/C++, Javascript).

In each case, include docs and tests appropriate to the style of
that language, including how to run them and in what environment.

Use these programs to illustrate the different currently popular
programming paradigms, as well as your mastery of the vernacular
within these programming language communities.

As a postscript, discuss which languages you found well suited
to which problem, and why.

The two problems are

A) the perfect squares crossword puzzle

   Replace the * below with twentyfive base 10 digits to form a
   crossword-like array of thirteen 3-digit perfect squares, with each
   3-digit number reading across or down.  (121 = 11^2 for example is
   a 3-digit perfect square.)

      *   * * *   * * *   *
      * * *   * * *   * * *
      *   * * *   * * *   *

B) family tree

   Write a program to generate a visual family tree from a .csv (comma
   separated value) file of people.

   Each line in the file should represent a person, and include at
   least (name, father, mother, date born, date died). The data format
   is up to you, but should be (a) well defined, and (b) allow for
   multiple people with the same name.  Generate some (fake) data to
```

run your code on, which includes at least 10 people across at least 3 generations.

The family tree should be either ascii art or easily displayed image (e.g. .png, .pdf, .svg, .html, ...) as you choose. You may use an external graphics library appropriate to the language; if so as usual quote your sources explicitly.

23.2.A the perfect squares crossword puzzle

```
A) the perfect squares crossword puzzle

   Replace the * below with twentyfive base 10 digits to form a
   crossword-like array of thirteen 3-digit perfect squares, with each
   3-digit number reading across or down.  (121 = 11^2 for example is
   a 3-digit perfect square.)

      *   * * *   * * *   *
      * * *   * * *   * * *
      *   * * *   * * *   *
```

23.2.A.Haskell

```haskell
--This file is a stream of consciousness:
--I wrote it from top to bottom (aside from the imports list),
--stopping many times along the way to typecheck, test functions
--in the interpreter GHCI, and think about what a nice next step is.


import Data.Char
import Data.List
import qualified Data.Map as Map
import Data.Map(Map)

import Test.QuickCheck

-- GHCI + knowing their bases
perfect3DigitSquares :: [Int]
perfect3DigitSquares = map (\n->n^2) [10..31]

-- since we don't care about efficiency here, just use strings
asDigits :: Int -> [Int]
asDigits = map digitToInt . show

-- tested in GHCI
perfect3DigitSquaresAsDigits :: [[Int]]
perfect3DigitSquaresAsDigits = map asDigits perfect3DigitSquares
```

```
{-


      *   * * *   * * *   *
      * * *   * * *   * * *
      *   * * *   * * *   *

grid: coordinates from top-left starting at (0,0),
number positions indicated by the top-left of them
(and implicitly whether it is across or down, and
we know it’s 3 long).

      0 1 2 3 4 5 6 7 8 9 10
    0 *   * * *   * * *   *
    1 * * *   * * *   * * *
    2 *   * * *   * * *   *

-}

-- location : x,y
type Loc = (Int,Int)
rows :: [Loc]
rows = [(2,0),(6,0),(0,1),(4,1),(8,1),(2,2),(6,2)]
cols :: [Loc]
cols = [(0,0),(2,0),(4,0),(6,0),(8,0),(10,0)]

type Board = Map Loc Int

-- |
-- (Yes, Haskell’s dictionary literal syntax sucks in its nonexistence:)
--
-- >>> areConsistent (Map.fromList [(’a’,5),(’b’,7),(’c’,8)]) (Map.fromList [(’a’,5),(
-- False
-- >>> areConsistent (Map.fromList [(’a’,5),(’b’,7),(’c’,8)]) (Map.fromList [(’a’,5),(
-- True
--
-- (Dear readers of the printed version: I couldn’t find a way
--  that worked to split each test over more than one line.
--  The first one’s argument lists are
--    [(’a’,5),(’b’,7),(’c’,8)] and [(’a’,5),(’b’,2),(’d’,9)]
--  and the second one’s argument lists are
--    [(’a’,5),(’b’,7),(’c’,8)] and [(’a’,5),(’b’,7),(’d’,9)].
-- )
--
-- (to run these tests:
-- % cabal install doctest
-- % ~/.cabal/bin/doctest 2A.hs
-- )
--
-- areConsistent :: Board -> Board -> Bool
areConsistent :: (Ord k, Eq v) => Map k v -> Map k v -> Bool
areConsistent b1 b2 = and (Map.elems (Map.intersectionWith (==) b1 b2))


-- There are 22 possible 3-digit squares,
-- and 13 slots to put squares in.
-- If we just tried every combination, there are 22^13 of them,
-- which is too many.
-- Each placement, however, rules out a lot of possibilities,
-- so a depth first search taking the slots in an order in which
-- they’re connected might be fast enough.

-- three locs in a row:
type Slot = [Loc]
rowToSlot :: Loc -> Slot
rowToSlot (x,y) = [(x,y), (x+1,y), (x+2,y)]
colToSlot :: Loc -> Slot
colToSlot (x,y) = [(x,y), (x,y+1), (x,y+2)]
slots :: [Slot]
slots = map rowToSlot rows ++ map colToSlot cols

-- Sorting by (x+y) should be good enough for an efficient depth-first
-- search ordering for this problem.
roughlySortedSlots :: [Slot]
roughlySortedSlots = sortBy
  (\[(x1,y1),_,_] [(x2,y2),_,_] -> compare (x1+y1) (x2+y2))
  slots

-- we want:
-- solution :: Board

-- We might as well solve for all the boards, because Haskell’s laziness
-- means that if we only ask for the first one then we won’t need to compute
-- all possible solutions.

solve :: Board -> [Slot] -> [Board]
solve boardSoFar [] = [boardSoFar]
solve boardSoFar (slot:slotsStillToFill) = let
  nextBoards = [Map.union boardSoFar miniboard
              | square <- perfect3DigitSquaresAsDigits,
                let miniboard = Map.fromList (zip slot square),
                areConsistent boardSoFar miniboard]
  in concat (map (\board -> solve board slotsStillToFill) nextBoards)

solution :: Board
solution = head (solve Map.empty roughlySortedSlots)
-- indeed using ’slots’ makes this too slow; the sorting helped!
-- But now how will I check if it’s correct?  I need a way to print it.

showBoard :: Board -> String
showBoard board = let
  minx = minimum (map fst (Map.keys board))
  miny = minimum (map snd (Map.keys board))
  maxx = maximum (map fst (Map.keys board))
  maxy = maximum (map snd (Map.keys board))
  xs = [minx..maxx]
  ys = [miny..maxy]
 in unlines (map (\y -> map (\x ->
    case Map.lookup (x,y) board of
      Nothing -> ’ ’
      Just n -> intToDigit n
  ) xs) ys)

{-
Now, in ghci,
> putStrLn (showBoard solution)
1 225 121 1
225 729 900
1 676 676 0

YAY IT WORKED!
On the first try after it typechecked, at that.
I could never have done it so easily in a less type-focused language.
Most of the time coding was spent thinking about what I actually wanted.
I would likely have done it in Haskell first even if I was ultimately aiming
to run it in another language, because Haskell was a language that
let me think about this problem.
-}

--might as well have a 'main' that shows the solution.
main :: IO ()
main = putStrLn (showBoard solution)

-- Now how will I write this in C/C++ and how will I write it in JavaScript?
-- JavaScript can do most of the things I do here; some of the Map operations
-- will be trickier (JS doesn't provide e.g. intersectionWith!) but I can
-- emulate this in a functional style in JS.  I'm not sure whether it will be
-- easy to document without types, though.
--
-- C/C++ is more imperative.  They lack garbage collection and lack
-- functional-style libraries.  I might have to implement 'solve' using
-- loops and recursion (using the stack as a part of the depth first
-- searching) - I guess that is not terribly different from the Haskell.

-- Haskell tests:
-- Sometimes Haskell is tested with QuickCheck, which tests
-- mathematical properties by trying them with several randomly generated
-- examples.
--
-- To run these:
-- % cabal install quickcheck
-- % cabal install quickcheck-script
-- % ~/.cabal/bin/quickCheck 2A.hs
prop_ConsistentWithSelf :: [(Int,Int)] -> Bool
prop_ConsistentWithSelf m =
  areConsistent m' m' where m' = Map.fromList m

prop_ConsistentWithEmpty :: [(Int,Int)] -> Bool
prop_ConsistentWithEmpty m =
  areConsistent m' Map.empty where m' = Map.fromList m

prop_inconsistentWithDifferentVal :: (Int,Int) -> Bool
prop_inconsistentWithDifferentVal (k,v) =
  not (areConsistent (Map.singleton k v) (Map.singleton k (v+1)))

prop_consistentWithDifferentPlace :: (Int,Int) -> Bool
prop_consistentWithDifferentPlace (k,v) =
  areConsistent (Map.singleton k v) (Map.singleton (k+1) v)

-- Extensibility:
-- Several things like, say, 'rows' would change with a different board.
-- All of those bindings would probably go inside a single function that
-- took that information, then.  'solve' relies on the three digit squares
-- but nothing else -- it could sensibly be tweaked to observe the size of
-- a slot and use corresponding-length squares.
```

23.2.A.JavaScript

```
// Copied from 2A.hs and rewritten as I go.
//
// Run in e.g. node.  In this directory (npm installs
// in ./node_modules by default), do:
//
// % npm install underscore
// % node 2A.js
//
// Tests are currently just asserts run when this file is loaded.

var _ = require(’underscore’);
var assert = require(’assert’);

var perfect3DigitSquares = _.map(_.range(10,32), function(n) { return n*n; });

function asDigits(integer) {
  // Since JS’s strings are nontrivial to use as arrays I’ll just write
  // the better, non-string-using, code here.
  if(integer === 0) {
    return [0];
  }
  var result = [];
  while(integer !== 0) {
    var digit = integer % 10;
    result.push(digit);
    integer = (integer - digit) / 10;
  }
  return result.reverse();
}

assert.deepEqual(asDigits(0), [0]);
assert.deepEqual(asDigits(13), [1,3]);
assert.deepEqual(asDigits(127), [1,2,7]);


var perfect3DigitSquaresAsDigits = _.map(perfect3DigitSquares, asDigits);

/*
```

```
      *   * * *   * * *   *
      * * *   * * *   * * *
      *   * * *   * * *   *

grid: coordinates from top-left starting at (0,0),
number positions indicated by the top-left of them
(and implicitly whether it is across or down, and
we know it's 3 long).

      0 1 2 3 4 5 6 7 8 9 10
    0 *   * * *   * * *   *
    1 * * *   * * *   * * *
    2 *   * * *   * * *   *

*/

// location : x,y
// [x,y] or {x:x, y:y} would be reasonable
// function loc(x, y) { return [x, y]; }
function mkloc(x, y) { return { x: x, y: y }; }
var rows =
  [mkloc(2,0),mkloc(6,0),mkloc(0,1),mkloc(4,1),mkloc(8,1),mkloc(2,2),mkloc(6,2)];
var cols =
  [mkloc(0,0),mkloc(2,0),mkloc(4,0),mkloc(6,0),mkloc(8,0),mkloc(10,0)];

// The board is a hash from loc to digit... but wait, JS objects can only
// have strings, not pairs, as keys!  So we'll have a way to transform
// locs to hash key strings and back.
//
// (I might not have realized this up-front if I hadn't been looking at
// the data declaration in Haskell.  Instead I would've realized it while
// writing the next function or so, and had to go back and write this then.)
function lockey(loc) { return loc.x + "," + loc.y; }
//(debugging: this needs to turn the x/y strings into numbers,
//and using 'map' makes this long enough that the functional style
//is not worth it anymore.)
//function keyLoc(key) { return mkloc.apply(null, key.split(',')); }
function keyLoc(key) {
  var xy = key.split(',');
  return mkloc(+xy[0], +xy[1]);
}
```

```
// JS doesn’t come with anything like intersectionWith.
function areConsistent(board1, board2) {
  for(var l in board1) if(_.has(board1, l)) {
    if(_.has(board2, l) && board2[l] !== board1[l]) {
      return false;
    }
  }
  return true;
}

// But at least JS has dictionary literals.
assert(!areConsistent({’a’: 5, ’b’: 7, ’c’: 8}, {’a’: 5, ’b’: 2, ’d’: 9}),
         "inconsistent boards");
assert(areConsistent({’a’: 5, ’b’: 7, ’c’: 8}, {’a’: 5, ’b’: 7, ’d’: 9}),
         "consistent boards");

// There are 22 possible 3-digit squares,
// and 13 slots to put squares in.
// If we just tried every combination, there are 22^13 of them,
// which is too many.
// Each placement, however, rules out a lot of possibilities,
// so a depth first search taking the slots in an order in which
// they’re connected might be fast enough.

// A slot is three locs in a row, in an array.
function rowToSlot(loc) {
  return [mkloc(loc.x, loc.y), mkloc(loc.x+1, loc.y), mkloc(loc.x+2, loc.y)];
}
function colToSlot(loc) {
  return [mkloc(loc.x, loc.y), mkloc(loc.x, loc.y+1), mkloc(loc.x, loc.y+2)];
}
var slots = _.map(rows, rowToSlot).concat(_.map(cols, colToSlot));

// Sorting by (x+y) should be good enough for an efficient depth-first
// search ordering for this problem.
var roughlySortedSlots = slots.sort(function(slot1, slot2) {
    return (slot1[0].x + slot1[0].y) - (slot2[0].x + slot2[0].y);
  });

// we want:
// var solution = a board
// There's no easy way for JS to represent a lazy list, so I'll just
// write this to return the first answer.
function solve(boardSoFar, slotsStillToFill) {
  //(debugging... this shows it's resulting in the
  //right thing, so the bug must be in showBoard:
  //console.log("step", boardSoFar, slotsStillToFill);)
  if(slotsStillToFill.length === 0) {
    return boardSoFar;
  }
  var nextBoards = [];
  _.each(perfect3DigitSquaresAsDigits, function(square) {
    var miniboard = _.object(_.map(slotsStillToFill[0], locKey), square);
    if(areConsistent(boardSoFar, miniboard)) {
      nextBoards.push(_.extend({}, boardSoFar, miniboard));
    }
  });
  for(var i = 0; i !== nextBoards.length; ++i) {
    // Does this copying-based impl of slotsStillToFill recursion
    // make the runtime asymptotically worse?  I'm not sure.
    // I could pass around an index into the array instead of slicing
    // it, given that JS doesn't come with linked lists.
    var solvedBoard = solve(nextBoards[i], slotsStillToFill.slice(1));
    if(solvedBoard !== null) {
      return solvedBoard;
    }
  }
  return null;
}

var solution = solve({}, roughlySortedSlots);

function getx(loc) {
  return loc.x;
}
function gety(loc) {
  return loc.y;
}

function showBoard(board) {
  var locs = _.map(_.keys(board), keyLoc);
  var minx = _.min(_.map(locs, getx));
  var miny = _.min(_.map(locs, gety));
  var maxx = _.max(_.map(locs, getx));
  var maxy = _.max(_.map(locs, gety));
  // debugging:
  // I introduced this to debug another thing and now
  // it's informing me that maxx is 9 when it should be
  // 10 (though the rest are correct and solution is
  // correct).  Perhaps the two digits are causing problems
  // somehow?  Ah - whoops we're doing min/max on strings!
  // (bug in keyLoc due to lack of type safety.)
  // Yay that fixed the program and it outputs the right
  // answer now.
  console.log(minx, miny, "&&", maxx, maxy);
  var xs = _.range(minx, maxx+1);
  var ys = _.range(miny, maxy+1);
  var lines = _.map(ys, function(y) {
    return "".concat.apply("", _.map(xs, function(x) {
      var tile = board[locKey(mkloc(x,y))];
      if(tile === undefined) { return ' '; }
      else { return ""+tile; }
    })) + '\n';
  });
  return "".concat.apply("", lines);
}

//might as well have a 'main' that shows the solution.
function main() {
  console.log(showBoard(solution));
}

main();

// Blurgh, even after fixing JsHint warnings and runtime
// exceptions, it's not outputing the right value.

// Fixes are described in comments that mention the word 'debugging'.
// It works now.
```

23.2.A.C++

```
// Copied from 2A.js and rewritten as I go.
//
// The Javascript was somewhat more imperative-style then
// the Haskell, which means it will be somewhat easier
// to translate into C++.
//
// % g++ -Wall -g -std=c++0x -O3 2A.cpp
// % ./a.out
//
// Tests are currently just asserts run at the beginning of main().
//
// Commented out std::cerrs were put in for debugging purposes and left
// there for anthropological observers' sake.  It turns out I was
// outputting a char as an int because I didn't cast it the right way
// and its numeric value was being output; silly weakly typed C++.
//
// Valgrind spies nothing wrong.

#include <vector>
#include <map>
#include <algorithm>
#include <iterator>
#include <iostream>
#include <cassert>

using std::vector;
using std::map;

vector<int> findPerfect3DigitSquares() {
  vector<int> result;
  for(int i = 10; i != 32; ++i) {
    result.push_back(i*i);
  }
  return result;
}
const vector<int> perfect3DigitSquares = findPerfect3DigitSquares();

typedef int digit;
vector<digit> asDigits(int integer) {
  vector<digit> result;
  if(integer == 0) {
    result.push_back(0);
  }
  else {
    while(integer != 0) {
      digit d = integer % 10;
      result.push_back(d);
      integer = (integer - d) / 10;
    }
    std::reverse(result.begin(), result.end());
  }
  return result;
}



vector<vector<digit>> findPerfect3DigitSquaresAsDigits() {
  vector<vector<digit>> result;
  for(int i : perfect3DigitSquares) {
    result.push_back(asDigits(i));
  }
  return result;
}
const vector<vector<digit>> perfect3DigitSquaresAsDigits =
                              findPerfect3DigitSquaresAsDigits();

/*


      *   * * *   * * *   *
      * * *   * * *   * * *
      *   * * *   * * *   *

grid: coordinates from top-left starting at (0,0),
number positions indicated by the top-left of them
(and implicitly whether it is across or down, and
we know it's 3 long).

      0 1 2 3 4 5 6 7 8 9 10
    0 *   * * *   * * *   *
    1 * * *   * * *   * * *
    2 *   * * *   * * *   *

*/

typedef int Coord;
struct Loc {
  Loc(Coord x, Coord y) : x(x), y(y) {}
  Coord x;
  Coord y;
};
const vector<Loc> rows =
  {Loc(2,0),Loc(6,0),Loc(0,1),Loc(4,1),Loc(8,1),Loc(2,2),Loc(6,2)};
const vector<Loc> cols =
  {Loc(0,0),Loc(2,0),Loc(4,0),Loc(6,0),Loc(8,0),Loc(10,0)};

// Use a tree map - less-than is slightly easier to implement than hash
// and it doesn't matter (and the hash-based unordered_map has only been
// standardized for a few years).
// Luckily, in C++, like Haskell and unlike Javascript, anything can be
// a key of the standard dictionary type!
// (You just have to implement the comparison functions.)
inline bool operator==(Loc a, Loc b) {
  return a.x == b.x && a.y == b.y;
}
inline bool operator<(Loc a, Loc b) {
  return a.x < b.x || (a.x == b.x && a.y < b.y);
}
typedef map<Loc, digit> Board;

// C++ has similar lack of functional ways of doing things to JavaScript,
// and we already imperative-ized this for JS, so it's not hard.
// On the other hand, STL iterators make the code a little messier looking.
// Luckily C++11 has 'auto', a limited form of type inference, and
// range-based for loops, which make this code more readable.
//
// Since this can work on any associative container, why write this:
//bool areConsistent(Board const& board1, Board const& board2) {
// when we could write it polymorphically:
template<typename Assoc>
bool areConsistent(Assoc const& board1, Assoc const& board2) {
  for(auto pair : board1) {
    auto i = board2.find(pair.first);
    if(i != board2.end() && i->second != pair.second) {
      return false;
    }
  }
  return true;
}


// There are 22 possible 3-digit squares,
// and 13 slots to put squares in.
// If we just tried every combination, there are 22^13 of them,
// which is too many.
// Each placement, however, rules out a lot of possibilities,
// so a depth first search taking the slots in an order in which
// they're connected might be fast enough.

// A slot is three locs in a row, in an array.
typedef vector<Loc> Slot;
Slot rowToSlot(Loc loc) {
  return vector<Loc>{Loc(loc.x, loc.y), Loc(loc.x+1, loc.y), Loc(loc.x+2, loc.y)};
}
Slot colToSlot(Loc loc) {
  return vector<Loc>{Loc(loc.x, loc.y), Loc(loc.x, loc.y+1), Loc(loc.x, loc.y+2)};
}
vector<Slot> findSlots(vector<Loc> const& rows, vector<Loc> const& cols) {
  vector<Slot> result;
  for(Loc loc : rows) { result.push_back(rowToSlot(loc)); }
  for(Loc loc : cols) { result.push_back(colToSlot(loc)); }
  return result;
}
const vector<Slot> slots = findSlots(rows, cols);

// Sorting by (x+y) should be good enough for an efficient depth-first
// search ordering for this problem.

vector<Slot> sortedForSearch(vector<Slot> slots) {
  std::sort(slots.begin(), slots.end(), [](Slot const& slot1, Slot const& slot2) {
    return (slot1[0].x + slot1[0].y) < (slot2[0].x + slot2[0].y);
  });
  return slots;
}
const vector<Slot> roughlySortedSlots = sortedForSearch(slots);
```

```
// we want:
// const Board solution = ...;

void showBoard(Board const& board, std::ostream& os);
// There's no easy way for C++ to represent a lazy list, so I'll just
// write this to return the first answer.
Board solve(Board const& boardSoFar, vector<Slot> slotsStillToFill) {
  //showBoard(boardSoFar, std::cerr);
  //std::cerr << "\n===" << slotsStillToFill.size() << "\n\n";
  if(slotsStillToFill.size() == 0) {
    return boardSoFar;
  }
  vector<Board> nextBoards;
  const Slot nextSlot = slotsStillToFill[0];
  for(vector<digit> square : perfect3DigitSquaresAsDigits) {
    Board miniboard;
    assert(nextSlot.size() == square.size());
    for(size_t i = 0; i != nextSlot.size(); ++i) {
      //std::cerr << "??" << square[i] << "\n";
      miniboard[nextSlot[i]] = square[i];
    }
    if(areConsistent(boardSoFar, miniboard)) {
      nextBoards.push_back(miniboard);
      std::copy(boardSoFar.begin(), boardSoFar.end(),
                std::inserter(nextBoards.back(), nextBoards.back().end()));
    }
  }
  // Does this copying-based impl subSlots make the runtime
  // asymptotically worse?  I'm not sure.
  // I could pass around an index into the array instead of slicing it.
  const vector<Slot> subSlots(slotsStillToFill.begin()+1, slotsStillToFill.end());
  for(Board const& board : nextBoards) {
    Board solvedBoard = solve(board, subSlots);
    if(solvedBoard.size() != 0) {
      return solvedBoard;
    }
  }
  return Board();
}

const Board solution = solve(Board(), roughlySortedSlots);
```

```
void showBoard(Board const& board, std::ostream& os) {
  Coord minx = std::numeric_limits<Coord>::max();
  Coord miny = std::numeric_limits<Coord>::max();
  Coord maxx = std::numeric_limits<Coord>::min();
  Coord maxy = std::numeric_limits<Coord>::min();
  for(auto const& p : board) {
    const Loc loc = p.first;
    //std::cerr << "??" << loc.x << "?" << loc.y << "??" << p.second << "\n";
    if(minx > loc.x) { minx = loc.x; }
    if(miny > loc.y) { miny = loc.y; }
    if(maxx < loc.x) { maxx = loc.x; }
    if(maxy < loc.y) { maxy = loc.y; }
  }
  for(Coord y = miny; y <= maxy; ++y) {
    for(Coord x = minx; x <= maxx; ++x) {
      auto i = board.find(Loc(x,y));
      if(i == board.end()) { os << ’ ’; }
      else { os << char(’0’ + i->second); }
    }
    os << ’\n’;
  }
}

int main() {
  assert((asDigits(0) == vector<digit>{0}));
  assert((asDigits(13) == vector<digit>{1,3}));
  assert((asDigits(127) == vector<digit>{1,2,7}));

  {
  // Well, this is a hacky-looking way to give C++ dictionary literals,
  // but in C++11, it works!
  typedef std::pair<char, int> P;
  typedef std::map<char, int> M;
  assert(!areConsistent(M{P(’a’, 5), P(’b’, 7), P(’c’, 8)},
                        M{P(’a’, 5), P(’b’, 2), P(’d’, 9)}));
  assert(areConsistent(M{P(’a’, 5), P(’b’, 7), P(’c’, 8)},
                       M{P(’a’, 5), P(’b’, 7), P(’d’, 9)}));
  }

  showBoard(solution, std::cout);
}
```

23.2.B family tree

```
B) family tree

   Write a program to generate a visual family tree from a .csv (comma
   separated value) file of people.

   Each line in the file should represent a person, and include at
   least (name, father, mother, date born, date died). The data format
   is up to you, but should be (a) well defined, and (b) allow for
   multiple people with the same name.  Generate some (fake) data to
   run your code on, which includes at least 10 people across at least
   3 generations.

   The family tree should be either ascii art or easily displayed
   image (e.g. .png, .pdf, .svg, .html, ...) as you choose. You may
   use an external graphics library appropriate to the language; if
   so as usual quote your sources explicitly.
```

23.2.B.notes / brainstorming

What visualization writer wouldn't use GraphViz? Okay, there are reasons, but in this case using this external library makes a lot of sense! Laying out a tree can be a messy operation and Graphviz has many methods and heuristics. So I suppose I will be showing off each language's GraphViz bindings/interfacing and comparing those, heh. Well, and parsing a CSV and turning it into that (or into .dot textual format).

What might a CSV format look like?

> "Each line in the file should represent a person, and include at least (name, father, mother, date born, date died). The data format is up to you, but should be (a) well defined, and (b) allow for multiple people with the same name."

So we'll need an ID, say, an int, to identify a person unambiguously.

> id,name,father-id,mother-id,birthdate,deathdate

(the dates in some format).

If this is realistic, not everyone will know their father or mother, so perhaps -1 means "not known/specified". Better yet, people can just have a variable number of parents, which fixes the gender problem (technology is such that even biological parents are not always one man + one woman these days) :

> id,name,parent-id;parent-id;,birthdate,deathdate

A weakness: this version of CSV cannot contain commas in fields. Some names (John Q. Public, Jr.) contain commas. I'll just make it semicolon-separated values rather than introduce quoting.

That seems fine enough. Now we have to

- generate some random data
- parse the data
- shovel it into GraphViz format (ids will be GraphViz node ids, and names will be specified as the text inside the node).

If a father-id or mother-id is someone we don't have info on, we should probably make that be a node with text "??" or similar, to make it possible to point out that someone has parents. This format doesn't allow "??"s for children, though (except by naming them "??"). Ah well.

Implementation wise: Some languages have GraphViz bindings. People have even used Emscripten to compile GraphViz to JavaScript. Graphviz also has a nice text format for graphs that's fairly easy to generate. So I'll start out by generating the text format, using JavaScript, because that's probably the best of these languages for thrown-together text processing.

...so, I did that, and made an example 2B.csv, and it shows that I don't know very much about family trees. Family trees usually have marriages in them. I could link people horizontally when they are both the same person's parent, if I found out a way to get GraphViz to do that. It would be a bit ugly for three parents but that seems acceptable for something that's less common. Anyway, I don't know very many families (knowing a person when I don't know anyone else in their family isn't good enough), so I don't feel I have the best sensitivity for the subject, and the direction to polish it in isn't super clear, and personally I would prefer paper and pencil for thinking about people's family trees, so I think I'll leave it at that for now.

23.2.B.JavaScript

```
#!/usr/bin/env node

// usage:
//   (edit 2B.csv)
//   node 2B.js | dot -Tpng | display -
// I wanted:
//   cat 2B.csv | node 2B.js | dot -Tpng | display -
// but node was giving me issues with importing process.stdin.
//
// This file could easily call all of these things
// in the pipeline itself, but it seems more Unixy to do it this way.

var _ = require(’underscore’);
var fs = require(’fs’);
var csvText = fs.readFileSync(’2B.csv’, ’utf8’);

function parseCsvText(text) {
  var lines = _.filter(text.split(’\n’), function(line) {
    return line.length > 0;
  });
  return _.map(lines, function(line) {
    var fields = line.split(’;’);
    return fields;
  });
}

function parseFamilyTreeIntoRows(text) {
  var lines = parseCsvText(text);
  var rows = [];
  _.each(lines, function(line) {
    rows.push({
      id: line[0],
      name: line[1],
      parents: line[2].split(’,’),
      birthdate: line[3],
      deathdate: line[4]
    });
  });
  return rows;
}

function asHashSet(arr) {
  var result = {};
  _.each(arr, function(elem) {
    result[elem] = true;
  });
  return result;
}

function transformRowsToGraphvizDotText(rows) {
  var describedPeopleIDs = asHashSet(_.pluck(rows, 'id'));
  var mentionedPeopleIDs = asHashSet(
                          _.flatten(_.pluck(rows, 'parents')).
                             concat(_.pluck(rows, 'id'))
                           );
  //var rowsByID = _.object(_.map(rows, function(row) { return [row.id, row]; }));
  var rowsByID = _.object(_.pluck(rows, 'id'), rows);

  var resultLines = [];
  function line(l) {
    resultLines.push(l + '\n');
  }
  line('digraph "family tree" {');
  _.each(_.keys(mentionedPeopleIDs).sort(), function(id) {
    var row = rowsByID[id];
    if(row === undefined) {
      line('"'+id+'" [label="??"]');
    }
    else {
      line('"'+id+'" [label="'+row.name+'"]');
      _.each(row.parents, function(parent) {
        line('"'+parent+'" -> "'+id+'"');
      });
    }
  });
  line('}');
  var result = ''.concat.apply('', resultLines);
  return result;
}

console.log(transformRowsToGraphvizDotText(parseFamilyTreeIntoRows(csvText)));
```

23.2.B.CSV (sample family tree data used by all 23.2.B programs)

```
bob;Bob Theresa;;birthdate;deathdate
mary;Mary Theresa Hynes;bob,1;;
charlie;Charles Hynes;3,4;;
evelyn;Evelyn Theresa-Hynes;mary,charlie;;
toby;Tobias Theresa-Hynes;mary,charlie;;
Yves;Yves Theresa-Hynes-Fairbern;mary,charlie;;
2;??;bob,1;;
```

23.2.B.C

```
// I might as well do this in C to see how that is.

// Usage:
//   gcc -Wall -std=c99 -g 2B.c
//   cat 2B.csv | ./a.out | dot -Tpng | display -

// Testing:
//   cat 2B.csv | valgrind ./a.out
// shows that it does not read any uninitialized memory
// or write any unallocated memory.
// It outputs the right thing.
// It could use better tests, but then again, it could use
// all the known bugs in it fixed too.
//
// C (and C++) have great tooling (gdb! valgrind! kcachegrind!
// lots of IDEs!) but they don't have as easy and fluid use of
// libraries as javascript (NodeJS, npm, or copying/linking js
// libs in the browser) or Haskell (cabal) or several other of
// the scripting languages.  This might be hip people's fault
// for not letting C be hip; someone attempted to have a CPAN-for-C
// named CCAN: http://ccodearchive.net/ which is small and I haven't
// tried but I wish that became easy and hip.  Or maybe C just
// isn't as good a language for writing libraries?

#include <stdio.h>
#include <string.h>
#include <assert.h>
#include <malloc.h>

// C does not have hashes or binary search trees built in.
// I'll just use linear-time searches for this example,
// though a good C program should use a library rather than
// have the overall program be n^2 complexity.
// ... actually, I didn't even do any linear searches, instead
// I just made the output slightly different.

// It also does not free memory, because the program terminates
// promptly anyway.  The next time I write a program like this,
// I should use Go, because it is at a similar abstraction level but
// uses garbage collection.

size_t countCharOccurrence(const char * text, char counted) {
    size_t count = 0;
    while(*text != '\0') {
        count += (*text++ == counted);
    }
    return count;
}
size_t countCharOccurrenceRange(const char * text, const char * end, char counted) {
    size_t count = 0;
    while(text != end) {
        count += (*text++ == counted);
    }
    return count;
}

struct row {
    const char * id;
    const char * name;
    const char ** parents; // null-terminated
    const char * birthdate;
    const char * deathdate;
};

const char * alloc_str(const char * begin, size_t len) {
    char * str = malloc(len + 1);
    memcpy(str, begin, len);
    str[len] = '\0';
    return str;
}
const char * alloc_str_range(const char * begin, const char * end) {
    assert(begin <= end);
    return alloc_str(begin, end - begin);
}

struct row parseRow(const char * begin, size_t len) {
    // The error handling here is pretty bad.
    const char * id_end = strchr(begin, ';');
    const char * name_end = strchr(id_end+1, ';');
    const char * parents_end = strchr(name_end+1, ';');
    const char * birthdate_end = strchr(parents_end+1, ';');
    const char * deathdate_end = begin + len;
    assert(deathdate_end > birthdate_end && "Not enough fields in row");
    struct row result;
    result.id = alloc_str_range(begin, id_end);
    result.name = alloc_str_range(id_end+1, name_end);
    result.birthdate = alloc_str_range(parents_end+1, birthdate_end);
    result.deathdate = alloc_str_range(birthdate_end+1, deathdate_end);

    size_t num_parents = 1 + countCharOccurrenceRange(name_end+1, parents_end, ',');
    result.parents = malloc((num_parents+1)*sizeof(const char *));

    const char * parent_begin = name_end+1;
    size_t parent_num = 0;
    if(parent_begin != parents_end) {
        for(const char * i = parent_begin; i != parents_end; ++i) {
            if(*i == ',') {
                result.parents[parent_num++] =
                    alloc_str_range(parent_begin, i);
                parent_begin = i+1;
            }
        }
        result.parents[parent_num++] = alloc_str_range(parent_begin, parents_end);
    }
    result.parents[parent_num] = NULL;
    return result;
}

// returns a malloc-allocated array of rows
// and puts the count of rows in 'len'
struct row* parseCsvText(const char * text, size_t* len) {
    const size_t num_lines = countCharOccurrence(text, '\n');
    // possibly an overestimate, if there are blank lines:
    struct row* rows = malloc((num_lines+1)*sizeof(struct row));
    size_t row_num = 0;

    size_t line_begin = 0;
    for(size_t i = 0; text[i] != '\0'; ++i) {
        // If the file had \r\n endings this would make the last
        // field have a \r (BUG!)
        if(text[i] == '\n') {
            if(i == line_begin) {
                continue;
            }
            rows[row_num++] = parseRow(text + line_begin, i - line_begin);
            line_begin = i + 1;
        }
    }
    *len = row_num;
    return rows;
}

// I’m not bothering to realloc or anything.
#define ARBITRARY_FILE_LENGTH_LIMIT 100000

int main() {
    char data[ARBITRARY_FILE_LENGTH_LIMIT+1];
    size_t text_len = fread(data, 1, ARBITRARY_FILE_LENGTH_LIMIT, stdin);
    data[text_len] = ’\0’;
    // Even in my laziness I don’t like silent incorrect behavior (truncating).
    assert(text_len < ARBITRARY_FILE_LENGTH_LIMIT && "file too long for us!");
    size_t rows_len;
    struct row* rows = parseCsvText(data, &rows_len);
    puts("digraph \"family tree\" {\n");
    for(size_t i = 0; i != rows_len; ++i) {
        printf("\"%s\" [label=\"%s\"]\n", rows[i].id, rows[i].name);
        for(size_t j = 0; rows[i].parents[j]; ++j) {
            printf("\"%s\" -> \"%s\"\n", rows[i].parents[j], rows[i].id);
        }
    }
    puts("}\n");
    return 0;
}
```

23.2.B.Haskell

```haskell
-- This is a fairly straightforward adaptation of 2B.js.
-- I looked for a CSV library to show how great it is to use libs
-- (in any language) but the one I saw didn't support arbitrary
-- separator characters (i.e. semicolons).

-- After writing it, I was bemused by how many fewer lines the
-- parsing functions were, even though they are saying the same
-- thing.  I think it's a combination of currying, a shorter
-- lambda than function(){return}, and in parseFamilyTreeIntoRows,
-- pattern matching.
-- A line in transformRowsToGraphvizDotText shows that I wrap lines
-- a bit less in Haskell, a combination of style habits and how
-- Haskell's libs are slightly more concise (more functions and
-- they don't have _. at the beginning of them).
--
-- Also, it ran on the first try after fixing type errors.
-- And it was fun to write.  Blargh I am too good at this.

-- usage:
--   cat 2B.csv | runghc 2B.hs | dot -Tpng | display -

import qualified Data.Set as Set
import qualified Data.Map as Map

split :: Char -> String -> [String]
split sep = go []
  where
    go :: String -> String -> [String]
    go accum [] = [reverse accum]
    go accum (c:cs)
      | c == sep  = reverse accum : go [] cs
      | otherwise = go (c:accum) cs

parseCsvText :: String -> [[String]]
parseCsvText = map (split ';') . filter (/= "") . split '\n'

data Row = Row {
  rowID :: String,
  rowName :: String,
  rowParents :: [String],
  rowBirthdate :: String,
  rowDeathdate :: String
}
parseFamilyTreeIntoRows :: String -> [Row]
parseFamilyTreeIntoRows text =
  map mkrow (parseCsvText text)
  where
    mkrow [rid, name, parents, birthdate, deathdate] =
      Row rid name (split ',' parents) birthdate deathdate

transformRowsToGraphvizDotText :: [Row] -> String
transformRowsToGraphvizDotText rows = let
    describedPeopleIDs = Set.fromList (map rowID rows)
    mentionedPeopleIDs = Set.fromList (map rowID rows ++ concatMap rowParents rows)
    rowsByID = Map.fromList (zip (map rowID rows) rows)
    showIDInfo personID = case Map.lookup personID rowsByID of
      Nothing -> ["\""++personID++"\" [label=\"??\"]"]
      Just row -> ["\""++personID++"\" [label=\""++rowName row++"\"]"] ++
                  map (\parentID -> "\""++parentID++"\" -> \""++personID++"\"")
                      (rowParents row)
  in
    unlines (
      ["digraph \"family tree\" {"] ++
      concatMap showIDInfo (Set.toList mentionedPeopleIDs)
        ++ ["}"]
    )

main :: IO ()
main = do
  csvText <- getContents
  putStr (transformRowsToGraphvizDotText (parseFamilyTreeIntoRows csvText))
```

23.3 language tradeoffs

```
question 3 (20%)
----------------

Discuss the strengths and weaknesses of these programming languages as
you see them. What sorts of problems or situations are good fits to
these languages, and why? Be specific, giving examples that justify
your comparisons and conclusions. (This may well cover some ground
you've already discussed in the previous two problems.  If so, you
don't have to repeat any of that, just refer back to it and bring up
anything that you feel hasn't yet been brought forward.)
```

Response

A lot about a language depends on its ecosystem.

If a language has lexical scope, first-class functions, and garbage collection, I have a basic level of happiness with it. The lambda calculus embeds into it. Almost every popular language has these, except C, and C++ lacks garbage collection.

Garbage collection, like malloc()/free(), has complicated performance properties. Realtime code might want to avoid or regulate it (where realtime means “deadlines”, not “fast”). That’s a reason one might write in C or C++. Also, C and C++ have compilers with excellent optimizations, and excellent profiling (my favorite is Valgrind’s Callgrind tool with the KCachegrind visualizer). Because they allow specifying data layout in memory, it is practical to optimize your code’s cache-friendliness without weird contortions. I think Go might also have a predictable data layout, but most languages don’t discuss it or have flexibility. This is probably more fundamental than the compiler optimizations; CPUs are faster than RAM and even dynamic-language JITs (the best ones) are getting towards the same order of magnitude speed as C (Emscripten, which can compile C and C++ to Javascript, says in its FAQ “Q. How fast will the compiled code be? A. Right now generated code is around 3-4 times slower than gcc -O3, however, note that there are substantial differences between benchmarks, and in some cases JS engine bugs cause significantly poorer performance. [...]"[167]). In fact, I am still unsatisfied with the C/C++ cache profiling tools I've found – I want a tool to tell me which functions to blame for trashing the cache (which makes *other* functions slower later on) – but I think I'm more likely to find them for C/C++ than anywhere else.

This tooling has next to nothing to do with the language, and everything to do with history.

This ability to optimize is an important reason we chose C++ for Lasercake, a three-dimensional simulation game with lots of moving things in it.

On another subject, Python, Haskell, Ruby, and Javascript all have lots of easy-to-install libraries, an ecosystem I'm happy with. Javascript is new to this category. (Perl should be here but every time I've tried using CPAN, installing packages has been a fight; I hear other people think the same about Haskell's Cabal system.)

In C and C++, there is rigamarole for every library. Build systems are always important. They vary. Linux distributions don't want libraries bundled with your code. Casual developers might. There is no central repository. Boost (boost.org) has a lot of C++ libraries that let your code be better or completely inscrutable. Despite C++ being capable of many functional memes, like persistent data structures or simply nonmutating algorithms, there are shockingly few people who have actually written good ones. I hope this gets better with C++11 lambdas and initializer-lists and maybe someday garbage-collection. (Reference counting is... mostly okay. Refcounting is slower than GC. Occasional algorithms call for circular references that need GC. I have the feeling that recursion might involve circular references if functions are data, as they certainly are when they have closures). On the plus side, they are a core technology of major OSes and also one of the few good ways to write a library that many languages have bindings to.

Every language has quirks. Ruby has at least three different kinds of functions. Javascript

[167] `https://github.com/kripken/emscripten/wiki/FAQ`

binds ‘this’ in functions that shouldn’t, and has slightly wrong scope for local ‘var’ bindings. Haskell’s record system is even worse than C’s (though still useful), and people have been trying to improve it for years. Python doesn’t have multi-line anonymous functions; I haven’t tried but this would make me nervous about writing evented-style code in Python. C’s implicit numeric casting has strange pitfalls. C++ templates form a Turing-complete compile-time language, which is useful but the syntax is incredibly terrible for the purpose because no one realized it was Turing-complete until after they’d created it. Scheme’s syntax-rules hygienic macro system was discovered to be not quite as hygienic as needed in all cases, and another syntax-* was invented to work for those, if I recall correctly. Go shares the peculiar distinction with C of having no natural way to express parametric polymorphism. Anyway, quirks are generally tolerable (well, the lack of parametric polymorphism is something I have difficult forgiving, but it isn’t really a quirk per se).

Each language has its own idioms to do things. You *could* write the code of 2B.hs:split, or 2B.c:countCharOccurrence, in another language, but it would hardly be idiomatic. It takes time to learn these idioms; it takes reading and writing code.

I like Haskell when I have a problem (like parsing or many others) where thinking about the exact semantic representations I want for data is half the challenge. 2.A. was such a problem.

I like Javascript when I want to be able to hack up an interface easily and share it with anyone. I like how it is not too terrible at any paradigm (it has functions and objects quite effectively), even though it is riddled with problems that make it harder to do things 100% correctly. Luckily, the *JavaScript: The Good Parts* principles help a lot. Almost-but-not-quite-right is *not* something we want for security-sensitive code (nearly anything exposed to the web qualifies). Sticking to “the good parts” makes the equality operation not coerce types in confusing ways, for example, and avoids pitfalls of the “new” keyword. It can’t make JavaScript strings be a proper UTF-16 implementation, alas.

I like how C++ has lots of system-level libraries and I know its many paradigms well. There is usually a paradigm for any given thing I want to do. (I've been a little frustrated with it's implementation-hiding abilities, which are: private/protected class members, and things written in a single implementation file and shared no where else. It keeps being not quite right and I want something better though I'm not sure what it would be. I'm quite happy, however, with C++'s support for polymorphism both through templates and through inheritance.)

My success-rate at bug-less code after the first compile is Haskell > Javascript > C++. Even if I count from when the C++ compiles and is debugged of memory errors (segfaults etc.), I do a conceptually wrong thing in it more often. JavaScript's inferiority to Haskell here is completely due to Haskell caring about types: it has no implicit coercions, it encourages you to declare types to explain your problem, and it even tells you up-front if you've done something wrong. I can't really count the static typing as a fundamental difference, although it probably is for big projects. In small JavaScript pieces, I find the type errors at runtime pretty quickly. But I find it harder to determine what I've done wrong from a dynamic type error amongst un-type-annotated code. In any language, sometimes it is a silly mistake and sometimes it is pointing out that I've thought about the problem all wrong. I am good at seeing that in Haskell. But I'm sure that's not true for everyone; people who grew up with dynamically or weakly typed languages are probably frustrated by Haskell's rigidity. I remember once being confused at how every member of a Haskell list must be exactly the same type. From a later perspective I was confused about how a heterogeneous list could even be used for anything meaningful, even if it were allowed. Theoretically, many dynamic language type errors could be caught before runtime, (not all, but many); I'm not sure how much this has been pursued theoretically and practically. There's some work on optional typing in the Lisps I haven't read. Python has three checking tools, pychecker, pylint, and pyflakes, that catch some things. C++ compiler warnings catch some semantic errors by dint of them manifesting as using uninitialized variables or the like.

Python was pretty good for collaborating at Marlboro because we all knew that language. Also, its documentation is the best. That allowed those of us who knew programming in other languages better to pick it up pretty easily. Pair programming with documentation in the observer's hand and code in the programmer's hand is pretty sweet.

Languages with more pitfalls are riskier to just jump into, try familiar-seeming things, and see if they work. I am especially unlikely to trust myself with PHP because it has many pitfalls, which I do not know well, and if I do something wrong it has security consequences (internet!). Other than that sort of risk, I'm fairly willing to try languages if there's a reason (or if it seems fun).

24 Linux exam

```
computer science plan exam : linux
===========================================

  * for Isaac Dupree from Jim Mahoney
  * out: by Fri Nov 15
  * due: end of day  Fri Nov 22

This is open take-home exam: books or web sources are OK as long as
the problem doesn't set other constraints, you cite them explicitly,
and that they aren't a drop-in solution to the problem. However, the
more your answer is a summary of someone else's article, the less we
will be impressed. Don't ask other people for help. Don't just give a
numerical result, give an explanation.  Your job is to convince us you
understand this stuff.

Be very explicit about which sources you used for each problem.

As always with my exams, if you think there's a mistake in one of the
questions or it doesn't make sense, you can (a) ask for clarification,
and/or (b) make and state an explicit interpretation and do the
problem that way.  (Again: the point is to show your mastery,
not to get the "right answer" per se.)

Good luck.
```

24.1 everything is a file

```
question 1
----------

One of the central ideas of the unix operating system is the
"everything is a file" concept. Explain what this all about: what
things are files, what aren’t, and the positive and negative aspects
to this approach.  Include the unix permission system in the
discussion. And also describe what you think are some of the
noteworthy file-related command line tools.
```

Response

Sometime after files and directories (folders) were invented, someone had the idea: what if every sort of system resource could be accessed as a file?

For example, Linux has special “device files”, typically in /dev. They mostly refer to things like disk partitions or USB mice. These files take up very little disk space. Opening a device like a disk partition has a file-like interface: it is a fixed length, but it can be read from, written to, seeked within, etc. Devices like mice have more peculiar interfaces, obviously. They might use file-reading to output the mouse-movement data in some format. They might use ioctl() syscalls. They might use some other mechanism.

(Some of these device files, like disks, are “block devices”, and some are “char devices”, which affects their interface slightly.)

Some “device files” are special virtual files. /dev/null can be written to for no effect, and reading from it always gives end-of-file. /dev/zero can be read for an infinite stream of zeroes. /dev/stdin is a way to refer to the current process’s standard input stream; similar for /dev/stdout and /dev/stderr; on my system these are symlinks to /proc/self/fd/0,1,2 respectively (see later). /dev/mapper/* are created by the device-mapper out of other devices; for example, if you have an encrypted partition, the version on disk might be

/dev/sda3, and if you've opened it then you might be able to access the data through /dev/mapper/mydisk or the like.

A convenient thing about device files is their filesystem permissions can be set (typically via udev configuration, which manages /dev for you). This unifies the concepts of user access between regular files and other things. That reduces the amount of concepts a sysadmin might have to know. That allows file-permission innovations like ACLs to apply naturally to device operations. These are an advantage of "everything is a file".

Most kernel-named things on Linux have an entry in /dev. Network interfaces do not. Ethernet is commonly 'eth0', wireless 'wlan0', and the system talking to itself 'lo', but these names do not appear in /dev. They might be referenced by syscalls. "Everything is a file" is applied inconsistently in Linux.

Also, in Linux, there are some special magical filesystems, such as /proc and /sys. To userspace, /proc/sys/vm/drop_caches, say, appears to be a normal file. But writing it with the string 1, 2, or 3[168] causes Linux to forget about the caches from disk that it's currently holding. Reading the file is pointless, and you can write it any number of times to repeat the effect. /proc/1/comm contains the name of the binary of PID 1 (which happens to always be the init process). You can use any PID number, or 'self' for the current process. I believe 'comm' is writable if you have permission – it changes what shows up in 'ps', for example). /sys is various raw information about the states of devices on your system, some of it writable.

For special or magical files that are sensibly readable or writable with textual data, it's easy to control them via the command-line. This is an advantage of "everything is a file", or a disadvantage that the shell has no easy way of doing syscalls, depending how you look at it. (Perl, by contrast, tends to have ways to do both.)

Linux also supports disk files that are 'sockets' and 'FIFO pipes', which take up little

[168] http://unix.stackexchange.com/questions/17936/setting-proc-sys-vm-drop-caches-to-clear-cache

space and are ways for processes to coordinate communicating with each other. These are not the only way for processes to communicate with each other.

Symlinks are special files that contain a text string that is a Unix path relative to the directory the symlink is in (or referring to /) that is where the symlink "points". If it points to a directory, the symlink can be included in paths as if it's the target directory. If it points to a file, it can be opened as if it's that file. It can also be explicitly checked as to whether it's a symlink, and modified as such.

Hard links are not special files, they're just acknowledgement that on Linux there can be two or more names for the same physical file on disk. All names of a file have equal standing. The physical file exists until all names have been removed (rm / unlink()) and all processes that had the file open have closed it. Directories typically cannot be hard-linked because it creates complications like a directory that is inside of itself on disk.

Then there are bind-mounts, where, if you're root, you can bind one directory tree on top of another directory. This is very much like making the second directory a symlink to the first one, except that there are different mechanisms for a program to detect this sort of alias, and it doesn't rely on the textual path remaining the same.

Links are sometimes useful. We could probably get by without them if they didn't exist.

Elevated permissions are sometimes file permissions ('setuid' bit), a part of "everything is a file" that is mostly bad. Some privileges these days are instead doled out by processes at runtime, e.g. by sending a message to init asking if it'll do something for you. Depending on who you are and what you say, it might oblige or refuse.

Fedora replaced setuid with specific filesystem capability bits (keeping the permissions-as-file-metadata but possibly reducing security risks if a setuid program has a bug). Openwall removed all setuid programs in favor of a few special setgid programs and architecting the rest of the system so that setuid privileges aren't needed (still better, but still filesystem permissions). NixOS eschews filesystem elevated permissions entirely because it's contradictory to its principle that installing a piece of software, or a different version of a software, should have no privilege effect. As a consequence, NixOS can leave old software versions with security flaws sitting on the disk until they're cleaned up, and their harm is limited to the UNIX users who are foolish enough to run them. Citations: e.g. `https://lwn.net/Articles/416494/`

24.2 distros

```
question 2
----------

Pick a distro that you are particularly familiar with and/or fond of.
Discuss in what ways it is different from and similar to other
common distros, including at least the file system layout, boot
procedure and system daemons, and the package manager.
```

Response

I'm fond of Arch Linux.

24.2.i Daemons

Arch Linux assumes you don't want to run a daemon unless you add it to the list/set of daemons to be run. Debian, by contrast, assumes that if you installed a package that has a daemon, that you want to run that daemon unless you specifically configure it not to run. I like Arch's approach better personally.

Arch's approach is used by OpenBSD for security reasons. This forces the sysadmin to be conscious of what services are running. This also prevents a race condition between installing a daemon and hardening its configuration. For example, SSH comes default configured to allow password-authentication on many distros, which I consider to be an insecure default (anyone on the Internet who can connect to your computer's port 22 can try bajillions of passwords if they have time, and the typical entropy of a Unix password is considerably less than the entropy of an SSH key). [I couldn't remember whether SSH defaulted to port 22 or 23 so I looked it up on Wikipedia.]

Arch's approach is used by NixOS because it has to be. Nix is a purely functional package manager in which each package is installed to a different prefix under /nix/store/. Different user environments can be created by creating symlink trees to all the packages that this user environment desires. This setup allows several different versions of the same package to be installed to disk at once. This makes rollbacks trivial. It helps but does not completely solve dependency hell for packagers. In any case, having a package installed in Nix must not have any effect on parts of the system that don't reference that particular package. That includes system configuration. That includes the fact that NixOS cannot have setuid programs because then there would be security implications in which third-party packages are currently installed vs. not.

Debian's approach is used by several distros, including Debian derivatives like Ubuntu, and non-Debian distros like I believe Fedora. This approach is probably more user-friendly for people who aren't super into tinkering with their system.

24.2.ii Filesystem layout

I mentioned how NixOS has a strange layout. Most Linux have about the same layout, which is roughly described by the Filesystem Hierarchy Standard (FHS). Kernels go in /boot, libraries in /lib and /usr/lib, binaries in /bin and /usr/bin, non-executable program data in /usr/share, system configuration files in /etc, user home directories in /home, system runtime data in /var, and so on. Kernel runtime information is in the magic filesystems /proc and /sys, and /dev is also about talking to the kernel (though its magic is in "device files" rather than a special "filesystem").

/dev and /proc are also found in BSD, I believe (though with not quite the same contents); /sys is a Linux invention.

Mac OS X and the defunct GoboLinux have user-friendly names for some of these directories, like /Users rather than /home. OS X uses but hides important Unix directories like /usr.

/tmp is often run as a tmpfs - a filesystem that's backed by RAM and thus its contents disappear whenever the computer is shut down. A few people don't like this because it means you can't easily put 10 GB files in /tmp, but it means that /tmp doesn't need to get cleaned out by boot scripts for systems that reboot at least occasionally, and it often makes /tmp accesses faster.

Lennart Poettering has been leading several minor changes to the Linux layout. Some people don't like this, but I'm a fan of most of the changes. For example:

- moving /var/run to /run (and leaving /var/run as a symlink to /run) because this makes the boot process better. /run can be needed before /var is mounted, when /var is on a separate filesystem.
- noticing that boot is already subtly broken in many ways on systems whose /usr is a separate filesystem mounted later than / is, and declaring that deprecated.
- making /bin, /lib, etc. be symlinks into the equivalent directories in /usr, and installing everything in /usr. This simplifies things. Solaris has done this since a long long time ago. I know that Arch and Fedora have adopted this change.

24.2.iii Package manager

Most modern Linux distros have a package manager, which takes on the NP-complete task of, given an instruction to install something, looks at the thing's dependencies, alternate dependencies, packages it cannot be installed at the same time as, and perhaps some other rules, and produces a suggestion of what packages to install and remove in order to meet the user's bidding.

For example, Amarok (a music player)'s GUI is written in the Qt library, so Amarok depends on Qt being installed (and many other libraries). This differs from Mac and Windows where every program that uses Qt would typically ship its own copy of the library, and each bundle would be downloaded from the Internet, except for programs and libraries that come with the base system.

Package managers also allow updating to new versions, removing packages, etc. They get packages from repositories run by volunteers (or occasionally paid people) of scripts that show how to build that package on this distro. Usually these repositories include a pre-built binary of that package, so that not every user has to wait for their computer to compile every package from source; Gentoo is the main exception.

A few popular Linux distros don't come with a package manager + full set of repositories, including Red Hat Enterprise Linux (besides the semi-official EPEL `https://fedoraproject.org/wiki/EPEL/FAQ`) and Openwall GNU/*/Linux (whose aim is to provide a core security-hardened system).

Each Linux distro handles upgrades differently. Arch is "rolling release" - Arch developers package and build new versions of software and then everyone must upgrade (or be unsupported, have security flaws in old versions be unfixed, and be unable to upgrade even a few packages without upgrading others). Ubuntu and Fedora have major releases every six months. The user can explicitly upgrade to the next major release. Meanwhile, these users are supported by continuous security-and-bugfix-only updates to their packages for some time (e.g. 18 months after the major version is released). This latter kind of security-fix update is less likely to break things, and less likely to confuse users. Debian has major releases but not very much on a time-based schedule; each major release happens when it's ready, usually 1-3 years after the last release. Gentoo is somewhat like Arch but tries to support multiple versions of a package, giving users choice at the expense of the amount of testing that developers can give their specific combination of packages.

Arch Linux is good for reporting bugs to upstream projects because it tends to package up-to-date versions of software and not patch them very much. This means that bugs are less likely to be already fixed, or the distro's fault.

24.2.iv Boot/system management

There are several init systems. These involve the first process the kernel starts when the system boots. SysVinit has long been common on Linux; it is configured by collections of shell scripts. There are a few variants of this general idea; e.g. Arch had a custom init configuration for a long time. SysVinit has some limitations. Some people wanted boot to happen faster, so they made ‘initng’, in which boot services that do not depend on each other can be started in parallel; some enthusiasts used this. Then Canonical (Ubuntu’s company) noticed that the traditional idea of sysadmin-configured daemons didn’t fit so well with diverse laptops, frequent hotplugged devices (e.g. cameras), and such things of today’s world. They made an init system called Upstart which deals better with the dynamism of the world and also starts things in parallel where possible. Ubuntu used this. A few other distros made it possible for interested users to switch to Upstart. A few years later came the competing init project “systemd”, led by Lennart Poettering of Red Hat. It focuses on using Linux features well to *manage* daemons, not just start them, assume they’re well, and be unable to robustly end them or the like. (I think Upstart may manage daemons decently too; I’m not informed on all the details.) Systemd also attempts to force its adopters to use consistent configuration layouts in /etc, because no one benefits from the way it’s slightly different from distro to distro. This is open-source, so they’re losing that battle in Debian (who will patch it), but being fairly successful otherwise. Arch Linux changed from its custom init system to systemd recently; Fedora has adopted it, naturally, as it is affiliated with Red Hat. I find systemd delightful to use and to configure, but people who have long been good at configuring SysVinit grumble.

24.2.v Community

Debian is a democracy of its developers (who are the people who write package-manager packages and do the many other tasks a distribution needs, are mostly volunteers, and number above 1000). They have endless arguments and “whenever we have to make a choice, we usually choose all of the options at once” [paraphrased because I can’t find the original source I remember it from], e.g. trying to allow the user to use any of the init systems that they want to.

Ubuntu is derived from Debian and collaborates on the collection of packages often. Ubuntu, in the end, is controlled by the for-profit company Canonical, though it has a significant community in practice too.

Fedora is community-governed in a way that as far as I’ve seen is pretty effective. Red Hat is like a benevolent mommy to Fedora.

Arch Linux is controlled by its developers (there are about 50, I think all volunteer). They are technically competent. The wider Arch community is kind of latently misogynistic, so I don’t read the email list anymore.

NixOS is mostly people in academia.

24.3 write a shell script

```
question 3
----------

Write a shell script that removes files ending in .log older than one
week from a given folder and all those below it.
```

Response

```
#!/bin/sh
if [ $# = 0 ]
then echo "usage: ./clean-old-logs dir" >&2; exit 1
fi
find "$1" -name '*.log' -a -type f -a -mtime +7 -delete

# -type f: don't remove directories that end in .log: they probably
#   shouldn't exist but if they do, we shouldn't cause havoc with things
#   we don't understand.
```

I referred to ‘`man find`‘ to remember the correct relative-time argument and the name/existence of -delete (I was going to use ‘`-execdir rm '{}' +`‘ if it didn't exist).

Test dir I made is attached. I created the test files with ‘`touch`‘, the dir with ‘`mkdir`‘, and the older files with ‘`touch -rSomeOlderFile`‘ for some older file I found on my system.

How to re-create the test directory[169]

```
mkdir linuxq3test{,/quux}
touch linuxq3test/{bar.bog,foo.log,quux/{blog,blog.log}}
touch -d '1 January 2000' linuxq3test/{fool.log,log,quux/{barl.log,not-a-blog}}
```

[169] This subsection was written after the exam period. When putting the Plan together, I wrote this section as a way to have all of the Plan's contents visible in the printed copy. This script is also superior to the .tar.gz I'd made because the .tar.gz's recently modified files are no longer recently modified. This script's creation is good for testing the above script for about seven days after creation. It is a failing in my exam performance that I did not create it at the time.

24.4 detecting illicit intrusions

```
question 4
----------

a) What are the signs that might make you think someone had gained
illicit access to a machine that you administer?

b) What are the things you would you do to tell if in fact the machine
had been hacked, and what might those things show you?

c) What would you do next?
```

Response

24.4.a Signs of illicit access

Weird things happening. For example, unexplained CPU usage, network usage, processes that weren't there before, suspicious* messages in system logs. *It turns out that the best definition of "suspicious" is often just "different from what's been seen before" and there is software that does fuzzy matching on logs based on this principle. I forget the details.

For example, I monitor my Linux laptop system all the time, and unfortunately it tends to have a large amount of weird things. Most of them are explainable: browsers just use a ridiculous amount of memory. Linux can have known bugs on some hardware. My hardware is not completely reliable (my latest laptop sometimes crashes if you plug or unplug the power cord while it is using the CPU/GPU at full force; an old Mac/Linux desktop I had would occasionally and unrepeatably segfault GCC, which was probably a hardware issue because it happened on many GCC and OS versions and I haven't seen it anywhere near as much on other hardware).

Other sorts of signs: running software you know is insecure, connected to networks you know are malicious (for example, old versions of software on a server that's connected to the

Internet). If you do that, it's likely that a completely automated, untargeted attack might compromise your server.

Other signs: if you see non-public data being used somehow, maybe it was stolen. For example, DigiNotar was an SSL certificate authority. It was hacked; people found out when a hacker generated a fake google.com certificate and used it somewhere publicly. That can't be done without private crypto keys from DigiNotar. Sometimes it's more subtle, like a gamer might be able to improve their reflexes in an online game if they had knowledge from the servers that they wouldn't normally be able to see. If one of your coworkers looks sheepish, maybe it was them. If the attack involved social-engineering of you or your buddies (e.g. calling you, pretending to be a customer needing help with the system), maybe you'll realize afterwards that something was suspicious about that.

24.4.bc Ways to investigate potentially being hacked

You can look at logs, monitor network traffic, look at active processes, and so forth on the potentially hacked computer. This might find things. (It also might show you a non-malicious cause of the odd system behaviour.) If the attacker has gotten root on your system, then this is unreliable, because they can make the system lie to you however they want to (if they are skilled and/or have good hacking software). If they can run arbitrary code as a user, then they can probably get root: the Linux kernel frequently has security holes that allow a local user to gain root. This is not a Linux-specific fact; this is why so many "locked" gaming consoles and cell phones can be "rooted" by their users/owners. Nearly every large program has flaws. Programs that can do a wide range of things are more easily attackable: for example, Linux has hundreds of system calls, and a mistake in any code-path from any one of them could be a security risk.

If you suspect they have root, monitor from known-good (or as best you can get) systems. Booting from read-only media such as CDs/DVDs that were created a long time ago can be helpful (though there are sometimes ways to have rootkits that boot CDs inside a hypervisor - I don't know details about this risk). A separate system might be able to monitor the suspect system's network traffic using Wireshark. If log messages are not just stored to disk but sent to a separate logging server, you can look at that: any message before the compromise will be real. You won't know at first *when* the compromise happened, but there might be a few suspicious messages just before the compromise was completed that the attacker couldn't prevent from being sent.

After a compromise is detected, see if you can find what security flaw they entered through (via logs or other careless things they left around) so that you can fix it before next time. Then, if it is a serious enough compromise that you don't trust your system (arbitrary code execution probably is; XSS such as posting to a web service in a way that could steal other users' passwords might not be, if nobody important logged in in the meantime), reinstall from media you trust. Run rootkit detectors (e.g. rkhunter, chkrootkit) because there's a chance they will detect something.

Identify any passwords used on that machine and assume they are compromised. The attacker could try to break the password hash that's stored on disk (effective for poorly hashed and/or low-to-medium-quality passwords), or wait until somebody logs in and steal their password then (effective unless you consistently use a non-password-revealing authentication mechanism such as SSH public/private keys or SRP[170]). Identify any other machines that that one had access to (e.g. via SSH keys or possibly shared passwords) and reinstall them too. Kernel.org was hit by a non-targeted attack in 2011 that used this mechanism to break from one machine into the next, and so on.[171]

Remember that if you can't identify when the compromise happened, your backups might be compromised too. Backups are just data, so a non-compromised system can at least look

[170] `http://srp.stanford.edu/`

[171] `https://lwn.net/Articles/464233/`

at the data without being lied to (unless the data is e.g. a PDF being viewed by a version of Acrobat Reader that can be hacked by viewing malicious PDF files, or many similar file-viewer vulnerabilities). If you find out that an entity with a hundred times more resources than you (e.g. the CIA) is targeting you specifically, you might want to give up or have more cautious ambitions rather than restore what you had before.

24.5 do a security audit

```
question 5
----------

Evaluate either cs.marlboro.edu or csmarlboro.org for security holes
for (a) users with a user account, (b) other on campus users, and
(c) off campus strangers.

(Please don’t break cs.marlboro.edu - that’s a production machine.
Downtime would inconvenience many people.)
```

Response

I’ll start with what anyone on the Internet can do.

(My shell prompt is %)

```
% nmap cs.marlboro.edu

Starting Nmap 6.01 ( http://nmap.org ) at 2012-11-16 12:57 EST
Nmap scan report for cs.marlboro.edu (10.1.2.19)
Host is up (0.0040s latency).
Not shown: 995 filtered ports
PORT     STATE  SERVICE
22/tcp   open   ssh
25/tcp   closed smtp
80/tcp   open   http
443/tcp  open   https
3000/tcp closed ppp
```

```
% nmap csmarlboro.org

Starting Nmap 6.01 ( http://nmap.org ) at 2012-11-16 12:56 EST
Nmap scan report for csmarlboro.org (96.126.107.158)
Host is up (0.078s latency).
rDNS record for 96.126.107.158: li364-158.members.linode.com
```

```
Not shown: 988 closed ports
PORT     STATE    SERVICE
22/tcp   open     ssh
80/tcp   open     http
111/tcp  filtered rpcbind
135/tcp  filtered msrpc
139/tcp  filtered netbios-ssn
161/tcp  filtered snmp
445/tcp  filtered microsoft-ds
1080/tcp filtered socks
1433/tcp filtered ms-sql-s
1434/tcp filtered ms-sql-m
3372/tcp filtered msdtc
5000/tcp open     upnp
```

Now we know csmarlboro.org is hosted on Linode (from the reverse DNS). Adding a few nmap flags (-sS, -O) didn't tell me any more besides that it's Linux, which I could have figured.

```
% curl --head http://cs.marlboro.edu/
HTTP/1.1 200 OK
Date: Fri, 16 Nov 2012 17:47:40 GMT
Server: Apache
Content-Type: text/html; charset=ISO-8859-1
```

This probably reveals something about the Apache version but I'm not sure what. I know it's Apache 1.3 but attackers might not (but they could try any relevant exploits anyway).

```
% curl --head http://csmarlboro.org/
HTTP/1.1 200 OK
Date: Fri, 16 Nov 2012 17:57:50 GMT
Server: Apache/2.2.17 (Ubuntu)
Last-Modified: Mon, 16 Jul 2012 22:18:20 GMT
ETag: "3dd25-2f4-4c4f9ceaa2300"
Accept-Ranges: bytes
Content-Length: 756
Vary: Accept-Encoding
```

```
Content-Type: text/html
X-Pad: avoid browser bug
```

This tells the server version, and that it's Ubuntu. Ubuntu's website tells me[172] that no Ubuntu version has that exact Apache version, so either it's an un-upgraded Ubuntu or a custom install of Apache on it. For both websites, neither

- https://httpd.apache.org/security/vulnerabilities_13.html
- https://httpd.apache.org/security/vulnerabilities_22.html

told me any vulnerabilities juicy enough for me to bother with them.

From a non-Marlboro server (I ssh'ed there) I can find out cs.marlboro's IP address

```
% nslookup cs.marlboro.edu
Server:         192.168.1.2
Address:        192.168.1.2#53

Non-authoritative answer:
Name:   cs.marlboro.edu
Address: 206.192.68.20
```

Double-checking,

```
% curl http://206.192.68.20/
```

...yep, that looks like cs.marlboro.edu, and why would someone fake or copy that site.

Now we can do GeoIP on that IP (e.g. http://www.geoiptool.com/) and find out that it's in Marlboro, VT. Well, we could have guessed, but it could plausibly be hosted somewhere else.

I tried to brute-force someone's SSH password using Metasploit (which I haven't used before so I've been learning about from the internet) and didn't succeed and after a couple

[172] http://packages.ubuntu.com/lucid/apache2

dozen login-attempts (spaced two seconds apart probably due to cs.marlboro sshd restrictions), sshd stopped responding to me. That blocking is probably by IP address, and I don't have a botnet; I'll give up on this. Still, if anybody has a weak password or an old Debian public-key login, that would be risky. I'm not sure if it's easy for someone with no logins and off campus to get a list of Marlboro usernames, but there's probably a way (and then there's guessing them).

Alright, the C software doesn't have any remote vulnerabilities that I know of or know how to exploit. Now to investigate web-server-level vulnerabilities (e.g. wiki JavaScript) and physical access.

Physical access: Somebody remote can find out via GeoIP that cs.marlboro is in the town of Marlboro. They can visit the college if they like. People will recognize them as a stranger. Not many Marlboro students even know where the CS machine is. I know it's in the room next to Jim's office but not which of the computers in there it is. Jim and Sam and probably a few other people know which one it is. The local IP network might have its 10.* addresses assigned by campus building, and if so, this attacker could probably deduce which building it's in. Still, they might look suspicious as a stranger poking around in the science building in a room that is sometimes locked.

The server serves some things differently to remote vs. on-campus users; for example, it serves a list of Marlboro people only to on-campus users. I haven't found a way to pretend to be local from a remote point. `https://libproxy.marlboro.edu/` only is a proxy for a fixed list of sites. CS is on-campus and works if you have an account. (Everyone who has an account is someone who is regularly on campus anyway, except Abe Stimson and a couple other recently graduated/transferred people, so that doesn't buy an off-campus user anything. That is, assuming the off-campus person can't find out about us and hack our computers to get CS credentials. That's probably possible for at least one of us, for someone with a lot of time on their hands, but I doubt that will happen.)

Maybe we can hack an arbitrary on-campus person: create and post a link to a malicious website to the Marlboro College Facebook group (or spam-email Marlboro peoples' emails and include a link; there is an off-campus-access LDAP database that shamefully reveals a lot of Marlboro emails.). Some people's laptop software will be vulnerable to that hack (or it could use social engineering: ask people to download a philosophy game about Halfway or something). Email might be more effective because the sender can be spoofed; pretend to be president Ellen or something. Or create a Facebook account that has the same name as an existing Marlboro student, and a cat picture; that student might not notice before the damage is done. This can be more effective than regular spam because it'll be tailored to Marlboro (assuming this attacker knows anything about Marlboro - which they can find out on our website and in Colleges That Change Lives).

Someone who cares and knows enough can just drive to near Persons Auditorium and use the WiFi there without being noticed.

However, all being on the on-campus network really buys you is being able to see the list of current students/staff/faculty on CS. That doesn't help (other than the fact that that itself is somewhat of a privacy violation).

Maybe some Marlboro student's password is easy to guess. Maybe the CS or Nook or similar web login systems have weaker protection than SSH does against brute-force login. It seems likely. Does that help? It could buy us two things: ability to edit some pages on the CS wiki, and passwords that might be the same as one of our SSH passwords to CS.

Editing pages on the CS wiki would be a defacement: partial success at hacking! As a wiki, it keeps history, so it could be reverted if anyone noticed, and someone could talk to the student whose laptop was hacked and perhaps cure their laptop and make them change their password. However, it is a wiki that (deliberately) allows editors to insert arbitrary JavaScript. This JavaScript could be used to steal more people's passwords: convince someone to log into CS while on the page you edited. Probably no one will look at most of the old CS pages you could edit, but you could post a link to that page on Facebook, and give people some incentive to log in (“Log in to play with this game I hacked together for class! Or to edit and add some dialogue lines.”), and because it would be actually on a Marlboro website, people would trust it more. With these passwords you can cause various havoc on campus but not much of it is CS-related, unless you get Jim’s password (he has WikiAcademia administrator privileges even if his SSH and web passwords are different). If you do get an SSH password...

```
cs% uname -a
Linux cs 2.6.32-42-generic-pae #95-Ubuntu SMP Wed Jul 25 16:13:09 UTC 2012
i686 GNU/Linux
```

That looks fairly well patched (maybe not perfect but definitely not terrible). There might be a Linux local root vulnerability on the black market, but there probably isn’t a publicly known one (it would’ve been fixed and a new kernel installed). Unfortunately, modern exploit economics dictate that you get richer by secretly selling an exploit rather than by responsibly disclosing it, so there are even more likely to be non-publicly-known exploits than in past decades.

Even if we don’t get root, we can be a nuisance.

We could run a forkbomb, effectively killing CS until somebody restarts it. Then we could log in again and start a forkbomb again, until somebody figures out what’s going on.

We can look at /etc/passwd, but the password hashes are safely in /etc/shadow, so we can’t crack them that way.

‘ps aux‘ doesn’t show me anything I know how to exploit. There’s an X server running (as root!); When I ‘env DISPLAY=:0 xev‘ the server correctly refused the connection. There are some Xauthority files hanging around (perhaps /var/run/gdm/auth-for-gdm-0xwJOf/database - on the X command line – or /home/sam/.Xauthority – but each of these correctly does not have read permission for me; /root is also not readable by me). Out of curiosity, I ran ‘startx‘ and got an interesting response:

```
cs% startx
xauth:  creating new authority file /home/idupree/.Xauthority
xauth:  creating new authority file /home/idupree/.Xauthority

X: user not authorized to run the X server, aborting.

^CInvalid MIT-MAGIC-COOKIE-1 keygiving up.
xinit:  Resource temporarily unavailable (errno 11):  unable to connect to X server
xinit:  No such process (errno 3):  unexpected signal 2.
```

We can ‘cat /proc/cpuinfo‘ and then maybe it’s a processor with disclosed security problems in its microcode. Those are far more common than they should be.

We can find out what devices are connected to the computer:

```
cs% lsusb
Bus 007 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 006 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 005 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 004 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 003 Device 004: ID 046d:c312 Logitech, Inc. DeLuxe 250 Keyboard
Bus 003 Device 003: ID 051d:0003 American Power Conversion UPS
Bus 003 Device 002: ID 046d:c018 Logitech, Inc. Optical Wheel Mouse
Bus 003 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 002 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
```

and ‘lspci‘ and ‘lshw‘. They’re not that interesting. It has FireWire; a FireWire device has full access to the computer’s memory, but that doesn’t help much (physical access gives us plenty of easy ways to hack it). It has audio that might be plugged into something... ‘cat </dev/urandom >/dev/dsp‘ points out I do not have permission to write to /dev/dsp. What if ALSA sound works though?

```
cs% cvlc /usr/share/evolution/2.28/sounds/default_alarm.wav
VLC media player 1.0.6 Goldeneye
[0x84838e0] inhibit interface error: Failed to connect to the D-Bus session daemon: /bin

[0x84838e0] main interface error: no suitable interface module
[0x83d3148] main libvlc error: interface "inhibit,none" initialization failed
[0xb6f00dd8] main interface error: no suitable interface module
[0x83d3148] main libvlc error: interface "globalhotkeys,none" initialization failed
[0x846bef0] dummy interface: using the dummy interface module...
[0x848c218] pulse audio output: No. of Audio Channels: 1
...
```

I'm not near CS physically so I have no idea if that's making noise. It's running! If I could make noise remotely, I could annoy people (and that's all; it's not good for anything). I suspect it isn't making noise; ‘alsamixer‘ says “cannot open mixer: No such file or directory” and ‘pactl stat‘ says “Default Sink: auto_null”.

But hmm! That pulseaudio server is running as ‘gdm‘, not as me. If I can hack it, I can gain some things (at least access to /var/run/gdm/auth-for-gdm-0xwJOf/database, which is owned by gdm and has something to do with that X server). We can do things with it: ‘pactl load-module module-remap-sink‘; ‘pactl list‘ to list loaded modules. This page `http://www.freedesktop.org/wiki/Software/PulseAudio/Documentation/User/SystemWide` claims that module loading might allow loading arbitrary code, but I haven't found how to do that (maybe they just mean that out of the hundred modules, maybe one has a vulnerability?). If there were a microphone attached (which there isn't) or another user using audio at the same time (there isn't), we could mess with those, because of PulseAudio's design.

Let's find out what world-writable things are hanging out on the disk: can I cause any havoc by writing one of them?

```
cs% find / -perm -o+w -a \! -type l > world_writable
```

(As an actual attacker I wouldn't make myself obvious by saving temp files to disk like that – saving them to a not-on-CS disk I control would be fine.)

/home/ezeidan/html/lava/ is world-writable... but the most I'm likely to get from that is hacking Elias's account, which is no more useful than mine.

I could make a script in /home/mahoney/html/private/misc_old_zonorus/abc/scratch in the vain hope that some day Jim will come back to it, be curious about the script and assume he wrote it and forgot it, and run it blindly. Or perhaps add a hack to one of the existing code files like /home/mahoney/html/private/misc_old_zonorus/misc/trust/html/trustee_screens.pm – which would escape detection, but again, probably escape ever being run as well.

There is a lot in /opt/backup, which doesn't help unless someone restores from backup. We might be able to cause a restore from backup by being aggressive enough with fork-bombs and filling up the disk, but it's risky and probably all the writable files in the backup are also writable on the main system already.

/var/www are all writable by Apache, so if I can hack the Apache user then I can mess with a fair bit of stuff. I don't know how to hack the Apache user, though. Oh, hmm - if there's a world-writable directory in there, I can make a CGI script and Apache might run it! After sorting out directories that require more permissions to access from the Web than from shell, such as /var/www/cs/htdocs/courses/spring2010/internet/protected/head_first/book_code/chapter_01/starbuzz, we're still in luck! /var/www/cs/htdocs/courses/fall2010/systems/bomblab is world-writable and anyone can go to `http://cs.marlboro.edu/courses/fall2010/systems/bomblab` . I tried creating a test.cgi there and accessing it; it resulted in the program text, rather than running the program. I tried making a .htaccess file in bomblab/ with `Options +ExecCGI` and (within that subdirectory) then got Apache errors – progress! I changed it to an empty .htaccess file and the errors went away. Now to find out what we can do with that file... hmm, none of the actual content I've tried has not created an Apache error there. Oh well. Putting them inside a directory named cgi_bin didn't help either. Making a .php file didn't work either. Maybe this won't work. A pity. All we can do is deface small parts of CS and then crash the server, and we needed to do lots of work to get here (unless we're already one of the CS students).

But with physical access – and the room is often left unlocked carelessly – one can boot CS from a USB stick, put in a rootkit, and steal all the keypresses that ever go through it.

Or we could bug the keyboard in Sci 217 to steal some passwords.

I wonder if we can spoof its MAC address and get some of the packets aimed for it to go to our machine instead. Or poison marlboro.edu's DNS - it appears to be run on-site. We wouldn't have the SSL certificate, though, so we'd only be able to phish people on HTTP and modify the HTML so that they log in unencrypted, and hope their browser doesn't warn or they don't care. Which is doable. I'm not sure how practical this MAC or IP spoof is.

I'm sure I've left something out – I'm not someone who goes around hacking things daily, and I don't even know the open-source penetration-testing tools like Metasploit well enough to try some types of attacks that probably exist.

Once we pwn the machine, we can turn it into an open DNS resolver just for the sake of making the Internet slightly worse.

I took another look at WikiAcademia HTTP headers. I'm logged in, and my login cookie is being sent in clear-text (perhaps I followed an http link to a CS page). This means that my physical neighbors can sniff my laptop WiFi, steal the cookie, and edit WikiAcademia as me until the cookie goes invalid. My neighbors might be malicious themselves or have hacked laptops. It's also likely that Jim's cookie can be stolen in this way without anyone noticing – any device in the Science Building could probably do it, and even a strange device in Sci 217 wouldn't attract notice. Also, anyone who can edit CS can probably get Jim to click a link on CS to a page they edited to have cookie-stealing JavaScript on it.

I don't know what abilities Jim has as WikiAcademia admin. His cookie could probably be used to delete most of the data and/or put up spam. Perhaps it could be used to phish users' passwords – not that any of the other users are as valuable a target as Jim is.

It appears that Ubuntu unattended-upgrades is installed but not fully configured. Documentation: https://help.ubuntu.com/community/AutomaticSecurityUpdates (section “Using the “unattended-upgrades” package”). For example, there is an Ubuntu Security Notice from two days ago [from when I wrote this sentence] regarding libtiff[173] and libtiff4 on CS is not the updated version. If an attacker can convince Apache to do something with a TIFF image, then they might be able to run arbitrary code as the ’apache’ user. Ubuntu Security Notices and updates come out often; Ubuntu is fairly prompt about security updates, among distributions, but it helps if the server receives those upgrades promptly.

http://cs.marlboro.edu/server-status is accessible to anyone on campus, and lets you spy on people somewhat, and learn some interesting paths on the server. Is preventing this worth the nuisance of doing so? Is this useful for attacking the server, or only for spying on people?

In httpd.conf, unrelated to security, this is out of date:

```
SetEnvIf User-Agent ".*MSIE.*" \
          nokeepalive ssl-unclean-shutdown \
          downgrade-1.0 force-response-1.0
```

It is for the benefit of IE <= 5, and slows things down for newer IE. It’s possible that newer IE still require ssl-unclean-shutdown (the Web disagrees with itself on this[174]), but the other params should definitely be deleted, if not the whole thing.

What sort of attackers will CS get? Well, it’s not a target like a bank, a politically charged site, or a famous site. But an attacker can insert links to sketchy businesses to increase those businesses’ Google ranking. CS is relatively likely to be targeted for this because (1) in 2007 lots of people believed that .edu sites were a better way to get search rank because they’re limited to educational institutions (2) it has a wiki, so it probably looks editable. Lots of

[173] http://www.ubuntu.com/usn/usn-1631-1/

[174] https://blogs.msdn.com/b/ieinternals/archive/2011/03/27/10146257.aspx?Redirected=true
http://newestindustry.org/2007/06/06/dear-apache-software-foundation-fix-the-msie-ssl-keepalive-settings/

university web pages have link spam subtly hacked into them (I forget citations but it should be easy to find some related articles by web search). It might also be used generically as part of a botnet for DDOSing, email spamming, or credential stealing of anyone who uses it. Non-targeted attacks against generic Linux servers are common enough that a vulnerable server connected to the IPv4 internet is somewhat likely to be compromised already. (CS might plausibly already be compromised and me not know. I found no signs, but I'm unfamiliar with its configuration and don't have root and wasn't actively looking.)

24.5.i Recommendations

- **(EASY)** 'chmod o-w' the files that are world-writable (besides ones that are supposed to be world-writable, like /tmp and some of /proc). (MEDIUM) Create a cron job to detect such files in the future and notify you – they tend to crop up when people aren't paying attention. When I ran `'find / -perm -o+w -a \! -type l'` it took on the order of an hour and did not impact web-server performance.

- **(EASY?)** Mark cs.marlboro cookies as 'Secure' and 'HttpOnly'. 'Secure' means that they will only be sent over https, so network eavesdroppers can't steal them. 'Secure' also prevents this attack[175]. 'HttpOnly' means that JavaScript can't read cookies to steal them, though the JavaScript session can still make requests that take advantage of those cookies; it is just a mitigation.

- **(EASY)** To make the Secure cookies more user-friendly, always 301-redirect http to https on cs. While you're at it, 301-redirect `http://cs/` to `https://cs.marlboro.edu/` to keep that shortcut from breaking. `http://cs.marlboro.edu/something` should redirect to `https://cs.marlboro.edu/something`, not to `https://cs.marlboro.edu/` .

[175] `https://en.wikipedia.org/wiki/HTTP_cookie#Publishing_false_sub-domain_.E2.80.93_DNS_cache_poisoning`

- **(MEDIUM to HARD)** Put user content into an iframe with a different domain-name origin, for example `https://cs-user-content.marlboro.edu/`. This prevents user content from stealing cookies/logins or (extra benefit) messing up the formatting of the wiki pages. For browsers that support it, the iframe can have the 'seamless' attribute so it looks better.

  ```
  <iframe sandbox=".." srcdoc="...">
  ```

  is a way to have inline sandboxed content, but not enough browsers support it yet.[176] It's probably alright to have cs-user-content.marlboro.edu be non-HTTPS and point to the cs HTTP server. (It would be silly to post a web program on the CS wiki that handles even trivially sensitive data; there are so many other places that are actually designed to serve web pages.) And to make this complete,

- **(EASY)** If students request a place to put their HTML, don't offer space on cs.marlboro, to make XSS less likely (in case their motivations, their laptop, or their code is vulnerable to malice; if they have server-side code, this also prevents vulnerabilities they write from affecting all of CS). Instead offer one of the other cloud servers that we've recently bought into for that specific purpose.

Just doing the EASY things would make it rather harder for a casual or automated attacker (which are the likely types to attack CS) to hurt CS or its users.

[176] http://www.whatwg.org/specs/web-apps/current-work/multipage/the-iframe-element.html#the-iframe-element
http://weblog.bocoup.com/third-party-javascript-development-future/

24.6 resources shared between processes

```
question 6
----------

Discuss how linux handles resources which are shared by processes.
What sorts of resources need attention in this context?  How are
threads and forks related to this question, and what is the
difference between them?
```

Response

In Linux, two natural units of separation are the “process” and the “thread”. Each process has an integer number of threads greater than zero; each thread belongs to exactly one process. Processes have a numeric ID on the system (PID) which is unique while the process is running though may be re-used after it quits.

(If Linux were created today, we might just have 64-bit PIDs and not have to worry about them being reused. If a 1 GHz processor could create one process every cycle, which it can’t, it would take more than 500 years to use up the address space ($2^{64}/10^{9}/86400/365$), which might be good enough. But instead, it’s typical to have 16-bit PIDs. PID reuse leads to race conditions in poorly written programs, which is not really the subject of this question. Even if PIDs weren’t reused during one system boot, stale PID files for daemons could still be sitting on the filesystem.)

Threads do not share an instruction pointer; each one is always at a different moment in running code. Typically, memory is shared between all the threads in one process and not between different processes. This is Linux, so we can change this fact; there are several different ways for a program to share part of its memory with another process (typically for the purpose of efficient zero-copy data transfer).

Processes create other processes using fork and exec. Fork creates a copy of the current process. It interacts poorly with threads (I think the copy doesn't have any of the threads besides the one that called fork, because the alternative is terrible for fork+exec). The copy has conceptually non-shared memory with the original process, but in fact unmodified memory is shared via copy-on-write memory pages. Then, often, the copy calls exec to completely replace itself with a different executable program.

There are many more resources, such as file descriptors, to a Linux process. A file descriptor is a process's reference to a file it has open, referenced by a number that is process-specific. Unless otherwise specified (e.g. by O_CLOEXEC when opening files), a process's file descriptors are inherited by its child processes. This is Linux, so the sharing of each kind of resource can be configured if you use the right Linux syscalls.

24.7 how do you set up linux

```
question 7
----------

a) When you are setting up a linux installation for yourself,
what are the components that you install?

b) Which are the system configuration files you feel are
particularly noteworthy, and what goes in 'em?

c) Similarly, what are the noteworthy config files in your
home directory?
```

Response

24.7.a components

I set up the package manager per distribution good/convenient practices. For example, I just installed Ubuntu 12.10 on a server, and the first thing I did was installed 'aptitude' (the slightly higher-level command line program similar to apt-get). Then I update/upgraded the system, and installed and configured unattended-upgrades so that it would install security updates even when I'm not paying attention to it.

I installed some monitoring tools I like, such as htop (a fancier terminal-based process monitor than top) and tcpdump (an IP traffic monitor). 'screen' and 'ssh' were already installed and configured because it's a VPS and those are useful there. I installed and ran chkrootkit and rkhunter to see if they could detect any rootkits in the virtual machine my Linux is in. Unfortunately, rkhunter output a bunch of warnings which looked likely to be because I'm running inside a virtual machine; I didn't research this. I installed build-essential (Debian's meta-package or package group that installs things like 'gcc' and 'make') because I plan to build some things on this server.

I remotely nmap'ed my server and it didn't have any ports obviously open that I didn't want open, so I didn't add firewalling (I have to decide between using iptables directly, and one of the several wrappers around iptables, and then learn that system enough to create security with it and also not lock out my SSH session. I've learned some of that and will probably add a firewall eventually).

Then I installed and configured openvpn and iptables rules so I could use it as a VPN-based proxy for when I'm traveling and using silly WiFi with port blocking or packet modification (or want to make it so that instead of the people next to me being able to hijack my packets, the VPS company can).

I backed up my server configuration files to another system (my laptop, which I then backup up to my laptop's backups). This is reasonable from a security perspective: My laptop is already permitted to control and observe the server in any way, and the server is not permitted to do the same for my laptop. If there was sensitive customer data or a lot of people relied on my server, there might need to be more complex privilege separation than this.

On even a desktop Linux, I edit /etc/ssh/sshd_config and disable password authentication, because if I want to be able to log in to my system remotely at all, publickey is more secure. Some distros automatically run sshd if ssh is installed, and have password authentication enabled by default!

On desktop Linux, I use filesystem encryption (LUKS), which somewhat protects my data from thieves, and makes it easier to erase the disk reliably (for e.g. giving away). I can also put encrypted backups on servers which the servers can't access. It's a bit trickier to encrypt server filesystems and I haven't needed to yet.

It's good to run ntpd to keep the system clock up to date (or an alternative for systems that are rarely online; or, I believe, nothing for inside OpenVZ/LXC lightweight virtualization containers[177]).

[177] http://www.webhostingtalk.com/showthread.php?t=1074561

24.7.b system configuration

As mentioned, in /etc/ssh/sshd_config, you disable password authentication (in addition to disabling the ssh daemon entirely), unless you are sure that every login has a strong password and/or that you wouldn't be too horrified for the server to be compromised. It's too easy to combine a couple configuration bits here and not notice the risk created.

There are not very many configuration files that have consistently appeared in my seven years of various Linux distros. Each distro often organizes config files differently (Gentoo vs. Debian vs. Ubuntu vs. Arch Linux), uses a different init system (sysvinit, upstart, systemd, and custom systems made by Gentoo and by Arch) which affects many config files, and some advancements have been made over the years. These have not changed for me: /etc/fstab /etc/passwd /etc/group /etc/sudoers In fstab, you configure what filesystems to mount. In passwd and group you configure user accounts (shadow contains their password hashes). sudoers configures who can become root via sudo (if root has a password, su can also let anyone become root). Just use Internet documentation and/or poking around to find what the current configuration mechanism is.

24.7.c user configuration

There are lots of configuration files in my laptop's home directory. .xinitrc or .xsession typically runs when your X11 GUI session starts, depending on whether it's started from a console or a GUI login screen. Some desktop environments then also start ~/.config/autostart/* desktop files too, or other desktop-environment-specific auto-starting settings. For shell configuration, ~/.bashrc or ~/.zshrc (or whichever shell you use) is useful. ~/.ssh/authorized_keys is a list of crypto public keys which can log in as this user. Various GUI programs feel the privilege to create non-dot directories in $HOME, so my $HOME is a subdirectory of the directory I treat as my home directory in order to keep all that garbage out of my way.

24.8 write your own question

```
question 8
----------

Pose an exam question for the next Marlboro student to take
a linux exam like this one, of your own design, and answer it.
```

Response

Hmm let's see... here are some ideas:

- Write a script in a language you are comfortable with (e.g. bash or python) that creates a mirror copy of directory tree, using symlinks or hard links to link non-directory files rather than copy them. Do not just use one of the existing commands that does this (e.g. `lndir`[178]), although you may be inspired by their documentation. Give the program sensible "`--help`" and "`--version`" output. For bonus points, copy the directory permissions, allow the user to choose what type of link to use, and/or other sensible command-line flags. For bonus points, cope with weird characters in filenames and with directory trees that change while you copy them.

- Vim vs. Emacs: fight! Talk about each editor's philosophy and their history. Explain what a text editor is and why it's important.

- Compare and contrast config file formats: XML, JSON, INI, shell, etc.

- In today's world, there are thousands if not millions of Linux installations. If you use Linux on your desktop and use Linux web applications, how do you decide between them when they offer the same services? Do you ever install Linux server applications on your personal computer, such as Apache, Nginx, Dovecot, PostgreSQL, Trac, etc?

[178] http://www.xfree86.org/4.8.0/lndir.1.html

Do you find the user experience acceptable? Why or why not? How could it be better? What are the challenges?

- Compare and contrast the Android and GNU/Linux user-spaces.
- If your grandma wants to use Linux, what might you tell her, and why?
- Linux has been used for everything from supercomputers to washing machines. Discuss one of these limits, or a similar one you prefer. Discuss the challenges of Linux scaling to this circumstance, and the alternatives (if any) it is competing against.

I think I'll say a bit about

> **Vim vs. Emacs: fight! Talk about each editor's philosophy and their history. Explain what a text editor is and why it's important.**

along with

> **Compare and contrast config file formats: XML, JSON, INI, shell, etc.**

.

A Unix philosophy is that configuration is in plain text, not binary formats, so that a human with a text editor can easily read and change any configuration. Text editors can be used from the command line. A terminal is (more or less) a grid of text characters, after all.

/etc/profile is shell. This, more than key-value pairs for environment variables, lets it set the umask (via shell builtin command) and include other files, among other things. However, this makes it harder to retrieve a variable from it. The code to retrieve values from such a config file might be more complicated, and (depending on security context) might risk running arbitrary code. Arch Linux PKGBUILDs (text files describing how the package manager should compile and install a package) are written in shell. The Arch User Repository (AUR) web interface has to parse these files to show summary information in web form (such as package name and dependencies). For security and reproducibility reasons, it does not run these files, but instead hackily parses them to get an approximation of what these values are. It only usually works right.

Shell, and many config file formats, have easy-to-use comments that start with # and end at the end of the line.

JSON is nice for hierarchical structured data, but forbids comments, which is good for JSON as a data transfer format but bad for JSON as a configuration file format. It's useful to make notes to future self or others about what the config means or why it is so.

XML has verbose comments, and is generally verbose. It is complicated enough that it requires a library to parse. But it's very standard, so a few Linux components used it. For example, HAL was created, HAL FDI config files used XML, and some years later (for completely unrelated reasons) HAL was deprecated and removed.

INI is a nice format for data that can be structured into sections but doesn't need arbitrary hierarchy. It came from Windows. On Linux it is used for .desktop files (which describe icon and user-visible name for GUI programs), and for systemd's service description files. It's easy to parse in a program, and easy for humans to edit. There are lots of config formats on Linux that are vaguely similar to this (e.g. xorg.conf), because it's easy to write one and we get by fine without standardizing. (I'm not sure whether standardizing would be useful; in a dream world, it might allow GUI system config editors to be written more easily, or tools like Puppet. There are flamewars on the Internet about this.)

GNOME now uses a binary format for its configuration data. Binary formats do have the advantage that arbitrary access and modification can be fast. But they exclude comments and Unixy editing. I suspect this to be a bad choice but can't prove it. The GNOME GUI-based and command-line tools to access this data store are alright. The only thing I've specifically regretted about it is lack of ability to write comments (to remind myself later why I've done something). The best way I found to search the data is to dump it all in the command-line to text and pipe that to grep. That's alright, though it took me a while to find that mechanism.

So, I didn't say that much about vim vs. emacs, because I don't care much these days. Any editor, command-line or GUI, preferably with line-based cut/paste (vim, emacs, nano, and kate have this!), is similarly usable to me. Vim is conceptually derived from Vi, a common Unix text editor. (Ed is technically the standard unix text editor but is not visually interactive and hardly ever used.) Vi is closer to the Unix philosophy of one tool for each task. Emacs is sometimes described as an operating system of its own! Richard Stallman originally wrote Emacs (with others helping sometimes). Emacs has a major fork XEmacs, due to project management and licensing disputes from more than a decade ago. Emacs tends to use more system resources than Vim, which is probably less relevant than it used to be. Besides, it is possible to configure Vim to use too many resources also. They both have GUI and console versions. They both require learning obscure keyboard shortcuts. Vim's are modal and Emacs's involve lots of modifier keys, neither of which is particularly ergonomic/human-friendly, but it is kind of like every button on your keyboard is a control in an airplane cockpit: potentially powerful and efficient. Nano is a simple, self-documenting console-based editor (a FOSS clone of Pico). There are several GUI-only text editors (not to be confused with word processors) that generally use prevailing GUI paradigms. Most text editors have syntax hilighting, which makes it easier to edit code and config files. (Nano doesn't, busybox vi doesn't, and any editor with hilighting turned off obviously doesn't.)

Since I'm mentioning text editors, I can't omit: In programming, text editors compete with IDEs. IDEs incorporate a text editor and have many more code-related functions than syntax hilighting. (Some of these functions can be configured into Vim or Emacs too.) IDEs are generally GUI programs. Some languages go better with IDEs than others. Java goes well with IDEs because it is too verbose to write by hand and because it is easy to understand programmatically (due to simple static type system and filesystem layout of code). Ruby goes poorly with IDEs because it is possible to change functions and classes at runtime and because it doesn't require as much typing for the same code. Most languages fall in between on this spectrum. It also depends greatly on whether a great IDE has been written for the language(s) you're working in.

Part V

Appendix: Source code of language prototypes

25 StarPlay Read-Me

This is a simulation somewhat like StarLogo (cf. Mitchel Resnick, *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds* [36])

with “turtles” - active agents that move around the world using simple scripts,

and “patches” - a grid of locations that can hold state (and also do things if they want to).

The world wraps on the side and top/bottom (it’s a torus).

The code I’m writing and playing with is in `playground/playground.{html,coffee}`

This runs completely in-browser; presently you program it in either CoffeeScript or the Lisp-syntax-inspired language I have created. There’s an example starting script already visible in the browser that you can click on and edit. There are buttons in the top-right to re-compile and run the script; it will give you an error message if you have a syntax error. (Currently, the line number it gives you is not very usable, and various other things are probably tedious. It is not yet self-documenting.)

Tested in: current Firefox and Chromium.

25.1 CoffeeScript intro for programmers

It compiles to JavaScript and tries to have as straightforward a correspondence between the CoffeeScript and the JavaScript in order to aid debugging and make people more comfortable with using CoffeeScript. Indentation matters. The last line of a function implicitly has a `return` prefixed to it if there isn't one there. `@` means `this`, and `@thing` means `this.thing`. `(a, b) -> stuff` creates a function. Zero-argument functions may have the argument tuple omitted (`-> stuff`). Function-call argument parens, object-construction curly-brackets, etc., may be omitted in some cases. An entire CoffeeScript is wrapped in `(function{...}())`, and `var` is implicit; thus it's impossible to write to global variables except by saying things like `window.foo = "bar"`.

`http://coffeescript.org/`

25.2 Credits

Inspiration from StarLogo and many people.

Libraries

Backbone.js, Underscore.js, jQuery, Raphaël, CoffeeScript

(most of which I don't use currently)

Code started from Raphael demo "graffle"

(though currently has nothing to do with it)

http://raphaeljs.com/graffle.html

25.3 Get it

Currently, its components are live on `http://www.idupree.com/starplay/` and `http://www.idupree.com/lispy/`, and its source code is at `https://github.com/idupree/Starplay`.

26 HTML+JavaScript StarPlay simulation code

Dependencies All the dependencies are freely MIT-licensed and fairly popular. Copies are included in the Git repository (or in the case of jQuery, my HTML points directly to

Google's content delivery network, and in the case of Sass, I have it installed as a Ruby gem on my computer). If I am using an older version of a library than current, hopefully it still works in the newer version, but I list these as known-working versions.

StarPlay requires

- Underscore.js (currently using 1.3.3), homepage `http://underscorejs.org/`
- Backbone.js (currently using 0.9.2), homepage `http://backbonejs.org/`
- Backbone.localStorage (apparently unversioned), homepage `http://documentcloud.github.com/backbone/docs/backbone-localstorage.html`
- jQuery (currently using 1.7.2), homepage `http://jquery.com/`
- CoffeeScript (currently using 1.3.1), homepage `http://coffeescript.org/`
- Sass (to compile .scss to .css; currently using 3.1.7), homepage `http://sass-lang.com/`

It generates words from a word list derived from the `/usr/share/dict/american-english` file on my computer, censored by me in an attempt not to generate swears or offensive or disturbing words. This original file is part of Arch Linux package `words 2.1-1` which claims to be a combination of GPL and permissively licensed and derived from `ftp://ftp.gnu.org/gnu/aspell/dict/0index.html`; the English words lists are covered by permissive licenses which I reproduced in `words-copyright-notices-en.txt` in my Git repository. The word list is not reproduced here because it contains 62691 words, but it is `words-int.json` in my Git repository.

It works in browsers that support `<canvas>`, which is everything except old mobile browsers and Internet Explorer 8 or older.[179]

[179] `http://caniuse.com/#search=canvas`

It also depends on the JavaScript implementation of Lisp-inspired language code listed below and contained in `lispy/` in the same Git repository.

26.1 README

```
To compile:

coffee --compile playground.coffee
sass playground.scss playground.css

( coffee: http://coffeescript.org/ , sass: http://sass-lang.com/ )

Then point your browser at playground.html to run it.

It only requires cookies/localStorage to be enabled because I shamefully
used the existing backbone.localStorage library rather than writing a
backbone.js backend that uses regular JavaScript values for storage.

There is also a prototype in the subdirectory 'lispy-hs' whose code
is completely unrelated to this HTML prototype.
```

26.2 HTML (playground.html)

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title>Starplay</title>
        <link rel="stylesheet" href="playground.css" />
        <script
          src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js"
        ></script>
        <script>
            "use strict";
            window.StarPlay = {};
            //window.StarPlay.wordsAjaxRequest = $.getJSON('words-int.json',
            //              function(data) { window.StarPlay.words = data; });
        </script>
        <script src="underscoreBackbonePlusLocalStorage-min.js"></script>
        <script src="lispy/lispy.js"></script>
        <script src="playground.js"></script>
        <script src="coffee-script.js"></script>
        <!--[if lte IE 8]>
        <script src="//html5shim.googlecode.com/svn/trunk/html5.js"></script>
        <![endif]-->
    </head>
    <body>
        <canvas id="gameCanvas"></canvas>
        <p>Stats: Turn <span id="turn"></span>, Turtles <span id="turtles"></span
                       >; <span id="error"></span></p>
        <div id="buttons">
        <button id="restart">Restart</button>
        <button id="pause_resume">Pause/Resume</button>
        </div>
        <div class="next-to-board">
            <div class="fn-heading">Name ; Rule ; Active?</div>
            <ul id="turtleFns"></ul>
            <button id="testplus">+</button>
        </div>
        <p class="onGitHub">(<a target="_blank"
                               href="https://github.com/idupree/Starplay"
                               >https://github.com/idupree/Starplay</a>)</p>
        <script src="words-int.js"></script>
    </body>
</html>
```

26.3 Stylesheet (playground.scss)

```
// CSS reset
article,aside,figure,footer,header,hgroup,menu,nav,section { display:block; }
body,div,dl,dt,dd,ul,ol,li,h1,h2,h3,h4,h5,h6,pre,code,form,fieldset,
  legend,input,button,textarea,select,p,blockquote,th,td{margin:0;padding:0}
h1,h2,h3,h4,h5,h6 {font-size: 100%;font-weight: inherit;}
img {color: transparent; font-size: 0; border: 0; vertical-align: middle;
     -ms-interpolation-mode: bicubic;}

@mixin box-shadow($val) {
        // firefox 3.6, older safari and chrome
        -moz-box-shadow: $val;
        -webkit-box-shadow: $val;
        box-shadow: $val;
}

$dark-background: #333;
$maroon: #a05;
$grassy: #3a3;
$bushy: #008000;
$red-error: #e55;

body, html {
  margin: 0;
  padding: 0;
  height: 100%;
  width: 100%;
  background: $dark-background;
  color: #fff;
  font: 300 100.1% "Helvetica Neue", Helvetica,
        "Arial Unicode MS", Arial, sans-serif;
}
p {
  text-align: center;
  margin: 5px;
}
canvas {
  float: left;
  margin: 5px 10px;
}
button {
  padding: 0 6px;
}
#error {
  color: red;
}
div.next-to-board {
  margin-left: 620px;
  margin-right: 5px;
  background-color: $maroon;
}
.fn-heading {
  padding: 6px 6px 2px;
  font-weight: bold;
  color: #eee;
```

```
}
#testplus {
  margin: 2px 5px 4px;
}
#turtleFns {
  list-style: none;

  > li {
    text-indent: -8em;
    padding-left: 8em;
    border: 4px solid $maroon;
    background-color: $grassy;

    > * {
      text-indent: 0;
      display: inline-block;
      vertical-align: middle;
    }

    > div.turtle-fn-menu > div {
      display: none;
      background-color: $grassy;
      > a {
        font-weight: bold;
        text-decoration: none;
        color: black;
        line-height: 23px;
        height: 23px;
        display: inline-block;
        vertical-align: bottom;
        > img {
          display: block;
        }
        &:hover, &:focus, &:active {
          @include box-shadow ((0 0 6px 2px white));
        }
      }
    }
    > div.turtle-fn-menu:hover > div,
    > div.turtle-fn-menu.menuOpen > div {
      display: block;
      position: absolute;
    }

    > span {
      border: 2px solid $bushy;
      min-width: 1.5em;
      min-height: 1em;
      padding: 0 1px;
      white-space: pre-wrap;
      background-color: $dark-background;
      margin: 3px;

      /*todo when using a real editor something  padding-left: 6em;
      text-indent: -6em;*/
    }
    &.hasError {
      background-color: $red-error;
    }
    > .error {
      font-weight: bold;
      font-style: italic;
    }
  }
}

p.onGitHub {
  text-align: right;
  font-weight: bold;
  margin-top: 1em;
}
```

26.4 Implementation code (playground.coffee)

```
# Copyright Isaac Dupree, MIT-licensed.

# This file implements a simulation inspired by StarLogo
# customizable in CoffeeScript and/or a custom Lisp-inspired language.

# Utility

tau = 6.283185307179586476925286766559005768394338798750211
modulo = (num, mod) ->
```

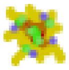
```
  result = num % mod
  result += mod if result < 0
  result

window.console ?= {}
window.console.log ?= ->

rand = {
  # Returns an integer in [min, max)
  intInHalfOpenRange: (min, max) ->
    Math.floor(Math.random() * (max - min)) + min

  # Returns an integer in [min, max]
  intInClosedRange: (min, max) ->
    Math.floor(Math.random() * (max+1 - min)) + min

  arrayIndex: (arr) ->
    rand.intInHalfOpenRange(0, arr.length)

  arrayMember: (arr) ->
    arr[rand.intInHalfOpenRange(0, arr.length)]

  # average case O(1) for mostly-okay arrays
  # (worst case O(n) if really unlucky),
  # up to likely O(n) for mostly-not-okay arrays.
  # Returns null if no item meets the predicate. (Or should it throw?)
  okayArrayMember: (arr, predicate) ->
    if arr.length == 0 then return null

    member = rand.arrayMember arr
    if predicate member then return member

    randomTries = Math.floor(arr.length / 10)
    for _ignored in [0..randomTries]
      member = rand.arrayMember arr
      if predicate member then return member

    okayArr = _.filter arr, predicate
    if okayArr.length != 0
    then return rand.arrayMember okayArr
    else return null
  #i dislike infinite loops, so don't do this:
```

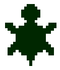
```
  #  loop
  #    member = randArrayMember arr
  #    if predicate member then return member
  }


# Simulation impl

sim = {}

coffeeenv = sim: sim, tau: tau, modulo: modulo
lispyenv = tau: lispy.wrapJSVal tau

#hack debug help
window.StarPlay.sim = sim
sim.fn = {}
sim.fn.turtle =
  clone: (mods = {}) ->
    baby = sim.newTurtle(@, mods)
    sim.turtles.push baby
    baby
  die: ->
    # TODO use a data structure for turtles that has O(1) delete
    sim.turtles = _.without(sim.turtles, @)
    null
  #...maybe have turtle-sets like jquery-sets ?
  forward: (dist = 1) ->
    @x = modulo (@x + dist * Math.cos @heading), sim.patches.width
    @y = modulo (@y + dist * Math.sin @heading), sim.patches.height
    @
  rotateLeft: (amount = tau / 4) ->
    @heading = modulo (@heading + amount), tau
    @
  rotateRight: (amount = tau / 4) ->
    @heading = modulo (@heading - amount), tau
    @
  patchHere: ->
    sim.patches[Math.floor(@x)][Math.floor(@y)]


sim.fn.patch = {}
sim.fn.world =
  # transferAmountFn(patch1, patch2) must return the numerical amount
  # of propName transferred from patch1 to patch2 in a turn;
  # actually fn(patch1, patch2) - fn(patch2, patch1) is transferred.
  diffuse4: (propName, transferAmountFn) ->
    sim.patches.each (patch, x, y) ->
      patch['delta:'+propName] = 0
      if not _.isFinite patch[propName] then patch.grass = 0
    sim.patches.each (patch1, x, y) ->
      for patch2 in [ sim.patches[modulo x+1, sim.patches.width ][y] ,
                      sim.patches[x][modulo y+1, sim.patches.height] ]
        deltaHere = transferAmountFn(patch2, patch1) - transferAmountFn(patch1, patch2)
        patch1['delta:'+propName] += deltaHere
        patch2['delta:'+propName] -= deltaHere
    sim.patches.each (patch, x, y) ->
      patch[propName] += patch['delta:'+propName]

sim.setAllPatches = (fn, width = 20, height = 20) ->
  sim.patches = ( ( sim.newPatch(fn(x,y), {x: x, y: y}) \
                  for y in [0...height]) for x in [0...width])
  sim.patches.width = width
  sim.patches.height = height
  sim.patches.each = (callback) ->
    for col, x in @
      for patch, y in col
        callback(patch, x, y)

sim.newTurtle = (->
  Turtle = ->
  Turtle.prototype = sim.fn.turtle
  return (attrs...) -> _.extend(new Turtle(), attrs...)
  )()
sim.newPatch = (->
  Patch = ->
  Patch.prototype = sim.fn.patch
  return (attrs...) -> _.extend(new Patch(), attrs...)
  )()


simATurn = (sim, isInit = false) ->
  onDynamicUserCodeError = (error, type, fnName) ->
    console.log error.message #?
    #TODO: fix more-UI-related model code in the simulation:
```

```
  #TODO: and fix the O(n) in fn.turtle.length behavior:
  thisPageTurtleFnList.where(type: type, name: fnName)[0].set error: error
  #TODO: remove this error message after a while if the error hasn't
  #happened for a while. Or something to make sure that if the error
  #needed to be fixed somewhere else, and you did, this message here
  #doesn't bug you indefinitely.
sim.time = 0 if isInit #time not turn because turn sounds like rotation
sim.time += 1 if not isInit
#Do world first because for initing that makes sense.
for own fnName, fn of sim.fn.world
  condition = fn.activation
  if condition? and (not fn.isInit == not isInit)
    try
      b = condition.apply(sim.fn.world)
      if !_.isBoolean(b)
        throw "condition did not return 'true' or 'false'"
      if b
        fn.apply(sim.fn.world)
    catch error
      onDynamicUserCodeError error, 'world', fnName
for own fnName, fn of sim.fn.turtle
  condition = fn.activation
  if condition? and (not fn.isInit == not isInit)
    for turtle in sim.turtles
      try
        b = condition.apply(turtle)
        if !_.isBoolean(b)
          throw "condition did not return 'true' or 'false'"
        if b
          fn.apply(turtle)
      catch error
        onDynamicUserCodeError error, 'turtle', fnName
for own fnName, fn of sim.fn.patch
  condition = fn.activation
  if condition? and (not fn.isInit == not isInit)
    sim.patches.each (patch) ->
      try
        b = condition.apply(patch)
        if !_.isBoolean(b)
          throw "condition did not return 'true' or 'false'"
        if b
          fn.apply(patch)
        catch error
          onDynamicUserCodeError error, 'patch', fnName
  return

#The rest of the code is UI stuff

guiState =
  canvas: null
  isRunning: false
  runningTimer: null

renderToCanvas = ->
  canvas = guiState.canvas
  ctx = canvas.getContext('2d')
  width = canvas.width
  height = canvas.height
  canvas.width = canvas.width # clear the canvas

  ctx.save()
#  ctx.scale(canvas.width / grid.width, canvas.height / grid.height)
  ctx.scale(canvas.width / sim.patches.width, canvas.height / sim.patches.height)

  sim.patches.each (patch, x, y) ->
    try
      ctx.fillStyle = _.result patch, 'color'
    catch error
      return #TODO report the error to user
    ctx.fillRect(x, y, 0.95, 0.95)

  for turtle in sim.turtles
    try
      ctx.fillStyle = _.result turtle, 'color'
    catch error
      return #TODO report the error to user
    ctx.save()
    ctx.translate(turtle.x, turtle.y)
    ctx.rotate(turtle.heading)
    ctx.beginPath()
    ctx.moveTo(0.4, 0)
    ctx.lineTo(-0.4, -0.3)
    ctx.lineTo(-0.4, 0.3)
    ctx.fill()
    ctx.restore()

  ctx.restore()


eachTurn = ->
  # outer try in case anything is messed up, like editing the init
  # script wrong making there be no sim.patches (that was an issue)
  try
    simATurn sim
    renderToCanvas()
  catch error
  $('#turn').text(sim.time)
  $('#turtles').text(sim.turtles.length)


# Env is an object { name: value ... } where the names
# are put into the script's environment (by making them
# be function-arguments).
#
# thisVal defaults to undefined; it specifies the value of 'this'
# in the top-level script environment.
#
# If the script returns a value, this function returns that value.
evalScriptInEnv = (scriptText, env, thisVal) ->
  # To get values, map from keys to make certain it's the same number
  # of items in the same order (even if _.values might do that).
  keys = _.keys env;
  values = _.map keys, (key) -> env[key]
  fn = Function.apply null, keys.concat scriptText
  return fn.apply thisVal, values

#compileValue = (coffeescript, env, thisVal) ->
#  scriptAsCoffeeFunction = ('->\n'+coffeescript).replace(/\n/, '\n ')
#  CoffeeScript.compile(, {bare:true})
coffeeeval = (coffeescript, env, thisVal) ->
  # because top-level doesn't make the last line a 'return' in normal coffee
  try
    readyCoffeeScript = ('return (->\n'+coffeescript).replace(
                         /\n/, '\n ') + '\n).call(this)'
    js = '"use strict";' + CoffeeScript.compile readyCoffeeScript, bare: true
  catch error
```

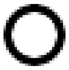
```
    # Adjust for the extra line I have to put at the beginning of the script.
    # Also, if it's a one-liner, don't bother with a line number at all.
    error.message = error.message.replace /\ on line ([0-9]+)/, (_all, line) ->
      if /\n/.test coffeescript then ' on line '+(line - 1) else ""
    throw error
  return evalScriptInEnv js, env, thisVal

# It might be nice to allow blocks with arguments too somehow,
# without requiring (fn () ...) on everything, somehow, hm
lispyeval = (lispyscript, env) ->
  parsed = lispy.parseProgram lispyscript
  return ->
    lispyvals = lispy.evaluate parsed, lispy.mkTopLevelEnv(_.extend({
      '@': lispy.wrapJSVal (tree, env) =>
        member = this[tree[1].string]
        if _.isFunction member
          result = member.apply(this, _.map(tree.slice(2),
                                            (v)->lispy.evaluate(v).value))
                   #but jsval vs. regular val!!! Which expects which?
        else
          if tree.length > 2
            throw "arguments given to a non-function member"
          result = member
        if _.isObject result
          # hack to avoid those self-returning methods causing trouble TODO
          return lispy.mkvoid()
        else
          return lispy.wrapJSVal result
        # aha the ??? is because this '@' fn value is created every time,
        # per obj.
      }, lispy.builtinsAsLispyThings, env))
    #console.log 'heh', lispyvals, lispy.crappyRender(lispyvals)
    if lispyvals.length > 0
      lispyval = lispyvals[lispyvals.length - 1]
      #console.log 'heh2', lispyval, lispy.crappyRender(lispyval)
      if _.has lispyval, 'value'
        return lispyval.value
    return null

userScriptEval = (script, thisVal) ->
  if script[0] == '('
    return lispyeval script, lispyenv
  else
    return coffeeeval script, coffeeenv, thisVal

#TODO use http://ace.ajax.org/ for code editor/syntax hilight etc.

startRunning = ->
  guiState.isRunning = true
  if not guiState.runningTimer
    go = ->
      eachTurn()
      guiState.runningTimer = setTimeout(go, 250)
    go()

stopRunning = ->
  guiState.isRunning = false
  if guiState.runningTimer
    clearTimeout guiState.runningTimer
    guiState.runningTimer = null


generateWordNotIns = (notInObjs) ->
  rand.okayArrayMember window.StarPlay.words,
      (word) -> _.all notInObjs, (obj) -> not _.has obj, word

# This has a name (identifier-style(?) string),
# implementation (CoffeeScript text evaluating to a value,
#                    possibly of function type),
# and activation (CoffeeScript text evaluating to a
#                    function returning boolean, or nothing)
class TurtleFn extends Backbone.Model
  setIfNot: (props, setOptions) ->
    for key, val of props
      if not @get(key)?
        obj = {}
        obj[key] = val
        @set obj, setOptions
  setIfNotF: (props, setOptions) ->
    for key, valf of props
      if not @get(key)?
        obj = {}
        obj[key] = valf()
        @set obj, setOptions
  initialize: ->
    @setIfNotF
      type: -> 'turtle'
      isInit: -> false
      name: -> generateWordNotIns [sim.fn.turtle, sim.fn.patch, sim.fn.world]
      implementation: -> '-> '
      activation: -> '-> true'
      error: -> null
    @on('change:type change:isInit change:name change:implementation change:activation',
        @updateSimCode, @)
    @updateSimCode()
  updateSimCode: ->
    delete sim.fn[@previous 'type'][@previous 'name']
    try
      fn = sim.fn[@get 'type'][@get 'name'] = userScriptEval @get('implementation')
      fn.type = @get 'type'
      fn.isInit = @get 'isInit'
      fn.activation = userScriptEval @get('activation') if @get('activation')?
      @set 'error': null
    catch error
      @set 'error': error.message
      console.log @get('name'), error, error.message, error.stack

class TurtleFnList extends Backbone.Collection
  model: TurtleFn
  localStorage: new Store('StarPlay-TurtleFnList')


class TurtleFnView extends Backbone.View
  #tagName: 'li'
  #className: 'turtle-fn-view'
  make: -> @$domTemplate.clone()[0]
  $domTemplate: $ """
    <li
      ><div class="turtle-fn-menu"
        ><img alt="turtle" tabindex="0" class="turtle-fn-type"
              src="turtle23x23.png" width="23" height="23"
        /><div
          ><a href="javascript:;" class="fn-become-turtle" title="turtle rule"
            ><img alt="be turtle" src="turtle23x23.png" width="23" height="23" /></a
          ><a href="javascript:;" class="fn-become-patch" title="patch rule"
            ><img alt="be patch" src="patch23x23.png" width="23" height="23" /></a
          ><a href="javascript:;" class="fn-become-world" title="world rule"
            ><img alt="be world" src="world23x23.png" width="23" height="23" /></a
          ><a href="javascript:;" class="fn-become-init" title="initialization rule"
            ><img alt="be init" src="init23x23.png" width="23" height="23" /></a
          ><a href="javascript:;" class="fn-delete"
            >Delete</a
        ></div
      ></div
      ><span class="turtle-fn-name" contentEditable="true"></span
      ><span class="turtle-fn-implementation" contentEditable="true"></span
      ><span class="turtle-fn-activation" contentEditable="true"></span
      ><output class="error"></output
    ></li>
    """
  events:
    'focus div.turtle-fn-menu *': -> @$('.turtle-fn-menu').addClass 'menuOpen'
    'blur div.turtle-fn-menu *': -> @$('.turtle-fn-menu').removeClass 'menuOpen'
    #'click .turtle-fn-delete': 'remove' #??maybe? perhaps deleting the name
    #  (or impl?) & it asks if you want to delete.
    'blur .turtle-fn-name': 'rename'
    'blur .turtle-fn-implementation': 'recompile'
    'blur .turtle-fn-activation': 'reactivate'
    'click .fn-become-turtle': -> @model.set type: 'turtle', isInit: false
    'click .fn-become-patch': -> @model.set type: 'patch', isInit: false
    'click .fn-become-world': -> @model.set type: 'world', isInit: false
    'click .fn-become-init': -> @model.set type: 'world', isInit: true
  initialize: ->
    @render()
    @model.on 'change:type', @renderType, @
    @model.on 'change:error', @renderError, @
    #@model.on 'change', @recompile, @
    #@model.on 'destroy',
  #no consistency/compilability checking yet
  #red background? ability to reset to previous? undoes?
  #possibly check that it compiles? also doesn't throw exceptions??
  #lint? in real time while typing?
  rename: -> @model.set 'name', @$('.turtle-fn-name').text()
  recompile: -> @model.set 'implementation', @$('.turtle-fn-implementation').text()
  reactivate: -> @model.set 'activation', @$('.turtle-fn-activation').text()
  renderType: ->
      type = if @model.get 'isInit' then 'init' else @model.get 'type'
      @$('.turtle-fn-type').attr('alt': type, 'src': type+'23x23.png')
  renderError: ->
      @$(’.error’).text (@model.get(’error’) || ’’)
      @$el.toggleClass ’hasError’, (@model.get ’error’)?
  render: ->
    @$(’.turtle-fn-name’).text @model.get ’name’
    @$(’.turtle-fn-implementation’).text @model.get ’implementation’
    @$(’.turtle-fn-activation’).text @model.get ’activation’
    @renderError()
    @renderType()
    @

  #later worry about codemirror

# cf http://www.chris-granger.com/2012/02/26/connecting-to-your-creation/
# which i saw a few days after starting this project

# name text turns red while it’s the same as another? and has a popup or?’

thisPageTurtleFnList = new TurtleFnList

runInitScript = ->
  sim.turtles = []
  delete sim.patches
  try
    simATurn sim, true
    return true
  catch error
    #TODO put error somewhere
    return false

$ ->
  canvas = guiState.canvas = $(’#gameCanvas’)[0]
  canvas.width = 600
  canvas.height = 600
  $(’#restart’).click ->
    if runInitScript() then startRunning() else stopRunning()
  $(’#pause_resume’).click ->
    if guiState.isRunning then stopRunning() else startRunning()
  thisPageTurtleFnList.on ’add’, (model) ->
    $(’#turtleFns’).append new TurtleFnView(model: model).el

  newFn = (type, name, implementation, activation, isInit = false) ->
    thisPageTurtleFnList.create
      type: type
      name: name
      implementation: implementation
      activation: activation
      isInit: isInit
  newFn 'turtle', 'speed', '(@ forward (- 5 4))', '(= (@ type) "bullet")'
  newFn 'turtle', 'activateGun', "-> @clone type: 'bullet', color: 'red'",
                                 "-> @type == 'crazy' and sim.time % 8 == 0"
  newFn 'turtle', 'wobble', """
    ->
      @rotateLeft tau / 16 * (Math.random() - 0.5)
      @forward 0.25""",
    """-> @type == 'crazy'"""
  #newFn 'turtle', 'patchHere',
  #         "-> sim.patches[Math.floor(@x)][Math.floor(@y)]", '-> false'
  newFn 'turtle', 'layGrass', "-> @patchHere().grass += 5", "-> true"

  newFn 'patch', 'decayGrass', "-> @grass *= 0.99", "-> true"
  newFn 'world', 'diffuseGrass', """
    -> @diffuse4 'grass', (patch1) -> patch1.grass / 10 / (4+1)
    """, "-> true"
  newFn 'world', 'setup', """
    ->
      sim.turtles = [sim.newTurtle(
        {x:5, y:5, color:'rgb(88,88,88)', heading:0, type:'crazy'})]
      patchcolor = ->
        'rgb(127,'+(Math.floor Math.min 30*@grass, 255)+',127)'
      sim.setAllPatches (x,y) ->
        {color: patchcolor, grass:0}
    """, '-> true', true

  #window.StarPlay.wordsAjaxRequest.done -> $('#testplus').click(
  #                                           -> thisPageTurtleFnList.create())
  #window.StarPlay.wordsAjaxRequest.fail -> $('#testplus').hide()
  $('#testplus').click(-> thisPageTurtleFnList.create())

  runInitScript()
  startRunning()
```

26.5 Image resources


denotes user initialization code. (init23x23.png)


denotes user turtle-program code. (turtle23x23.png)

denotes user patch-program code. (patch23x23.png)


denotes user world/global code. (world23x23.png)

27 JavaScript Lisp-like language implementation

Dependencies The JavaScript depends on underscore.js (`http://underscorejs.org/`), a small, popular MIT-licensed JavaScript library. My repository online includes a copy of Underscore.js 1.4.2 locally; I do not reproduce it here because it is not my work. The HTML wrapper includes underscore.js before this javascript in order to provide underscore.js to my code.

27.1 HTML-based playground for the language (lispy.html)

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title>Starplay</title>
        <link rel="stylesheet" href="playground.css" />
        <script src="underscore-min.js"></script>
        <script
          src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js"
        ></script>
        <script src="lispy.js"></script>
        <script src="lispy-tests.js"></script>
        <script>
          $(function() {
            "use strict";
            var $code = $(’#code’);
            var $out = $('#out');
            var $testresults = $('#testresults');
            $code.on('change keyup', function() {
              var code = $code.val();
              var out;
              try {
                out = lispy.rep(code);
              }
              catch(e) {
                out = (""+e);
              }
              $out.text(out);
            });
            $code.val(
"((fn (x) x) 3)\n(+ 3 4)\n45\n\n((fn (x)\n  (if (= (mod x 2) 0)\n      (/ x 2)\n      (+
            );
            $code.trigger('change');
            try { lispy.test(); }
            catch(e) { $testresults.text(""+e); }
          });
        </script>
        <!--[if lte IE 8]>
        <script src="//html5shim.googlecode.com/svn/trunk/html5.js"></script>
        <![endif]-->
    </head>
    <body>
      <p>hi</p>
      <textarea id="code" cols="80" rows="10"></textarea>
      <pre id="out">??</pre>
      <pre id="testresults"></pre>
    </body>
</html>
```

27.2 Language implementation (lispy.js)

```
// Copyright Isaac Dupree, MIT-licensed.

// This file implements a simple lisp-like language
// with symbolic evaluation.

(function() {
```

```
"use strict";

var lispy = (window || exports).lispy = {};

var assert = function(b, str) {
  if(!b) {
    throw ("assert failure!  " + (""+str));
  }
};

// TODO testsuite
// TODO consistent } else { style

// Tokenizing and parsing create abstract-syntax-tree objects
// (S-expressions, or sexps) which are the main internal representation
// of the program.
//
// There is no bytecode or intermediate language.
// This is partly intentional to make it simple to retain
// source-level info (e.g. line, column, source text) while computing,
// to make it easier to show the user what's been going on with their code.
var types = {  //strs easier for debugging, objs maybe faster
  // token types
  openParen: "openParen",//{},   // (
  closeParen: "closeParen",//{}, // )
  number: "number",//{},         // 123
  identifier: "identifier",//{}, // abc
  boolean: "boolean",//{},       // true
  comment: "comment",//{}, //TODO
  string: "string",//{},         // "Hi there."
  // (In JS, void is a keyword,
  // so we use _void for simplicity)
  _void: "_void",//{},           // lack of result, e.g. from ((fn () ))
  EOF: "EOF",//{}                // end-of-file (used for parsing)

  // composite types
  // code
  list: "list",//{},             // S-expressions, e.g. (f a b), (f (a b) c),
                                 // (arg1 arg2) in a fn, etc.
  program: "program",//{}        // 1
                                 // 23
                                 // 456
                                 // (the sequence of S-expressions that
                                 //  make up a file.)
  imperative: "imperative",//{}  // Function bodies, which are a sequence
                                 // of S-expressions, become this
                                 // during evaluation.

  // data
  // (I am not using/exposing lisp's homeomorphic abilities presently.)
  // (There is no way to write a literal array or dict presently;
  //  create them with builtin functions (array ...) or (TODO) (dict ...).
  array: "array",//{},           // A runtime sequence
  dict: "dict",//{}, //SOMEWHAT TODO: a runtime assoc

  // other types
  unboundVariable: "unboundVariable",//{}
                                 // When an unbound variable is evaluated,
                                 // it becomes this.
  builtinFunction: "builtinFunction"//{},
                                 // Builtin functions (i.e. ones that simply
                                 // contain a JS function(){}) are this.
};

function isLiteralValueToken(tok) {
  return tok.type === types.number ||
    tok.type === types.identifier ||
    tok.type === types.string ||
    tok.type === types.boolean;
}

// Data in lispy-land are represented by a JS object
// with { 'type': types.something } and other fields
// depending on what the type is.  These functions
// provide ways to create basic lispy-land data:
// mk*() for specific types, and wrapJSVal() for
// any type that occurs both in JS values and lispy-land.
var mkvoid = lispy.mkvoid = function() {
  return { type: types._void, string: "void" };
};
function mknum(n) {
  return { type: types.number, value: n, string: (""+n) };
}
function mkbool(b) {
  if(b) {
    return { type: types.boolean, value: true, string: "true" };
  } else {
    return { type: types.boolean, value: false, string: "false" };
  }
}
function mkidentifier(s) {
  return { type: types.identifier, string: s };
}
var mkUnboundVariable = lispy.mkUnboundVariable = function() {
  return { type: types.unboundVariable, string: ’#unbound-variable’ };
}
function mkfn(f) {
  return { type: types.builtinFunction, value: f,
           string: ("(#builtin-javascript-"+f+")") };
}
function mknamedfn(name, f) {
  return { type: types.builtinFunction, value: f, string: ("#builtin:"+name) };
}
function mkstr(s) {
  return { type: types.string, value: s,
           string: s.replace(/\\/g, ’\\\\’).replace(/"/g, ’\\"’) };
}
// TODO: how to detect WHETHER an ’object’ is a lispy object:
// i should put a field in.
// *should* arrays have a string rep? PROBABLY NOT
function mkarray(a) {
  return { type: types.array, value: a };
}
function mkdict(d) {
  return { type: types.dict, value: d };
}
lispy.wrapJSVal = function(v) {
  if(_.isNumber(v)) {
    return mknum(v);
  }
  else if(_.isBoolean(v)) {
    return mkbool(v);
  }
  else if(_.isFunction(v)) {
    return mkfn(v);
  }
  else if(_.isString(v)) {
    return mkstr(v);
  }
  else {
    throw ’wrapJSVal: not implemented yet: ’ + v;
  }
};
lispy.wrapNamedJSVal = function(name, v) {
  if(_.isFunction(v)) {
    return mknamedfn(name, v);
  }
  else {
    return lispy.wrapJSVal(v);
  }
};


// == Environments ==
// These represent all variables in scope at a particular point.
// An environment is a JS object with a ’lookup’ method that takes
// a varname (JS string) and returns the corresponding sexp if that
// varname is in the environment, or undefined otherwise.
//
// Sexps returned by lookup must not contain free variables.
// TODO: Must they be evaluated to some normal form?
//
// It is dubious but possible to make an environment that
// changes over time.

lispy.mkEmptyEnv = function() {
  return {
    lookup: function(varname) { return undefined; }
  };
};
lispy.mkTopLevelEnv = function(bindings) {
  return {
    bindings: bindings,
    lookup: function(varname) {
      if(_.has(this.bindings, varname)) {
        return this.bindings[varname];
      } else {
        return undefined;
      }
    }
  };
};
// This is experimental/untested and may not be suitable;
// it could be used to allow accesses to undefined variables
// to automatically define them at top level with some default value.
lispy.mkTopLevelEnvWithDefault =
function(bindings, methodToCallIfVarnameNotFound) {
  return {
    bindings: bindings,
    methodToCallIfVarnameNotFound: methodToCallIfVarnameNotFound,
    lookup: function(varname) {
      if(_.has(this.bindings, varname)) {
        return this.bindings[varname];
      } else {
        return this.methodToCallIfVarnameNotFound(varname);
      }
    }
  };
};
// parentEnv: an environment
// bindings: an object mapping varname to sexp
lispy.mkSubEnv = function(parentEnv, bindings) {
  return {
    bindings: bindings,
    lookup: function(varname) {
      if(_.has(this.bindings, varname)) {
        return this.bindings[varname];
      } else {
        return parentEnv.lookup(varname);
      }
    }
  };
};
// parentEnv: an environment
// unbindingsList: a list of variables to unbind
lispy.mkSubEnvOmitting = function(parentEnv, unbindingsList) {
  var unbindings = {};
  _.each(unbindingsList, function(v) {
    unbindings[v] = undefined;
  });
  return lispy.mkSubEnv(parentEnv, unbindings);
};

// == Builtin functions ==
// Consider a JS var 'sexp' representing '(+ 2 3)';
// sexp.type will be types.list.  '+' is sexp[0].
// If the '+' in scope is
//   {type: types.builtinFunction, value: f}
// then the eval loop will call 'f(sexp, env)' where
// env represents the variables in scope (see Environments).
//
// Builtin functions cannot count on their arguments being
// evaluated, and must do so explicitly if they wish to e.g. do math
// on their arguments.  As a benefit, it is simple to write 'if' as a
// builtin function that does not evaluate both the true and false
// branches.

// These are convenience functions for use writing builtins.
function evaluateToNumber(sexp, env) {
  var evaled = lispy.evaluate(sexp, env);
  assert(evaled.type === types.number,
         lispy.printSexp(sexp) + " is not a number");
  return evaled.value;
}
function evaluateToBool(sexp, env) {
  var evaled = lispy.evaluate(sexp, env);
  assert(evaled.type === types.boolean,
         lispy.printSexp(sexp) + " is not a boolean");
  return evaled.value;
}
// The required sexp length will be one more than
// the number of arguments (the function name is also
// part of the sexp length).
function arityAssert(sexp, requiredSexpLengthCount) {
  assert(sexp.length === requiredSexpLengthCount,
         lispy.printSexp(sexp) + " arg count");
}
function modulo(num, mod) {
  assert(mod > 0, "modulo: non-positive divisor " + mod);
  var result = num % mod;
  if(result < 0) {
    result += mod;
  }
  return result;
}

// These are the typical set of builtins.
// You can make them available to lispy programs by passing
// builtinsEnv as the env parameter to the lispy code you evaluate.
var builtins = {
  // We just offer binary ops currently, not the lisp pattern of
  // monoidal ops like ’+’, ’*’, ’or’, ’and’ accepting any number of
  // arguments.

  // TODO: don’t rely on floating-point math.
  // (This TODO will not be done in the prototype implementation.)
  ’+’: function(sexp, env) {
    arityAssert(sexp, 3);
    return mknum(evaluateToNumber(sexp[1], env) + evaluateToNumber(sexp[2], env));
  },
  ’-’: function(sexp, env) {
    arityAssert(sexp, 3);
    return mknum(evaluateToNumber(sexp[1], env) - evaluateToNumber(sexp[2], env));
  },
  ’*’: function(sexp, env) {
    arityAssert(sexp, 3);
    return mknum(evaluateToNumber(sexp[1], env) * evaluateToNumber(sexp[2], env));
  },
  ’/’: function(sexp, env) {
    arityAssert(sexp, 3);
    return mknum(evaluateToNumber(sexp[1], env) / evaluateToNumber(sexp[2], env));
  },
    //console.log(sexp);
    //floating point math?
  ’mod’: function(sexp, env) {
    arityAssert(sexp, 3);
    return mknum(modulo(evaluateToNumber(sexp[1], env),
                        evaluateToNumber(sexp[2], env)));
  },
  ’negate’: function(sexp, env) {
    arityAssert(sexp, 2);
    return mknum(-evaluateToNumber(sexp[1], env));
  },
  //should "and"/"or" use the "return the first/last valid value" thing and
```

```
//have all this implicit boolean convertability?
//These implementations do not evaluate the second argument if the first
//one shows we don't need to know its value (intentionally) (due to the JS
//short circuiting behavior here).
'and': function(sexp, env) {
  arityAssert(sexp, 3);
  return mkbool(evaluateToBool(sexp[1], env) && evaluateToBool(sexp[2], env));
},
'or': function(sexp, env) {
  arityAssert(sexp, 3);
  return mkbool(evaluateToBool(sexp[1], env) || evaluateToBool(sexp[2], env));
},
'not': function(sexp, env) {
  arityAssert(sexp, 2);
  return mkbool(!evaluateToBool(sexp[1], env));
},
//equality/lessthan ?
// THIS IS NOT A VERY GOOD IMPLEMENTATION, TODO
'=': function(sexp, env) {
  arityAssert(sexp, 3);
  var arg1 = lispy.evaluate(sexp[1], env);
  var arg2 = lispy.evaluate(sexp[2], env);
  assert(isLiteralValueToken(arg1),
    lispy.printSexp(arg1)+" must be of atomic type to be compared using =");
  assert(isLiteralValueToken(arg2),
    lispy.printSexp(arg2)+" must be of atomic type to be compared using =");
  return mkbool(arg1.value === arg2.value);
},
'not=': function(sexp, env) {
  arityAssert(sexp, 3);
  var arg1 = lispy.evaluate(sexp[1], env);
  var arg2 = lispy.evaluate(sexp[2], env);
  assert(isLiteralValueToken(arg1),
    lispy.printSexp(arg1)+" must be of atomic type to be compared using not=");
  assert(isLiteralValueToken(arg2),
    lispy.printSexp(arg2)+" must be of atomic type to be compared using not=");
  return mkbool(arg1.value !== arg2.value);
},
// CURRENTLY ONLY ARE A THING FOR NUMBERS:
// TODO doing this deterministically for every type could be interesting.
// First consider what this means for mutable data types. (if there are any.)
'<': function(sexp, env) {
  arityAssert(sexp, 3);
  return mkbool(evaluateToNumber(sexp[1], env) < evaluateToNumber(sexp[2], env));
},
’>’: function(sexp, env) {
  arityAssert(sexp, 3);
  return mkbool(evaluateToNumber(sexp[1], env) > evaluateToNumber(sexp[2], env));
},
’>=’: function(sexp, env) {
  arityAssert(sexp, 3);
  return mkbool(evaluateToNumber(sexp[1], env) >= evaluateToNumber(sexp[2], env));
},
’<=’: function(sexp, env) {
  arityAssert(sexp, 3);
  return mkbool(evaluateToNumber(sexp[1], env) <= evaluateToNumber(sexp[2], env));
},
// IIRC ’if’ needs to be a builtin in strictly evaluated languages
’if’: function(sexp, env) {
  arityAssert(sexp, 4);
  var b = evaluateToBool(sexp[1], env);
  return (b ? lispy.evaluate(sexp[2], env) : lispy.evaluate(sexp[3], env));
},
// these create
//’array’
//’dict’
//(mutable? then they’d need a name and stuff)
’array’: function(sexp, env) {
  var result = [];
  _.each(sexp.slice(1), function(v) {
    result.push(lispy.bindFreeVars(lispy.evaluate(v, env), env));
  });
  return mkarray(result);
}
/*’dict’: function(sexp, env) {
  assert(sexp.length % 2 === 1, lispy.printSexp(sexp) + " arg count is even");
  var result = {};
  var key = null;
  _.each(sexp, function(v) {
    if(key) {
      result[key]//...wait JS only has strings for keys. hmm.
    }
    result.push(lispy.evaluate(v, env));
  });
```

```
    return mkarray(result);
  }*/
};
var builtinsAsLispyThings = {};
_.each(builtins, function(val, key) {
  builtinsAsLispyThings[key] = lispy.wrapNamedJSVal(key, val);
});
lispy.builtins = builtins;
lispy.builtinsAsLispyThings = builtinsAsLispyThings;
var builtinsEnv = lispy.builtinsEnv = lispy.mkTopLevelEnv(builtinsAsLispyThings);

// TODO consider:
//what if all composite types (fn, list, assoc) got names
//let's see
//i'll have to set up indirection for:
//beta-reduce
//any builtins that operate on list, assoc

// Legal characters in lispy identifiers are the same as Scheme allows
// (mostly; even [:alnum:] doesn't work in JS regexps; this code ought to
// allow non-ASCII letters but it's too hard to do in browser JS to justify
// doing it in this language-prototype).
var identifierChar = /[\-!$%&*+.\/:<=>?@\^_~0-9a-zA-Z]/;
// following Haskell, pretend there are tabstops every 8 characters
// for the sake of defining what column a character is at:
var tabwidthForColumnCount = 8;
// following the most common conventions for line and column numbering:
var initialLine = 1;
var initialColumn = 0;

// tokenize : string -> array of token details objects
function tokenize(str) {
  var result = [];
  var pos = 0;
  var line = initialLine;
  var column = initialColumn;
  var token = function(details, len) {
    details.pos = pos;
    details.line = line;
    details.column = column;
    details.endLine = line; // TODO multi line tokens (strings?)
    details.endColumn = column + len - 1;
    details.len = len;
    details.string = str.slice(pos, pos + len);
    result.push(details);
    pos += len;
    // assumes tokens don’t contain \n
    column += len;
  };
  while(pos < str.length) {
    if(/[ \t\n\r]/.test(str[pos])) {
      if(str[pos] === ’ ’) {
        column += 1;
      } else if(str[pos] === ’\t’) {
        column += tabwidthForColumnCount;
        column -= (column % tabwidthForColumnCount);
      } else if(str[pos] === ’\n’) {
        column = initialColumn;
        line += 1;
      }
      // else assume any \r is in a \r\n combination and ignore it.
      pos += 1;
    } else if(str[pos] === ’"’) {
      var strlen = 1;
      assert(pos + strlen < str.length, "unterminated string literal");
      while((str[pos + strlen - 1] === ’\\’ || str[pos + strlen] !== ’"’)) {
        ++strlen;
        assert(pos + strlen < str.length, "unterminated string literal");
      }
      ++strlen;
      var strstr = str.slice(pos + 1, pos + strlen - 1);
      strstr = strstr.replace(/\\"/g, ’"’).replace(/\\\\/g, ’\\’);
      token({type: types.string, value: strstr}, strlen);
    } else if(str[pos] === ’(’) {
      token({type: types.openParen}, 1);
    } else if(str[pos] === ’)’) {
      token({type: types.closeParen}, 1);
    } else if(/^-?\.?[0-9]/.test(str.slice(pos, pos+3))) {
      var numlen = 1;
      while(pos + numlen < str.length &&
            /[\-0-9a-zA-z_,.]/.test(str[pos + numlen])) {
        ++numlen;
      }
      // TODO use our own not JS’s hex/oct/dec number syntax?
```

```
      // TODO int vs non integer numbers?
      //var numval = parseInt(str.slice(pos, pos + numlen));
      var numstr = str.slice(pos, pos + numlen);
      var numval = +numstr;
      assert(!isNaN(numval), "number interpretation of ’" + numstr +
              "’ failed at line " + line + " column " + column + "!");
      // TODO forbid 12&& being a number token followed by an
      // identifier token without any spaces?  Ah by eating up
      // a whole identifier and then if it begins number-like
      // then make it a number or fail.
      token({type: types.number, value: numval}, numlen);
    } else if(identifierChar.test(str[pos])) {
      var idlen = 1;
      while(pos + idlen < str.length && identifierChar.test(str[pos + idlen])) {
        ++idlen;
      }
      // are true/false keywords? why? why not immutable globals?
      // why not #t / #f?
      var idstr = str.slice(pos, pos + idlen);
      if(idstr === ’true’) {
        token({type: types.boolean, value: true}, idlen);
      } else if(idstr === ’false’) {
        token({type: types.boolean, value: false}, idlen);
      } else {
        token({type: types.identifier}, idlen);
      }
    } else {
      throw ("tokenizer fail at line " + line + " column " + column + "!");
    }
  }
  token({type: types.EOF}, 0);
  //console.log(result);
  return result;
}

// Returns { parsed: list of sub-lists or tokens, endPos: n }
// with endPos one-after-end
function parseList(toks, pos, type) {
  var ourNest = [];
  ourNest.type = type;
  while(true) {
    if(toks[pos].type === types.openParen) {
      var result = parseList(toks, pos + 1, types.list);
      ourNest.push(result.parsed);
      pos = result.endPos;
    } else if(isLiteralValueToken(toks[pos])) {
      ourNest.push(toks[pos]);
      pos += 1;
    } else if((toks[pos].type === types.closeParen && type === types.list) ||
              (toks[pos].type === types.EOF && type === types.program)) {
      pos += 1;
      return { parsed: ourNest, endPos: pos };
    } else {
      //TODO maybe different error handling?
      throw ("parse failure at line " + toks[pos].line +
             " column " + toks[pos].column);
    }
  }
}

// parseProgram : string -> sexps
lispy.parseProgram = function(str) {
  return parseList(tokenize(str), 0, types.program).parsed;
};

// parseSexp : string -> sexp
lispy.parseSexp = function(str) {
  var prog = lispy.parseProgram(str);
  assert(prog.length === 1,
         "parseSexp: program containing not exactly one sexp");
  return prog[0];
};

// parseProgram ""
// parseProgram "()"
// parseProgram "(1)"

// TODO next:
//   (fn (a b c) (+ b c))
//  fn and application and +  ...and do/sequencing ?
//  application: eval each argument (incl function) and then enter function.
//    or rather: eval function, look at its strictness

//lispy.renameProgram //well we could use a de Bruijn form, we could use ptrs,
// we could just turn all the strings into symbols/atoms (numbers)


//This should probably (be used when) eval args first,
//not duplicate them like that
//Or name the args (give them wacky names)
//beta-reduce depth first deeper first left-to-right
//ALSO bug beware http://foldoc.org/name+capture
//if args are not in NORMAL FORM with all their non bound vars
//already substituted (locally non bound ones; & globally
//non bound ones as errors or undef)
// http://foldoc.org/Weak+Head+Normal+Form
// http://stackoverflow.com/questions/6872898/haskell-what-is-weak-head-normal-form
// http://en.wikipedia.org/wiki/Beta_normal_form


// copies all non-numeric own keys from orig to dest
function keepMetaDataFrom(orig, dest) {
  for(var key in orig) {
    if(_.has(orig, key) && !/^[0-9]+$/.test(key)) {
      dest[key] = orig[key];
    }
  }
  return dest;
}

//...larger arguments get (randomly)named, perhaps
lispy.betaReduceO_N = function(sexp) {
  // pattern-match ((fn (params...) body...) args...)
  assert(lispy.isHeadBetaReducible(sexp), "beta beta");
  //assert(sexp.type === types.list);
  //assert(sexp.length >= 1);
  var fn = sexp[0];
  var args = sexp.slice(1);
  //assert(fn.length > 2);
  //assert(fn[0].type === types.identifier);
  //assert(fn[0].string === 'fn');
  //assert(fn[1].type === types.list);
  var params = fn[1];
  var body = keepMetaDataFrom(fn, fn.slice(2));
  body.type = types.imperative;
  var substitutions = {};
  assert(params.length === args.length,
    lispy.printSexp(sexp) + " equal params length"); //no silly stuff!
  for(var i = 0; i !== params.length; ++i) {
    assert(params[i].type === types.identifier, "params are identifiers");
    substitutions[params[i].string] = args[i];
  }

  // We need to substitute it everywhere except where it's
  // named in a let or lambda
  // [and except where it's quoted / part of a macro]
  return lispy.substitute(body, lispy.mkTopLevelEnv(substitutions));
};

// evaluates all beta-redexes not within a lambda (?)
// evaluates until we have a int or lambda?
// prevents list literals from being partly evaled?

// TODO make fn be scoped identifier, or
// make the parser recognize it specially.
lispy.isLambdaLiteral = function(sexp) {
  return sexp.type === types.list &&
    sexp.length > 1 &&
    sexp[0].type === types.identifier &&
    sexp[0].string === 'fn' &&
    sexp[1].type === types.list;
};

lispy.isHeadBetaReducible = function(sexp) {
  return sexp.type === types.list &&
    sexp.length >= 1 && lispy.isLambdaLiteral(sexp[0]);
};

lispy.rep = lispy.readEvalPrint = function(str) {
  return lispy.printSexp(lispy.evaluate(lispy.parseProgram(str), builtinsEnv));
};

// lispy.rep("((fn (x) (x x)) (fn (x) (x x)))")
// lispy.printSexp(lispy.evaluate(lispy.parseProgram(
//   "((fn (x y) y (x y)) (fn (x) x) 34)")))
// lispy.printSexp(lispy.evaluate(lispy.parseProgram(
//   "((fn (x y) y) 23 34)")[0]))
```

```
//TODO fix desc: env is an environment (see section Environments).
//It is used to look up free variables.
//(Strictly evaluated substitution e.g. betaReduceO_N
// cannot implement recursion or letrec,
// so env is necessary, not just conventional.
// It can be represented with a (let (...) ...)
// around the to-be-evaluated sexp if necessary.)
//BUG TODO- members of 'env' cannot have any free variables.
//I *think* pre-substituting them is fine but having a (env (...) ...)
//primitive would be fine (like 'let' but clears the scope) (or let for
//all the free variables also works fine) (assuming they *can* be referenced
//thus).
//
//returns the evaluated version of the sexp.
lispy.evaluate = function(sexp, env) {
  if(arguments.length === 1) {
    env = lispy.mkEmptyEnv();
  }
  while(true) {
    var lookedup;
    if(sexp.type === types.identifier &&
       (lookedup = env.lookup(sexp.string))) {
      // substitute plain identifiers from env:
      //   ident
      sexp = lookedup;
    }
    else if(sexp.type === types.list && sexp.length >= 1 &&
        sexp[0].type === types.identifier &&
        (lookedup = env.lookup(sexp[0].string))) {
      // substitute function names that are about to be called from env:
      //   (ident ...)
      var headEvaledSexp = shallowCopyArray(sexp);
      headEvaledSexp[0] = lookedup;
      sexp = headEvaledSexp;
    }
    else if(lispy.isHeadBetaReducible(sexp)) {
      // call lambda:
      //   ((fn (...) ...) ...)
      sexp = lispy.strictBetaReduceO_N(sexp, env);
    }
    else if(sexp.type === types.program) {
```

```
  // evaluate programs in sequence:
  //   (+ 1 2)
  //   (+ 3 4)
  sexp = keepMetaDataFrom(sexp, _.map(sexp, function(subsexp) {
    return lispy.evaluate(subsexp, env);
  }));
  break;
}
else if(sexp.type === types.imperative) {
  // evaluate function-bodies in sequence:
  //   (+ 1 2) (+ 3 4)
  var result = mkvoid();
  _.each(sexp, function(subsexp) {
    result = lispy.evaluate(subsexp, env);
  });
  sexp = result;
  break;
}
else if(sexp.type === types.list &&
    sexp[0].type === types.builtinFunction) {
  // evaluate builtins:
  //   (+ 1 2)
  sexp = sexp[0].value(sexp, env);
  break;
}
else if(sexp.type === types.list &&
    sexp[0].type === types.list) {
  // attempt to evaluate the function part:
  //   ((if true + -) 7 3)
  var headEvaled = lispy.evaluate(sexp[0], env);
  // TODO identity-based equality comparison is
  // fragile here? (to prevent infinite loop
  // trying to reduce something that we can't
  // reduce anymore)
  // TODO if function-calling called evaluate
  // once on its body then there would be no
  // infinite loop/recursion worries, right?
  // (aside from legitimately nonterminating
  // computations, of course).
  if(headEvaled === sexp[0]) {
    break;
  }
```

```
      else {
        var headEvaledSexp = shallowCopyArray(sexp);
        headEvaledSexp[0] = headEvaled;
        sexp = headEvaledSexp;
      }
    }
    else {
      // No reductions found: we must be done.
      break;
    }
  }
  //sexp = lispy.bindFreeVars(sexp, env);
  return sexp;
};

function shallowCopyArray(arr) {
  return keepMetaDataFrom(arr, _.map(arr, _.identity));
}

lispy.strictBetaReduceO_N = function(sexp, env) {
  assert(lispy.isHeadBetaReducible(sexp), "strictBeta beta");
  var argsEvaledSexp = shallowCopyArray(sexp);
  for(var i = 1; i !== sexp.length; ++i) {
    argsEvaledSexp[i] = lispy.evaluate(sexp[i], env);
  }
  return lispy.betaReduceO_N(argsEvaledSexp);
};

/*
  substitute({
    'foo': { type: types.number, value: 3, string: "3", ... },
    'bar': { type: types.identifier, string: "something", ... },
  },
  [ { type: types.identifier, string: "+", ... },
    { type: types.identifier, string: "foo", ... },
    { type: types.number, value: 7, string: "7", ... }
  ])

wait, does it mutate or return a new sexp?
return a new sexp. there might be sharing
of some tokens - we intend that they are never
mutated by any code
```

```
TODO can 'fn' be bound? that is not guarded against.
*/
lispy.substitute = function(sexp, env) {
  var lookuped;
  if(sexp.type === types.list || sexp.type === types.program ||
                                 sexp.type === types.imperative) {
    if(lispy.isLambdaLiteral(sexp)) {
      var bindings = _.pluck(sexp[1], 'string');
      var subEnv = lispy.mkSubEnvOmitting(env, bindings);
      return keepMetaDataFrom(sexp, _.map(sexp, function(sub) {
        return lispy.substitute(sub, subEnv);
      }));
    }
    else {
      return keepMetaDataFrom(sexp, _.map(sexp, function(sub) {
        return lispy.substitute(sub, env);
      }));
    }
  }
  else if(sexp.type === types.identifier &&
          (lookuped = env.lookup(sexp.string))) {
    return lookuped;
  }
  else { //other token
    return sexp;
  }
};

// returns a set { var: true, ... } of the free vars in sexp
// Asymptotic complexity is currently suboptimal.
lispy.freeVarsIn = function(sexp, boundVars) {
  if(boundVars === undefined) { boundVars = {}; }
  var freeVars = {};
  if(sexp.type === types.list || sexp.type === types.program ||
                                 sexp.type === types.imperative) {
    if(lispy.isLambdaLiteral(sexp)) {
      var bindings = _.clone(boundVars);
      _.each(_.pluck(sexp[1], 'string'), function(v) { bindings[v] = true; });
      _.each(sexp.slice(2), function(sub) {
        _.extend(freeVars, lispy.freeVarsIn(sub, bindings));
      });
    }
    else {
      _.each(sexp, function(sub) {
        _.extend(freeVars, lispy.freeVarsIn(sub, boundVars));
      });
    }
  }
  else if(sexp.type === types.identifier && !_.has(boundVars, sexp.string)) {
    freeVars[sexp.string] = true;
  }
  return freeVars;
};

lispy.bindFreeVars = function(sexp, env) {
  // We sort this for determinacy's sake.
  var freeVars = _.keys(lispy.freeVarsIn(sexp)).sort();
  var unbindables = {};
  var varsToBind = [];
  var bindings = [];
  _.each(freeVars, function(varname) {
    var val = env.lookup(varname);
    if(val) {
      varsToBind.push(mkidentifier(varname));
      bindings.push(val);
    } else {
      unbindables[varname] = mkUnboundVariable();
    }
  });

  if(_.size(unbindables)) {
    sexp = lispy.substitute(sexp, lispy.mkTopLevelEnv(unbindables));
  }

  if(varsToBind.length === 0) {
    return sexp;
  }
  else {
    //TODO implement 'let' as syntactic sugar for
    //such immediately-applied-function
    var paramsSexp = varsToBind;
    paramsSexp.type = types.list;
    var lambdaSexp = [mkidentifier('fn'), paramsSexp, sexp];
    lambdaSexp.type = types.list;
    var applySexp = [lambdaSexp].concat(bindings);
    applySexp.type = types.list;
    return applySexp;
  }
};

lispy.printSexpNonWhitespacePreserving = function(sexp) {
  var result;
  if(sexp.type === types.program || sexp.type === types.imperative) {
    result = '';
    _.each(sexp, function(subsexp) {
      result += lispy.printSexpNonWhitespacePreserving(subsexp);
      result += '\n';
    });
    return result;
  }
  else if(sexp.type === types.list) {
    result = '(';
    _.each(sexp, function(subsexp, index) {
      if(index !== 0) {
        result += ' ';
      }
      result += lispy.printSexpNonWhitespacePreserving(subsexp);
    });
    result += ')';
    return result;
  }
  else if(sexp.type === types.array) {
    result = '(array';
    _.each(sexp.value, function(subsexp) {
      result += ' ';
      result += lispy.printSexpNonWhitespacePreserving(subsexp);
    });
    result += ')';
    return result;
  }
  else if(sexp.type === types.dict) {
    result = '(dict';
    _.each(sexp.value, function(subsexpVal, subsexpKey) {
      result += ' ';
      result += lispy.printSexpNonWhitespacePreserving(subsexpKey);
```

```
   result += ' ';
   result += lispy.printSexpNonWhitespacePreserving(subsexpVal);
  });
  result += ')';
  return result;
 }
 else {
  return sexp.string;
 }
};

// TODO perhaps use position information to do a better job
// and perfectly roundtrip just-parsed lispy code.
lispy.printSexp = lispy.printSexpNonWhitespacePreserving;


//lispy.betaReduceO_1

}());
```

28 Haskell Lisp-like language implementation

28.1 README

```
===== Usage =====

> runghc Lispy.hs [test.scm]

or

> ghc -O2 Lispy.hs -o LispyExe
> ./LispyExe [test.scm]

Requires GHC and recent versions of Haskell libraries 'containers' and
'vector'.  If you don't have new enough versions, you may be able to
get these by installing the Haskell Platform then running

> cabal update; cabal install containers; cabal install vector

(I have GHC 7.6.1 as of this writing.)
```

===== What it does =====

Evaluates the Lispy expression in the passed file (defaulting to test.scm).

This prototype written in Haskell is just an interpreter; it has no game elements. Running it interprets the contents of ‘test.scm’ as an expression, evaluates it, and (if its evaluation did not exceed the number of computation steps Lispy.hs allows) prints the result.

The syntax resembles Scheme. The built-in operators are ‘lambda’, ‘if’, ‘letrec’, ‘begin’, and all the functions and constants listed in Lispy.Types.builtinNames (in Lispy/Types.hs). Unlike Scheme, functions have a fixed number of arguments (‘+’ takes exactly two arguments, no more). Like Scheme and Lua, ‘or’ and ‘and’ return their first or second argument. Every value is truthy except for ‘nil’. ‘(not nil)’ is ‘true’. ‘(if condition then else)’ requires both a then and an else branch. ‘letrec’ is local variable binding; it allows recursion and is as powerful as Scheme’s ‘let$*$’ and ‘letrec’ combined.

The table builtins are inspired by Lua. In Lua, there is a single composite data type called the ‘table’. It is an associative array implemented by (more or less) a hash table. Sequences are represented by a table with keys 1, 2, 3 ... n.

In this prototype, tables are purely-functional, key-ordered associative containers. They are implemented as self-balancing binary search trees. Tables-as-sequences start at 0 by default rather than 1, though it doesn’t make a difference: sequence iteration uses the keys’ ordering and ignores actual key values unless you look at them.

Using names like ‘lambda’ and ‘letrec’ is unsuitable for a friendly robot language, but renaming them will be trivial. Even switching the concrete syntax is relatively trivial: I deliberately avoided Lisp-specific language concepts such as macros. For prototyping, I’m sticking with programming-language-theory names and trivial Lisp-like syntax.

28.2 Lispy.hs (‘main’ module)

```haskell
module Main (main) where
```

```haskell
--import Control.Exception
--import System.Timeout
import System.Environment

--import qualified Data.Char as Char
--import Data.Text as Text
import qualified Data.Text.IO
--import Data.Ratio
import Data.List as List
--import Data.Vector as Vector
--We prefer Data.Map.Strict but it is too new, so we explicitly
--strictify any values we're inserting (just to make extra sure they
--don't use time or memory different than we're expecting).
--import Data.Map as Map
--import Data.Set as Set
--import Data.Foldable as Foldable
--import Data.Monoid as Monoid
--import Data.Sequence as Seq
--import Data.Maybe
--import Control.Applicative

import Lispy.Types
import Lispy.Parse
import Lispy.CompileToBytecode
import Lispy.Interpret
import Lispy.Show

repn :: Int -> (a -> a) -> a -> a
repn 0 _ a = a
repn n f a = repn (n-1) f (f a)

run :: Int -> LispyState -> IO ()
run n state
  | n > 3000 = putStrLn "Took too long; giving up."
  | otherwise = do
  putStrLn ("\n\n\nStep " List.++ show n)
  putStr (showStateStack state)
  run (n+1) (singleStep state)


main :: IO ()
main = do
```

```
  args <- getArgs
  let filename = case args of
        [] -> "test.scm"
        [name] -> name
        _ -> error "too many command line arguments"
  inputText <- Data.Text.IO.readFile filename
  let parsed = doParse parseSexp filename inputText
    --"abc (d  e 34) (())"
    --"(lambda (x y z) (x y (x z)))"
  let compiled = case parsed of
                   Right ast ->
                     compile builtinTextToVarIdx ast
  print parsed
  print compiled

  run 0 (startProgram compiled)

  --putStr (showStateStack (repn 12 singleStep (startProgram compiled)))
  --putStr (showStateStack (repn 300 singleStep (startProgram compiled)))
```

28.3 Lispy/Types.hs

```
{-# LANGUAGE DeriveFunctor, OverloadedStrings, BangPatterns #-}

module Lispy.Types (module Lispy.Types, module Text.Parsec) where

import Data.Text as Text
--import Data.List as List
import Data.Vector as Vector
--We prefer Data.Map.Strict but it is too new, so we explicitly
--strictify any values we're inserting (just to make extra sure they
--don't use time or memory different than we're expecting).
import Data.Map as Map
import Data.Set as Set
--import Data.Foldable as Foldable
--import Data.Monoid as Monoid
--import Data.Sequence as Seq
--import Data.Maybe
--import Control.Applicative
import Data.Int
```

```haskell
import Text.Parsec (SourcePos, sourceLine, sourceColumn)


-- |
-- bcResultName is a binding for the result of the call, permitting
-- later code to use the result value.
--
-- bcRelativeDestination is relative to the next instruction, e.g.
-- "GOTO 0" is equivalent to a NOP.  For MAKE_CLOSURE, it points to
-- the first instruction of the created function.
data BytecodeInstruction
  = CALL { bcResultName :: VarIdx,
               bcFunc :: VarIdx, bcArgs :: Vector VarIdx }
  | TAILCALL { bcFunc :: VarIdx, bcArgs :: Vector VarIdx }
  | LITERAL { bcResultName :: VarIdx, bcLiteralValue :: LispyNum }
  | NAME { bcResultName :: VarIdx, bcOriginalName :: VarIdx }
  | RETURN { bcOriginalName :: VarIdx }
  | MAKE_CLOSURE { bcResultName :: VarIdx, bcVarsInClosure :: Set VarIdx,
                   bcRelativeDestination :: Int, bcParams :: Vector VarIdx }
  | GOTO { bcRelativeDestination :: Int }
  | GOTO_IF_NOT { bcRelativeDestination :: Int, bcConditionName :: VarIdx }
  --UNTIL
  deriving (Eq, Ord, Show)
type VarIdx = ASTIdx

-- Currently we ignore int overflow and just let the numbers twos-complement
-- wrap.  Luckily, in Haskell, twos-complement sized int types are
-- modulo rather than C's unspecified behavior on signed overflow:
-- http://www.haskell.org/ghc/docs/latest/html/libraries/base/Data-Int.html
type LispyNum = Int64

type ASTIdx = Int
data SourceLocInfo = SourceLocInfo
  { sourceText :: !Text
  , sourceBegin, sourceEnd :: !Text.Parsec.SourcePos
  }
  deriving (Eq, Ord, Show)
data Located a = L { sourceLocInfo :: !SourceLocInfo, unL :: !a }
  deriving (Eq, Ord, Functor)
data AST
  = ASTNumber { astIdx :: !ASTIdx, astNumber :: !LispyNum }
  | ASTIdentifier { astIdx :: !ASTIdx, astIdentifier :: !Text }
  | ASTList { astIdx :: !ASTIdx, astList :: !(Vector (Located AST)) }
lASTIdx :: Located AST -> ASTIdx
lASTIdx (L _ ast) = astIdx ast

data CompiledProgram = CompiledProgram
  { programBytecode :: !(Vector (ASTIdx, BytecodeInstruction))
  , programAST :: !(Located AST)
  , programASTsByIdx :: !(Vector (Located AST))
  }

-- | Used in CompileToBytecode.hs
data CompileScope = CompileScope
  { compileScopeEnv :: Map Text VarIdx
  -- | when compileScopeIsTailPosition, the returned code
  -- is expected to return or tailcall, never to let the
  -- end of the returned code-block be reached.
  , compileScopeIsTailPosition :: Bool   --Maybe ASTIdx
  }


-- | Builtins: used in main and in Interpret.hs

-- Every builtin function must have worst-case running time
-- that is very short.  Constant or log(n) time are generally
-- acceptable.
--
-- TableFromSequence and TableFromPairs can take O(the number
-- of arguments you pass).  Which is a compile-time constant.
-- Function calling also takes O(number of arguments).  Probably
-- we should just set an arbitrary limit on max number of arguments,
-- like 32 or 100.
data Builtin = Plus | Minus | Times | Negate
  | LessThan | LessEqual | GreaterThan | GreaterEqual
  | Equal | NotEqual | And | Or | Not
  -- Currently, Nil is false and every other value is true.
  -- Currently, even builtins like "nil" that should be builtin
  -- constants are functions instead
  | Nil | Truth
  | EmptyTable
  | TableFromSequence -- ^ (table-sequence a b c) gives table {0:a, 1:b, 2:c}
  | TableFromPairs -- ^ (table 4 a 2 b 1 c) gives table {1:c, 2:b, 4:a}
  | TableSize -- ^ table -> number
  | TableViewKey -- ^ table, key -> iterator
  | TableUnView -- ^ iterator -> table
  | TableViewMin -- ^ table -> maybe iterator
  | TableViewMax -- ^ table -> maybe iterator
  | TableViewNext -- ^ iterator -> maybe iterator
  | TableViewPrev -- ^ iterator -> maybe iterator
  | TableViewSet -- ^ iterator -> value -> iterator
  | TableViewDelete -- ^ iterator -> iterator
  | TableViewGetKey -- ^ iterator -> value
  | TableViewGetValue -- ^ iterator -> maybe value
  | TableViewElemExists -- ^ iterator -> boolean
  -- Data.Map.split is the only log(n) function we can't create
  -- out of the above building blocks and keep it log(n), but
  -- how useful is it?  Why doesn't Data.Map have a join function
  -- that can re-join after a split in log(n) (or does union
  -- have that behavior and it's just not documented)?
  deriving (Eq, Ord, Bounded, Enum, Show, Read)

-- TODO something less arbitrary than just giving each builtin
-- a high index that probably won't conflict with things.
builtinVarIdxs :: [(VarIdx, Builtin)]
builtinVarIdxs = fmap (\bf -> (negate (fromEnum bf + 40000000), bf))
                                      [minBound..maxBound]
builtinNames :: [(Text, Builtin)]
builtinNames =
  [("+", Plus)
  ,("-", Minus)
  ,("*", Times)
  ,("negate", Negate)
  ,("<", LessThan)
  ,("<=", LessEqual)
  ,(">", GreaterThan)
  ,(">=", GreaterEqual)
  ,("=", Equal)
  ,("not=", NotEqual)
  ,("and", And)
  ,("or", Or)
  ,("not", Not)
  ,("nil", Nil)
  ,("true", Truth)
  ,("empty-table", EmptyTable)
  ,("table-sequence", TableFromSequence)
  ,("table", TableFromPairs)
  ,("table-size", TableSize)
  ,("table-view-key", TableViewKey)
  ,("table-unview", TableUnView)
  ,("table-view-min", TableViewMin)
  ,("table-view-max", TableViewMax)
  ,("table-view-next", TableViewNext)
  ,("table-view-prev", TableViewPrev)
  ,("table-view-set", TableViewSet)
  ,("table-view-delete", TableViewDelete)
  ,("table-view-get-key", TableViewGetKey)
  ,("table-view-get-value", TableViewGetValue)
  ,("table-view-elem-exists", TableViewElemExists)
  ]
builtinVarIdxToData :: Map VarIdx Builtin
builtinVarIdxToData = Map.fromList builtinVarIdxs
builtinDataToVarIdx :: Map Builtin VarIdx
builtinDataToVarIdx = Map.fromList (fmap (\(x,y)->(y,x)) builtinVarIdxs)
builtinTextToData :: Map Text Builtin
builtinTextToData = Map.fromList builtinNames
builtinDataToText :: Map Builtin Text
builtinDataToText = Map.fromList (fmap (\(x,y)->(y,x)) builtinNames)
builtinTextToVarIdx :: Map Text VarIdx
builtinTextToVarIdx = fmap (builtinDataToVarIdx Map.!)
                           (Map.fromList builtinNames)
builtinVarIdxToText :: Map VarIdx Text
builtinVarIdxToText = Map.fromList (fmap (\(x,y)->(y,x))
                                         (Map.toList builtinTextToVarIdx))


builtinsComputedValues :: StackFrameComputedValues
builtinsComputedValues = fmap b (Map.fromList builtinVarIdxs)
  where
    b Nil = NilValue
    b Truth = TrueValue
    b EmptyTable = ImmTableValue Map.empty
    b f = BuiltinFunctionValue f

-- | Used in Interpret.hs
type StackFrameComputedValues = Map VarIdx RuntimeValue
type InstructionPointer = Int
type PendingValueIdx = Int
```

```haskell
isTruthy :: RuntimeValue -> Bool
isTruthy NilValue = False
isTruthy (PendingValue _) = error "bug: isTruthy on PendingValue"
isTruthy _ = True
truthValue :: Bool -> RuntimeValue
truthValue False = NilValue
truthValue True = TrueValue

-- | Haskell's Map API does not support all the O(1)* iteration operations
-- that such a tree can support, but O(log n) is close enough that we can
-- pretend.
--
-- The iterator can point to a nonexistent key, to allow insertion there.
-- Should this be the case?
--
-- Iterator equality/ordering is unspecified but deterministic
-- between iterators into different maps. TODO is this the best choice?
--
-- (*) actually amortized O(1), worst-case O(log n), I believe.
data MapIterator k v = MapIterator !k !(Map k v) deriving (Show)

instance (Eq k) => Eq (MapIterator k v) where
  (MapIterator k1 _) == (MapIterator k2 _) = (k1 == k2)
instance (Ord k) => Ord (MapIterator k v) where
  compare (MapIterator k1 _) (MapIterator k2 _) = compare k1 k2

createMapIterator :: (Ord k) => k -> Map k v -> MapIterator k v
createMapIterator k m = MapIterator k m

mapIteratorGetMap :: (Ord k) => MapIterator k v -> Map k v
mapIteratorGetMap (MapIterator _ m) = m

mapMinIterator :: (Ord k) => Map k v -> Maybe (MapIterator k v)
mapMinIterator m =
  fmap (\((k, _), _) -> MapIterator k m) (Map.minViewWithKey m)

mapMaxIterator :: (Ord k) => Map k v -> Maybe (MapIterator k v)
mapMaxIterator m =
  fmap (\((k, _), _) -> MapIterator k m) (Map.maxViewWithKey m)
```

```
mapIteratorNext :: (Ord k) => MapIterator k v -> Maybe (MapIterator k v)
mapIteratorNext (MapIterator k m) =
  fmap (\(k’, _) -> MapIterator k’ m) (Map.lookupGT k m)

mapIteratorPrev :: (Ord k) => MapIterator k v -> Maybe (MapIterator k v)
mapIteratorPrev (MapIterator k m) =
  fmap (\(k’, _) -> MapIterator k’ m) (Map.lookupLT k m)

mapIteratorSet :: (Ord k) => MapIterator k v -> v -> MapIterator k v
mapIteratorSet (MapIterator k m) !v =
  (MapIterator k (Map.insert k v m))

mapIteratorDelete :: (Ord k) => MapIterator k v -> MapIterator k v
mapIteratorDelete (MapIterator k m) =
  (MapIterator k (Map.delete k m))

mapIteratorGetKey :: (Ord k) => MapIterator k v -> k
mapIteratorGetKey (MapIterator k _) = k

mapIteratorGetValue :: (Ord k) => MapIterator k v -> Maybe v
mapIteratorGetValue (MapIterator k m) = Map.lookup k m

mapIteratorExists :: (Ord k) => MapIterator k v -> Bool
mapIteratorExists (MapIterator k m) = Map.member k m

data RuntimeValue
  = NilValue
  | TrueValue
  | NumberValue !LispyNum
  -- | Tables, as in Lua, are used for both sequences and maps.
  -- Unlike in Lua, our tables are ordered by key.  They are
  -- balanced binary search trees.  Also unlike in Lua, they
  -- are purely functional ("Imm": immutable) currently.
  -- Creating a new version of an existing table with an extra
  -- or removed key is O(log n) average and worst case.
  --
  -- Being ordered is very convenient: it means that iterating a table
  -- whose keys are integers will just automatically go in the right order.
  -- Worst-case log(n) is also excellent.  The only downside is that
  -- the constant factor costs of binary search trees are not so great.
  --
  -- NOTE: if you put a recursive closure in a table
  -- so it has PendingValue closure members then the ordering/identity
  -- is somewhat arbitrary.  But using function values as table
  -- keys is silly anyway: anything deterministic seems fine.
  | ImmTableValue !(Map RuntimeValue RuntimeValue)
  | ImmTableViewValue !(MapIterator RuntimeValue RuntimeValue)
  | BuiltinFunctionValue !Builtin
  -- | The instruction pointer points to the beginning of the
  -- function, and the computed values are anything in the
  -- function's closure.
  | FunctionValue !LispyStackFrame !(Vector VarIdx {-params-})
  -- | This is created when a closure in a letrec
  -- closes over a value that has not yet finished being
  -- defined/evaluated.
  -- TODO investigate compiling letrecs to a clever use
  -- of the Y combinator instead (requiring no runtime
  -- support, but making recursive references be values
  -- with different identities than the function they refer to).
  -- I think I'd use Tarjan's SCC-finding algorithm.
  -- For mutual recursion, I think I'd pass to the Y combinator
  -- a function that takes an int parameter "which function are we
  -- recursing to".
  | PendingValue !PendingValueIdx
  deriving (Eq, Ord, Show)

data LispyStackFrame = LispyStackFrame
  { lsfInstructionPointer :: InstructionPointer
  , lsfComputedValues :: StackFrameComputedValues
  }
  deriving (Eq, Ord, Show)

data LispyStack = LispyStack
  { lsFrame :: LispyStackFrame
  , lsParent :: Maybe (VarIdx, LispyStack)
  }
  deriving (Eq, Ord, Show)

data LispyState = LispyState
  { lsCompiledProgram :: CompiledProgram
  , lsStack :: LispyStack
  -- | Values are never removed from this map (unless we prove
  -- that there is no longer a PendingValue value with that index
  -- sitting around anywhere).
  , lsPendingValues :: Map PendingValueIdx RuntimeValue
  , lsNextPendingValueIdx :: PendingValueIdx
  }
```

28.4 Lispy/Parse.hs

```
{-# LANGUAGE MultiParamTypeClasses, FlexibleInstances #-}

-- | This module parses S-expressions.  Just about the only semantic
-- interpretation it does is turning literal numbers into numeric
-- values.
module Lispy.Parse (parseSexp, parseLispyProgram, doParse) where

import qualified Data.Char as Char
import Data.Text as Text
import Data.List as List
import Data.Vector as Vector
--We prefer Data.Map.Strict but it is too new, so we explicitly
--strictify any values we're inserting (just to make extra sure they
--don't use time or memory different than we're expecting).
--import Data.Map as Map
import Data.Set as Set
--import Data.Foldable as Foldable
--import Data.Monoid as Monoid
--import Data.Sequence as Seq
--import Data.Maybe
import Control.Applicative

--import Data.Attoparsec.Text as P
import qualified Text.Parsec as P




import Lispy.Types

------------------ PARSING -----------------------

-- TODO using ByteString instead of Text would improve the asymptotic
-- complexity (Text doesn't have O(1) length or slicing)
-- but then we need to parse UTF-8 chars with parsec.
```

```
data LocText = LocText {-#UNPACK#-}!Int !Text
instance (Monad m) => P.Stream LocText m Char where
  uncons (LocText loc text) = return (fmap (fmap (LocText (loc+1)))
                                           (Text.uncons text))
  {-# INLINE uncons #-}

type LispyParser = P.Parsec LocText ASTIdx

parseComment :: LispyParser ()
parseComment = (P.char ’;’ >> P.skipMany (P.noneOf "\n\r")) P.<?> "comment"

parseWhitespaceAndComments :: LispyParser ()
parseWhitespaceAndComments =
  P.skipMany (P.skipMany1 P.space <|> parseComment)

schemeIdentifierChar :: LispyParser Char
schemeIdentifierChar = P.satisfy (\c ->
    {-Attoparsec has inClass-}
    Set.member c (Set.fromList "-!$%&*+.\\/:<=>?@^_~")
    || Char.isAlphaNum c
  )

parseNaturalNumber :: LispyParser LispyNum
parseNaturalNumber = go 0
  where
    go n = do
      d <- P.digit
      let n’ = n * 10 + fromIntegral (Char.digitToInt d)
      P.option n’ (go n’)

parseNumber :: LispyParser LispyNum
parseNumber =
  fmap negate (P.char ’-’ >> parseNaturalNumber)
  <|> parseNaturalNumber

parserNextASTIdx :: LispyParser ASTIdx
parserNextASTIdx = do
  idx <- P.getState
  P.putState (idx + 1)
  return idx

parseAtom :: LispyParser AST
parseAtom =
  P.choice (List.map P.try
  [ do
      num <- parseNumber P.<?> "number"
      idx <- parserNextASTIdx
      return (ASTNumber idx num)
  , do
      ident <- (P.many1 schemeIdentifierChar) P.<?> "identifier"
      idx <- parserNextASTIdx
      return (ASTIdentifier idx (Text.pack ident))
  ])
{-
  [ fmap (\rat -> Number (LispyNum rat True)) rational
  , fmap (\str -> Ident str) $ many1 schemeIdentifierChar
  , fmap (const Void) $ string "#void"
  , fmap (const (Boolean True)) $ string "#true"
  , fmap (const (Boolean False)) $ string "#false"
  , fmap (const UnboundVariable) $ string "#unbound-variable"
  -- warn/fail if it’s not a known builtin function?
  , fmap (\str -> BuiltinFunction str) $ P.char ’#’ >> many1 schemeIdentifierChar
  ]-}

parseList :: LispyParser AST
parseList = do
  _ <- P.char ’(’
  idx <- parserNextASTIdx
  asts <- P.many parseSexp
  _ <- P.char ’)’
  return (ASTList idx (Vector.fromList asts))

parseSexp :: LispyParser (Located AST)
parseSexp = P.between parseWhitespaceAndComments parseWhitespaceAndComments
  (parseWithLocation (parseAtom <|> parseList) P.<?> "s-expression")

parseWithLocation :: LispyParser a -> LispyParser (Located a)
parseWithLocation parser = do
  beginLoc <- P.getPosition
  LocText beginCharIdx beginText <- P.getInput

  parsed <- parser

  endLoc <- P.getPosition
  LocText endCharIdx _ <- P.getInput
  let info = SourceLocInfo
        (Text.take (endCharIdx - beginCharIdx) beginText)
        beginLoc
        endLoc

  return (L info parsed)

parseLispyProgram :: LispyParser (Located AST)
parseLispyProgram = do
  idx <- parserNextASTIdx
  fmap (fmap (ASTList idx . Vector.fromList))
    (parseWithLocation (P.many parseSexp))

doParse :: LispyParser a -> P.SourceName -> Text -> Either P.ParseError a
doParse parser sourceName text = let
    fullParser = do
      result <- parser
      P.eof
      return result
  in
  P.runParser fullParser 0 sourceName (LocText 0 text)
```

28.5 Lispy/CompileToBytecode.hs

```
{-# LANGUAGE ViewPatterns, OverloadedStrings #-}

module Lispy.CompileToBytecode where

import Data.Text as Text
import Data.List as List
import Data.Vector as Vector
--We prefer Data.Map.Strict but it is too new, so we explicitly
--strictify any values we're inserting (just to make extra sure they
--don't use time or memory different than we're expecting).
import Data.Map as Map
import Data.Set as Set
import Data.Foldable as Foldable
import Data.Monoid
import Data.Sequence as Seq
--import Data.Maybe
```

```
--import Control.Applicative

import Lispy.Types



scopedVarsUsed :: (Foldable f) => f BytecodeInstruction -> Set VarIdx
scopedVarsUsed = foldMap (\instr -> case instr of
    CALL _ func args        -> mappend (Set.singleton func)
                                       (foldMap Set.singleton args)
    TAILCALL func args      -> mappend (Set.singleton func)
                                       (foldMap Set.singleton args)
    LITERAL _ _             -> mempty
    NAME _ originalName     -> Set.singleton originalName
    RETURN originalName     -> Set.singleton originalName
    MAKE_CLOSURE _ vars _ _ -> vars
    GOTO _                  -> mempty
    GOTO_IF_NOT _ cond      -> Set.singleton cond
  )


indexAST :: Located AST -> Vector (Located AST)
indexAST astToIndex = let
  makeASTIdxMap :: Located AST -> Map ASTIdx (Located AST)
  makeASTIdxMap lAST@(L _ (ASTList idx children)) =
    Map.insert idx lAST (foldMap makeASTIdxMap children)
  makeASTIdxMap lAST@(L _ ast) =
    Map.singleton (astIdx ast) lAST
  astIdxMap = makeASTIdxMap astToIndex
  in Vector.generate (Map.size astIdxMap) (astIdxMap Map.!)

-- This implementation could be faster because Seq has O(1) length.
seqToVector :: Seq a -> Vector a
seqToVector s = Vector.fromList (Foldable.toList s)

-------------------- COMPILING ---------------------

compile :: Map Text VarIdx -> Located AST -> CompiledProgram
compile builtins ast = let
  instrs = compile’ (CompileScope builtins True) ast
  in CompiledProgram
    (seqToVector instrs)
    ast
    (indexAST ast)

-- We use the Data.Sequence.Seq list data structure
-- because it has O(1) cons and snoc and length,
-- and O(log n) concatenation.
--
-- A diff list that carried its length as an int
-- would probably have a much better constant factor speed wise.
-- Diff lists are also trivial to make in any language that
-- has lambdas, whereas Seq is a purely functional data structure
-- that is not as widely implemented.
-- (Alternatively, an imperative implementation could adapt this
-- code to an imperative style.)
compile' :: CompileScope -> Located AST -> Seq (ASTIdx, BytecodeInstruction)
compile' scope (L _ ast) = let
  instr i = Seq.singleton (astIdx ast, i)
  in
  case ast of
  ASTNumber _ v -> mconcat [
    instr (LITERAL (astIdx ast) v),
    if compileScopeIsTailPosition scope
    then instr (RETURN (astIdx ast))
    else mempty
    ]
  ASTIdentifier _ v ->
    case Map.lookup v (compileScopeEnv scope) of
      -- TODO show src loc in error too
      Nothing -> error ("undeclared identifier " List.++ Text.unpack v)
      Just varIdx ->
        if compileScopeIsTailPosition scope
        then instr (RETURN varIdx)
        else instr (NAME (astIdx ast) varIdx)
  ASTList _ list -> case Vector.toList list of
    [] -> error "() as a nil list literal not presently supported"
    (func : args) -> case unL func of
      ASTIdentifier _ "if" -> case args of
        --[cond, then_] ->
        [cond, then_, else_] -> let
          condCode = compile' scope{compileScopeIsTailPosition=False} cond
          thenCode = compile' scope then_
          elseCode = compile' scope else_
          elseCode2 = if compileScopeIsTailPosition scope
            then elseCode
            else mconcat [
              elseCode,
              instr (NAME (astIdx ast) (lASTIdx else_))
              ]
          thenCode2 = if compileScopeIsTailPosition scope
            then thenCode
            else mconcat [
              thenCode,
              instr (NAME (astIdx ast) (lASTIdx then_)),
              instr (GOTO (Seq.length elseCode2))
              ]
          in mconcat [
            condCode,
            instr (GOTO_IF_NOT (Seq.length thenCode2) (lASTIdx cond)),
            thenCode2,
            elseCode2
            ]
        _ -> error "syntax error: ’if’ must match (if cond then else)"
      ASTIdentifier _ "letrec" -> let
        letrecSyntaxMsg =
          "syntax error: ’letrec’ must match (letrec ((var expr)...) result)"
        in case args of
        [L _ (ASTList _ bindings), result] -> let
          parseBinding :: Located AST -> (VarIdx, Text, Located AST)
          parseBinding (L _ (ASTList _
            (Vector.toList -> [L _ (ASTIdentifier idx varname), expr])))
            = (idx, varname, expr)
          parseBinding _ = error letrecSyntaxMsg
          parsedBindings = fmap parseBinding bindings
          bindingEnv = Map.fromList (Vector.toList (
            (fmap (\ (idx, varname, _expr) -> (varname, idx)) parsedBindings)))
          resultScope = scope { compileScopeEnv =
            (Map.union bindingEnv (compileScopeEnv scope)) }
          bindingsCode = foldMap (\(idx, _varname, expr) -> mconcat [
              compile’ resultScope{compileScopeIsTailPosition=False} expr,
              instr (NAME idx (lASTIdx expr))
            ]) parsedBindings
          resultCode = compile’ resultScope result
          in mconcat [
            bindingsCode,
            resultCode,
            if compileScopeIsTailPosition scope
            then mempty
            else instr (NAME (astIdx ast) (lASTIdx result))
            ]
        _ -> error letrecSyntaxMsg
      ASTIdentifier _ "lambda" -> case args of
        [L _ (ASTList _ paramsAST), body] -> let
          params = fmap (\(L _ (ASTIdentifier idx varname)) -> (varname, idx))
                        paramsAST
          bindingEnv = Map.fromList (Vector.toList params)
          bodyScope = scope {
            compileScopeEnv = (Map.union bindingEnv (compileScopeEnv scope)),
            compileScopeIsTailPosition = True
            }
          bodyCode = compile’ bodyScope body
          bodyClosureUses = Set.intersection
            (Set.fromList (Map.elems (compileScopeEnv scope)))
            (scopedVarsUsed (fmap snd bodyCode))
          in mconcat [
            instr (MAKE_CLOSURE (astIdx ast) bodyClosureUses 1 (fmap snd params)),
              if compileScopeIsTailPosition scope
              then instr (RETURN (astIdx ast))
              else instr (GOTO (Seq.length bodyCode)),
            bodyCode
            ]
        _ -> error "syntax error: ’lambda’ must match (lambda (var...) body)"
      ASTIdentifier _ "begin" -> case List.reverse args of
        [] -> mempty
        (last_ : (List.reverse -> init_)) -> mconcat [
          foldMap (compile’ scope{compileScopeIsTailPosition=False}) init_,
          compile’ scope last_,
          if compileScopeIsTailPosition scope
          then mempty
          else instr (NAME (astIdx ast) (lASTIdx last_))
          ]
      _ -> let
        funcCode = compile’ scope{compileScopeIsTailPosition=False} func
        argsCode = foldMap (compile’ scope{compileScopeIsTailPosition=False}) args
        call = if compileScopeIsTailPosition scope
          then TAILCALL
          else CALL (astIdx ast)
```

```
        in mconcat [
          funcCode,
          argsCode,
          instr (call (lASTIdx func) (Vector.fromList (fmap lASTIdx args)))
          ]
```

28.6 Lispy/Interpret.hs

```
{-# LANGUAGE BangPatterns #-}

module Lispy.Interpret (startProgram, singleStep) where

--import Data.Text as Text
import Data.List as List
import Data.Vector as Vector
--We prefer Data.Map.Strict but it is too new, so we explicitly
--strictify any values we're inserting (just to make extra sure they
--don't use time or memory different than we're expecting).
import Data.Map as Map
import Data.Set as Set
--import Data.Foldable as Foldable
import Data.Monoid
--import Data.Sequence as Seq
--import Data.Maybe
--import Control.Applicative

import Lispy.Types
import Lispy.Show (showRuntimeValue)

------------------- INTERPRETING -------------------


varargPairs :: [RuntimeValue] -> Maybe [(RuntimeValue, RuntimeValue)]
varargPairs (a:b:rest) = fmap ((a, b) :) (varargPairs rest)
varargPairs [] = Just []
varargPairs (_:[]) = Nothing

--nothingToNil :: Maybe RuntimeValue -> RuntimeValue
maybeRuntimeValue :: (a -> RuntimeValue) -> Maybe a -> RuntimeValue
maybeRuntimeValue _ Nothing = NilValue
maybeRuntimeValue f (Just a) = f a
```

```haskell
-- since Data.Map.Strict is newish
mapFromListStrict :: (Ord k) => [(k,v)] -> Map k v
mapFromListStrict = Map.fromList . fmap (\p@(!k,!v)->p)

-- "pure" as in "no side effects"
pureBuiltinFunction :: Builtin -> [RuntimeValue] -> RuntimeValue
pureBuiltinFunction Plus [NumberValue a, NumberValue b] = NumberValue (a + b)
pureBuiltinFunction Minus [NumberValue a, NumberValue b] = NumberValue (a - b)
pureBuiltinFunction Times [NumberValue a, NumberValue b] = NumberValue (a * b)
pureBuiltinFunction Negate [NumberValue a] = NumberValue (negate a)
pureBuiltinFunction LessThan [a, b] = truthValue (a < b)
pureBuiltinFunction LessEqual [a, b] = truthValue (a <= b)
pureBuiltinFunction GreaterThan [a, b] = truthValue (a > b)
pureBuiltinFunction GreaterEqual [a, b] = truthValue (a >= b)
pureBuiltinFunction Equal [a, b] = truthValue (a == b)
pureBuiltinFunction NotEqual [a, b] = truthValue (a /= b)
-- We use the Lua convention of returning the first truthy value
-- that makes the and/or operator have the correct truthiness.
pureBuiltinFunction And [a, b] = if isTruthy a then b else a
pureBuiltinFunction Or [a, b] = if isTruthy a then a else b
pureBuiltinFunction Not [a] = truthValue (not (isTruthy a))
pureBuiltinFunction TableFromSequence vals =
    ImmTableValue (mapFromListStrict (List.zip (fmap NumberValue [0..]) vals))
pureBuiltinFunction TableFromPairs vals =
  case varargPairs vals of
    Nothing -> error "Odd number of arguments to table-from-pairs"
    Just pairs -> ImmTableValue (mapFromListStrict pairs)
pureBuiltinFunction TableSize [ImmTableValue m] =
  NumberValue (fromIntegral (Map.size m))
pureBuiltinFunction TableViewKey [ImmTableValue m, k] =
  ImmTableViewValue (createMapIterator k m)
pureBuiltinFunction TableUnView [ImmTableViewValue i] =
  ImmTableValue (mapIteratorGetMap i)
pureBuiltinFunction TableViewMin [ImmTableValue m] =
  maybeRuntimeValue ImmTableViewValue (mapMinIterator m)
pureBuiltinFunction TableViewMax [ImmTableValue m] =
  maybeRuntimeValue ImmTableViewValue (mapMaxIterator m)
pureBuiltinFunction TableViewNext [ImmTableViewValue i] =
  maybeRuntimeValue ImmTableViewValue (mapIteratorNext i)
pureBuiltinFunction TableViewPrev [ImmTableViewValue i] =
  maybeRuntimeValue ImmTableViewValue (mapIteratorPrev i)
```

```
pureBuiltinFunction TableViewSet [ImmTableViewValue i, v] =
  ImmTableViewValue (mapIteratorSet i v)
pureBuiltinFunction TableViewDelete [ImmTableViewValue i] =
  ImmTableViewValue (mapIteratorDelete i)
pureBuiltinFunction TableViewGetKey [ImmTableViewValue i] = mapIteratorGetKey i
pureBuiltinFunction TableViewGetValue [ImmTableViewValue i] =
  maybeRuntimeValue id (mapIteratorGetValue i)
pureBuiltinFunction TableViewElemExists [ImmTableViewValue i] =
  truthValue (mapIteratorExists i)
pureBuiltinFunction _ _ =
  error "wrong number of arguments to builtin function (or bug non-function builtin)"

startProgram :: CompiledProgram -> LispyState
startProgram program = LispyState
  program
  (LispyStack (LispyStackFrame 0 builtinsComputedValues) Nothing)
  mempty
  0

singleStep :: LispyState -> LispyState
singleStep state@(LispyState
              program@(CompiledProgram bytecode _ _)
              stack@(LispyStack
                frame@(LispyStackFrame instructionPointer computedValues)
                parent)
              pendingValues
              nextPendingValueIdx) =
  case snd (bytecode Vector.! instructionPointer) of
    CALL result func args -> call (Just result) func args state
    TAILCALL func args -> call Nothing func args state
    LITERAL result value -> bindValue result (NumberValue value) state
    NAME result origName -> bindValue result (computedValues Map.! origName) state
    RETURN origName -> ret (computedValues Map.! origName) state
    MAKE_CLOSURE result vars dest params -> case
      Set.foldl'
        (\(nextPending, newComputedValues, varsInClosure)
          varInClosure ->
           case Map.lookup varInClosure computedValues of
             Just !val -> (nextPending, newComputedValues,
               Map.insert varInClosure val varsInClosure)
             Nothing -> let
               !newPending = PendingValue nextPending
               nextPending' = nextPending+1
               newComputedValues' = Map.insert varInClosure
                 newPending newComputedValues
               in (nextPending', newComputedValues',
               Map.insert varInClosure newPending varsInClosure))
        (nextPendingValueIdx, computedValues, mempty)
        vars
      of (nextPending, newComputedValues, varsInClosure) -> let
            value = FunctionValue
              (LispyStackFrame (instructionPointer+1+dest) varsInClosure)
              params
            -- Note: the bindValue happens *after* putting in
            -- newComputedValues.  This lets bindValue detect
            -- whether the current closure is pending (self-recursive
            -- functions) so that it will define it in pendingValues.
            in bindValue result value
              (updateStackFrame (\fr ->
                fr{ lsfComputedValues = newComputedValues }) state
              ){
                lsNextPendingValueIdx = nextPending
              }
    GOTO dest -> goto (instructionPointer+1+dest) state
    GOTO_IF_NOT dest cond ->
      if not (isTruthy (computedValues Map.! cond))
      then goto (instructionPointer+1+dest) state
      else goto (instructionPointer+1) state


dePendValue :: LispyState -> RuntimeValue -> RuntimeValue
dePendValue state (PendingValue idx) =
  case Map.lookup idx (lsPendingValues state) of
    Nothing -> error "Used a value in a letrec before it was defined"
    Just val -> dePendValue state val
dePendValue _ val = val

call :: Maybe VarIdx -> VarIdx -> Vector VarIdx -> LispyState -> LispyState
call resultElseTail func args state = let
  computedValues = (lsfComputedValues (lsFrame (lsStack state)))
  funcVal = computedValues Map.! func
  argVals = fmap (computedValues Map.!) args
  in case dePendValue state funcVal of
    FunctionValue initialFrame params -> let
```

```
        argEnv = mapFromListStrict (Vector.toList (Vector.zip params argVals))
        newStackFrame = initialFrame{lsfComputedValues =
            Map.union argEnv (lsfComputedValues initialFrame)}
      in state{
        lsStack = case resultElseTail of
          Nothing -> --tail call
            LispyStack newStackFrame (lsParent (lsStack state))
          Just result -> --non-tail
            LispyStack newStackFrame (Just (result, lsStack state))
        }
    BuiltinFunctionValue bf ->
      callBuiltinFunction resultElseTail bf args state
    _ -> error "runtime error: calling a non-function"

callBuiltinFunction :: Maybe VarIdx -> Builtin -> Vector VarIdx
                    -> LispyState -> LispyState
callBuiltinFunction resultElseTail bf args state = case bf of
  _ -> let
    computedValues = (lsfComputedValues (lsFrame (lsStack state)))
    val = pureBuiltinFunction bf
      (Vector.toList (fmap (dePendValue state . (computedValues Map.!)) args))
    in case resultElseTail of
      Nothing -> --tail call
        ret val state
      Just result -> --non-tail
        bindValue result val state

updateStackFrame :: (LispyStackFrame -> LispyStackFrame)
                 -> LispyState -> LispyState
updateStackFrame f state = state{ lsStack = let stack = lsStack state in
  stack{ lsFrame = f (lsFrame stack) } }

bindValue :: VarIdx -> RuntimeValue -> LispyState -> LispyState
bindValue result !value state = let
  frame = lsFrame (lsStack state)
  computedValues = lsfComputedValues frame
  instructionPointer = lsfInstructionPointer frame
  in case
    Map.insertLookupWithKey (\_ newValue _ -> newValue)
      result value computedValues of
  (oldVal, newComputedValues) -> let
    newState = updateStackFrame (\fr -> fr{
        lsfComputedValues = newComputedValues,
        lsfInstructionPointer = instructionPointer+1
      }) state
    in case oldVal of
      Just (PendingValue pendingValueIdx) ->
        newState{
          lsPendingValues = Map.insert pendingValueIdx value
            (lsPendingValues state)
          }
      _ -> newState

goto :: InstructionPointer -> LispyState -> LispyState
goto absoluteDest state = updateStackFrame (\fr -> fr{
    lsfInstructionPointer = absoluteDest
  }) state

ret :: RuntimeValue -> LispyState -> LispyState
ret retVal state =
  case lsParent (lsStack state) of
    Nothing -> error ("returning from the main program: "
      List.++ showRuntimeValue state retVal)
    Just (retValDest, newStack) ->
      bindValue retValDest retVal (state{lsStack = newStack})
```

28.7 Lispy/Show.hs

```
module Lispy.Show where

--import qualified Data.Char as Char
import Data.Text as Text
import Data.List as List
import Data.Vector as Vector
--We prefer Data.Map.Strict but it is too new, so we explicitly
--strictify any values we're inserting (just to make extra sure they
--don't use time or memory different than we're expecting).
import Data.Map as Map
import Data.Set as Set
import Data.Foldable as Foldable
import Data.Monoid
--import Data.Sequence as Seq
--import Data.Maybe
```

```
--import Control.Applicative

import Lispy.Types

---------------------- SHOWING ----------------------

instance (Show a) => Show (Located a) where
  showsPrec prec (L _ val) = showsPrec prec val

instance Show AST where
  showsPrec _ (ASTNumber _ num) = shows num
  showsPrec _ (ASTIdentifier _ ident) = showString (Text.unpack ident)
  showsPrec _ (ASTList _ members) =
    if Vector.null members
    then showString "()"
    else
    showChar '(' .
    Vector.foldr1
      (\s rest -> s . showChar ' ' . rest)
      (fmap (shows . unL) members) .
    showChar ')'

instance Show CompiledProgram where
  showsPrec _ (CompiledProgram bytecode (L progSource _) astsByIdx) =
    showString (Text.unpack (sourceText progSource)) .
    showChar '\n' .
    appEndo (foldMap
      (\(bytecodeIdx, (astidx, instr)) -> Endo (
        let
          motherInstructionDesc = showsVarIdx astsByIdx astidx
          bytecodeIdxDesc = shows bytecodeIdx
        in
        showString (List.replicate
          (max 0 (3 - List.length (bytecodeIdxDesc ""))) ' ') .
        shows bytecodeIdx .
        showString "  " .
        motherInstructionDesc .
        showString (List.replicate
          (max 1 (22 - List.length (motherInstructionDesc ""))) ' ') .
        showsBytecodeInstruction astsByIdx bytecodeIdx instr . showChar '\n'
      ))
      (Vector.indexed bytecode))
```

```
showsVarIdx :: Vector (Located AST) -> VarIdx -> ShowS
showsVarIdx astsByIdx idx =
  case Map.lookup idx builtinVarIdxToText of
    Just builtinName ->
      showChar '#' .
      showString (Text.unpack builtinName)
    Nothing ->
      shows idx .
      showChar '(' .
      case astsByIdx Vector.! idx of
        L source ast -> showsASTConciseSummary ast .
          showChar ':' . shows (sourceLine (sourceBegin source)) .
          showChar ':' . shows (sourceColumn (sourceBegin source)) .
          showChar ')'

showsASTConciseSummary :: AST -> ShowS
showsASTConciseSummary ast = case ast of
  ASTNumber _ num -> shows num
  ASTIdentifier _ ident -> showString (Text.unpack ident)
  ASTList _ members -> case Vector.headM members of
    Just (L _ ast') -> showChar '(' . showsASTConciseSummary ast' .
      if Vector.length members > 1 then showString "...)" else showChar ')'
    Nothing -> showString "()"

showsBytecodeInstruction :: Vector (Located AST) -> Int -> BytecodeInstruction
                         -> ShowS
showsBytecodeInstruction astsByIdx bytecodeIdx bytecode = let
    var = showsVarIdx astsByIdx
    relDest dest = showChar '+' . shows dest . showChar '(' .
      shows (bytecodeIdx + 1 + dest) . showChar ')'
  in case bytecode of
  CALL result func args ->
    showString "CALL " .
    var result .
    showString " = " . var func .
    appEndo (foldMap (\idx -> Endo (showChar ' ' . var idx)) args)
  TAILCALL func args ->
    showString "TAILCALL " .
    var func .
    appEndo (foldMap (\idx -> Endo (showChar ' ' . var idx)) args)
  LITERAL result value ->
```

```
    showString "LITERAL " .
    var result .
    showString " = " .
    shows value
  NAME result origName ->
    showString "NAME " .
    var result .
    showString " = " .
    var origName
  RETURN origName ->
    showString "RETURN " .
    var origName
  MAKE_CLOSURE result vars dest params ->
    showString "MAKE_CLOSURE " .
    var result .
    showString " =" .
    ( if Set.null vars
      then id
      else
      showString " [" .
      List.foldr1
        (\s rest -> s . showChar ’ ’ . rest)
        (fmap var (Set.toList vars)) .
      showChar ’]’
    ) .
    showChar ’ ’ .
    relDest dest .
    appEndo (foldMap (\idx -> Endo (showChar ’ ’ . var idx)) params)
  GOTO dest ->
    showString "GOTO " . relDest dest
  GOTO_IF_NOT dest cond ->
    showString "GOTO_IF_NOT " . relDest dest .
    showChar ’ ’ .
    var cond

{-
showsProgramBytecode :: Vector (ASTIdx, BytecodeInstruction) -> ShowS
showsProgramBytecode = appEndo . foldMap
  (\(_, instr) -> Endo (shows instr . showChar ’\n’))
showProgramBytecode :: Vector (ASTIdx, BytecodeInstruction) -> String
showProgramBytecode = flip showsProgramBytecode ""
-}
```

```
showsStackFrame :: LispyState -> LispyStackFrame -> ShowS
showsStackFrame state (LispyStackFrame instructionPointer computedValues) =
  let program = lsCompiledProgram state in
  showString "Instruction pointer: " .
  shows instructionPointer .
  let (astidx, code) = (programBytecode program) Vector.! instructionPointer in
  showString "  " .
  showsVarIdx (programASTsByIdx program) astidx .
  showString "  " .
  showsBytecodeInstruction (programASTsByIdx program) instructionPointer code .
  appEndo (foldMap (\(idx, val) -> Endo (
    showString "\n\t" .
    showsVarIdx (programASTsByIdx program) idx .
    showString " = " .
    showsRuntimeValue state val
    )) (Map.toList computedValues)) .
  showChar '\n'

showsStack :: LispyState -> LispyStack -> ShowS
showsStack state (LispyStack frame parent) =
  let program = lsCompiledProgram state in
  showsStackFrame state frame .
  case parent of
    Just (retValDest, nextStack) ->
      showString "  will return value to " .
      showsVarIdx (programASTsByIdx program) retValDest .
      showChar '\n' .
      showsStack state nextStack
    Nothing -> id

showsStateStack :: LispyState -> ShowS
showsStateStack state@(LispyState _ stack _ _) = showsStack state stack

showStateStack :: LispyState -> String
showStateStack ls = showsStateStack ls ""


showsMap :: (k -> ShowS) -> (v -> ShowS) -> Map k v -> ShowS
showsMap kf vf m = case Map.toList m of
  [] -> showString "{}"
  (p1:ps) -> let showPair (k, v) = kf k . showString ": " . vf v in
    showChar ’{’ .
    showPair p1 .
    appEndo (foldMap (\p -> Endo (showString ", " . showPair p)) ps) .
    showChar ’}’

-- How do you serialize a closure?  By its ASTIdx?  And then you can
-- map that back to the code... by choosing the first one that matches
-- that?  Sounds good.  Because we don’t trust random pointers into
-- our bytecode, or the compiler is nondeterministic
-- (well, with this bytecode, that’d be harmless, but I gather Lua’s is riskier).
--
-- Closures could alternatively be serialized as their AST or source text,
-- with bindings for the textual names of all closed-over variables.
--
-- What is a bit more difficult is serializing the stack, in that it can
-- have several opcodes corresponding to one AST value.
-- Well, they could be serialized with an int saying which one
-- they’re on (the first being 0, and the last being n-1, where
-- n is the number of opcodes generated by a given AST node).

showsRuntimeValue :: LispyState -> RuntimeValue -> ShowS
showsRuntimeValue _ NilValue = showString "nil"
showsRuntimeValue _ TrueValue = showString "true"
showsRuntimeValue _ (NumberValue n) = shows n
showsRuntimeValue s (ImmTableValue m) =
  showsMap (showsRuntimeValue s) (showsRuntimeValue s) m
showsRuntimeValue s (ImmTableViewValue i) =
  showsRuntimeValue s (mapIteratorGetKey i) .
  showChar ’@’ .
  showsMap (showsRuntimeValue s) (showsRuntimeValue s) (mapIteratorGetMap i)
showsRuntimeValue _ (BuiltinFunctionValue bf) =
  showString (Text.unpack (builtinDataToText Map.! bf))
showsRuntimeValue state (FunctionValue
    (LispyStackFrame instructionPointer computedValues) _) =
  let program = lsCompiledProgram state in
  showString "<closure: " .
  shows instructionPointer .
  let (astidx, _) = (programBytecode program) Vector.! instructionPointer in
  showString " (" .
  showsVarIdx (programASTsByIdx program) astidx .
  showString ") " .
```

```haskell
  showsMap (showsVarIdx (programASTsByIdx program))
           (showsRuntimeValue state)
           computedValues .
  showChar '>'
showsRuntimeValue state (PendingValue pv) =
  case Map.lookup pv (lsPendingValues state) of
    Nothing -> showString "<pending " . shows pv . showChar '>'
    Just v -> let
        -- avoid emitting infinite text for loops
        sanitizedState = state{lsPendingValues =
          Map.delete pv (lsPendingValues state) }
      in
      showString "<via pending " .
      shows pv .
      showString ": " .
      showsRuntimeValue sanitizedState v .
      showChar '>'

showRuntimeValue :: LispyState -> RuntimeValue -> String
showRuntimeValue p v = showsRuntimeValue p v ""
```

Bibliography

[1] Geek feminism wiki. http://geekfeminism.wikia.com/.

[2] Craig Brandon, Nicole Garman, and Michael Ryan. *Good Night Irene: Stories and Photos about the tropical storm that devastated Vermont, the Catskills, and the Berkshires.* Surry Cottage Books, Keene, New Hampshire, 2012.

[3] Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering.* Reading, Massachusetts, anniversary edition, 1995.

[4] C.T. Lawrence Butler and Amy Rothstein. *On Conflict and Consensus: a handbook on Formal Consensus decisionmaking.* Food Not Bombs Publishing, (800) 569-4054 or www.consensus.net, second edition, 1987.

[5] E. Gabriella Coleman. *Coding Freedom: The Ethics and Aesthetics of Hacking.* Princeton University, Princeton, New Jersey, 2012.

[6] Edsger W. Dijkstra. A case against the GO TO statement, 1968. http://www.cs.utexas.edu/users/EWD/transcriptions/EWD02xx/EWD215.html.

[7] Peter Elbow. *Writing without teachers.* Oxford University Press, second edition, 1998.

[8] Roger Fisher, William Ury, and Bruce Patton. *Getting to yes: Negotiating agreement without giving in.* Penguin Group USA, New York, second edition, 2011.

[9] Karl Fogel. *Producing Open Source Software.* 2005-2010. http://producingoss.com/.

[10] David Goldberg. What every computer scientist should know about floating-point arithmetic, 1991. http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html.

[11] Cecilia M Gorriz and Claudia Medina. Engaging girls with computers through software games. *Communications of the ACM*, 43(1):42–49, 2000.

[12] David Graeber. *Fragments of an Anarchist Anthropology.* Prickly Paradigm Press, Chicago, 2004.

[13] Paul Graham. Beating the averages, 2001-2003. `http://www.paulgraham.com/avg.html`.

[14] Reuben Hersh. *What is mathematics, really?* Oxford University Press, 1997.

[15] Sam Hughes. Gay marriage: the database engineering perspective, 2008. `http://qntm.org/gay`.

[16] William Strunk Jr. and E.B. White. *The elements of style.* Fourth edition, 2000.

[17] et al. Keith Knight, Mat Schwarzman. *Beginner's Guide to Community-Based Arts: Ten Graphic Stories about Artists, Educators & Activists across the U.S.* New Village Press, Oakland.

[18] Andrew Kennedy. *Programming languages and dimensions.* Number 391. University of Cambridge, Computer Laboratory, 1996.

[19] Andrew Kennedy. Types for units-of-measure: Theory and practice. 2009.

[20] Brian W. Kernighan and P. J. Plauger. *The elements of programming style.* Bell Telephone Laboratories, second edition, 1978.

[21] Richard A. Lanham. *Style: An anti-textbook.* Paul Dry Books, Philadelphia, 2007.

[22] Liz Lerman and John Borstel. *Liz Lerman's Critical Response Process.* Liz Lerman Dance Exchange, Takoma Park, Maryland, 2003.

[23] Emily Meyer and Louise Z. Smith. *The Practical Tutor*. Oxford University Press, New York, 1987.

[24] Eric A. Meyer. "considered harmful" essays considered harmful, 2002. `http://meyerweb.com/eric/comment/chech.html`.

[25] Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[26] Niels Moller and Torbjorn Granlund. Improved division by invariant integers. *Computers, IEEE Transactions on*, 60(2):165–175, 2011.

[27] Nick Montfort, Patsy Baudoin, John Bell, Ian Bogost, Jeremy Douglass, Mark C. Marino, Michael Mateas, Casey Reas, Mark Sample, and Noah Vawter. *10 PRINT CHR$(205.5+RND(1)); : GOTO 10*. The MIT Press, Cambridge, Massachusetts, November 2012.

[28] Chris Okasaki. *Purely functional data structures*. PhD thesis, Carnegie Mellon, 1996.

[29] Seymour Papert. *Mindstorms: Children, Computers, And Powerful Ideas*. Basic Books, New York, 1980.

[30] Charles Perrow. *Normal Accidents: Living With High-Risk Technologies*. Basic Books, New York, 1984.

[31] Brenda D Phillips, Deborah SK Thomas, Alice Fothergill, and Lynn Blinn-Pike. *Social vulnerability to disasters*. CRC Press Boca Raton, 2010.

[32] Lydia Pintscher, editor. *Open Advice: FOSS: What We Wish We Had Known When We Started*. 2012. `http://open-advice.org/`.

[33] Julie Powers-Boyle. *Ecology of Disturbances: Fire and Ice in the Forests.* Undergraduate thesis, Marlboro College, Marlboro, Vermont, 2010.

[34] Eric S. Raymond. The cathedral and the bazaar, 1997-2000. `http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/`.

[35] Eric S. Raymond and Rick Moen. How to ask questions the smart way, 2001. `http://www.catb.org/esr/faqs/smart-questions.html`.

[36] Mitchel Resnick. *Turtles, Termites and Traffic Jams: Explorations in Massively Parallel Microworlds.* The MIT Press, Cambridge, Massachusetts, 1994.

[37] Michael Rohd. *Theatre for Community, Conflict and Dialogue: The Hope Is Vital Training Manual.* Heinemann, Portsmouth, NH.

[38] Marshall B. Rosenberg. *Nonviolent communication: A language of life.* PuddleDancer Press, Encinitas, California, 2003.

[39] Matthew Skala. What colour are your bits?, 2004. `http://ansuz.sooke.bc.ca/entry/23`.

[40] Diomidis Spinellis. *Code Reading: The Open Source Perspective.* Pearson, Boston, 2003.

[41] Richard M. Stallman. *Free Software, Free Society: selected essays of Richard M. Stallman.* Free Software Foundation, Boston, 2002.

[42] Starhawk. *The Empowerment Manual: A Guide for Collaborative Groups.* New Society Publishers, Gabriola Island, BC, Canada, 2011.

[43] Mads Tofte. *Operational semantics and polymorphic type inference.* PhD thesis, University of Edinburgh, 1988.

[44] L.H. Willis and G.V. Welch. *Aging power delivery infrastructures*, volume 12. CRC Press, New York, 2000.

[45] Robert A. Wilson. *Graphs, colourings and the four-colour theorem.* Oxford University Press, New York, 2002.

[46] Jamie Zawinski. The CADT model, 2003. `http://www.jwz.org/doc/cadt.html`.