CS 106 Syllabus

Norman Ramsey

Spring 2023

Contents

Introduction and welcome

How does one implement a scripting language like, say, Python? The best simple method is to translate the high-level language through a series of intermediate languages into code that runs on a virtual machine. In this course, you'll build both a Universal Forward Translator and a Simple Virtual Machine.

- A Simple Virtual Machine resembles the Universal Machine from CS 40. The SVM will enable us to ignore the unpleasant problems of language implementation by throwing (virtual) machine resources at them. It will also provide a congenial platform for learning about garbage collection.
- A Universal Forward Translator takes source code to (virtual) machine code in a long series of very small, simple steps. Each step does one job well, and each one is so simple that when you finish it, you'll be asking, "was *that* all there is to it?" The UFT will also provide a congenial platform for practicing functional programming.

You'll build as part of a learning community that can freely share ideas and code. The implementations you build will be useful by themselves and will be fun to extend further. And creating them will develop both your machine-level programming technique and your functional-programming technique.

What background do I need?

To succeed in the course, you'll need two kinds of background:

- For the SVM, you will need to be comfortable manipulating bits, bytes, words and pointers. Ideally, you will already have experience with a simple machine emulator, such as the Universal Machine from CS 40.

  CS 40 is more than adequate preparation for the SVM.
- For the UFT, you will need to be comfortable with the jargon and techniques of functional programming. You will need to have experience with a functional language such as Clojure, Elm, Erlang, Haskell, OCaml, Racket, or Standard ML. Experience with Standard ML will be ideal, as some translation infrastructure will be provided in Standard ML.

  CS 105 is more than adequate preparation for the UFT.

Parts of the SVM will be specified using operational semantics, at the level covered in CS 105. If you haven't had 105, don't worry;

the semantics are there to clarify details, and you'll be able to work effectively without them.

How will we study the topic?

The course will be organized as follows:

- The project is divided into *modules*, and each module is supported by two synchronous classes. Thursday classes will be *labs* offering a supervised opportunity to start building the code for the module; Tuesday classes will be *code reviews* examining code turned in the previous night.

- Class meetings will be supported by extensive handouts and the occasional amateur video, not lectures. You are expected to read and/or watch before the Thursday lab—that's why the course is worth 4 SHU's. Some concepts, like parsing combinators, will be supported via papers from the professional literature.

  No textbooks are required. For the module on garbage collection, I recommend my book *Programming Languages: Build, Prove, and Compare*, chapters 3 and 4. But the book is expensive and I don't want anyone to feel they have to buy it. I have asked for a copy to be placed on reserve at Tisch.

- The project will start with the virtual-machine language, continue with virtual object code, and will add incrementally more sophisticated languages until eventually it arrives at a flexible, expressive high-level language in the Scheme family. You'll be running code with four VM instructions in the very first week, and almost every week you will add to the set of codes you can run.

- Most of the project will be built outside of class. And each Monday night at 11:59pm US Eastern time, you'll deliver a piece: your *homework*. Then you'll have 24 hours (and class review) to reflect on what you've learned, and on Tuesday night, you'll submit a short *reflection* listing what you've learned. As evidence for what you've learned, your reflection will often cite code that you delivered the previous night. As detailed below, your reflection determines your grade.

  Homework will receive feedback from your peers and/or from the course staff.

- At the end of the term, you will participate in a workshop where you will present your work to your peers and to a jury of outside experts. Afterward, you'll make any changes suggested during the workshop, and you'll submit one final reflection.

- There will be no papers or examinations.

What will happen in labs?

Lab will look a lot like recitation for CS 105: you'll work in small teams on problems directly related to homework. There are three important differences:

- You'll prepare for lab by reading handouts (and watching the occasional video), not by attending lectures.

- If you aren't prepared for lab, you'll get less out of it than you would out of a comparable recitation. Essentially your only option will be to use the lab time for preparation.

- Unlike a recitation, the lab problems won't be *like* problems on the homework; the code you write in lab you will *be* part of your homework. Your lab results can be shared with your peers, and you can use your peers' lab work (with acknowledgement) in your own homework.

What will happen at code review?

Code review has two purposes:

- To identify and correct coding problems ("I don't understand how this works").

- To identify good practices that should be spread throughout the code ("This is great; please write more like it").

Code review will take two forms: *Small-group code review* will involve two or three students mutually reviewing each other's code for about 30 minutes. *Plenary code review* will involve a team presenting to a review panel composed of three students, which the whole class will observe.

Classes devoted to code review will also have time for general questions.

What does small-group code review look like?

Small-group code review will be based on the answers to three questions that are required with every homework: what you are proud of, what you want help with, and what you would like the instructor to review. These questions will be the focus of small-group work with two other students. Each student's code will be the center of attention for about 10 minutes, for a total of about half an hour. There will be no written record.

What does plenary code review look like?

Plenary code review is a more heavyweight process: one project gets everyone's attention for half an hour. Plenary review will use the projection screen in our classroom. In order to help the most students, pair work and group work will be prioritized—but solo contributors will probably also have opportunities to have their work reviewed in the plenary setting.

Plenary review divides the class into three parts: the *contributors*, who have collectively written the code under review, the *panelists*,

who evaluate the code under review, and the *class*, who watch and learn. Contributors have these roles:

- The *presenter* does the talking. They will answer questions or, if prompted by the panel, choose code to be shown.
- The *navigator* runs the display. They will make sure that the code being described is available for the whole class to review.

In order to give everyone a chance to learn what it is like to present code, the instructor may sometimes ask a different contributor to take over the presenter's role.

Panelists identify problems and good practices. (Every code base has problems—we'll find them.) Panelists need not propose fixes. The three panelists have these roles:

- The *regular panelist* critiques the code being presented.
- Just like the regular panelist, the *moderator* critiques the code being presented. The moderator also keeps the review on track. That may include managing questions from the rest of the class; helping the presenter if they get lost; making sure panelists don't waste the presenter's time, e.g., by speaking all at once or arguing with each other; suggesting that it may be time to move on to another part of the code; or reminding other panelists that code different from theirs isn't necessarily a problem.
- The *recordkeeper* turns every conversion into two bulleted lists: one of identified problems and one of good practices. If fixes are suggested, which is optional, the recordkeeper puts them on a third list. The recordkeeper may also critique the code being presented, but their primary job is creating the record.

The presenter and the panelists speak aloud. The *rest of the class* are encouraged to shout out questions, challenges, and interjections.

Presenters will do all of the following:

- Set the context for the review by explaining the design from the top down
- Focus on trouble spots and good work (in context)
- Respond well to questions, especially questions about design, organization, names, and interfaces/APIs/types
- Acknowledge mistakes and unfortunate choices
- Push back if the panel mischaracterizes a non-mistake

Panelists will be given a set of recommended prompts.

Who presents? Who's on the panel?

Everyone will get a turn in both roles. We'll take volunteers first, and we'll draft people as needed. Early volunteers will earn bonus participation points.

Panel reviews? Has Norman lost his mind?

We're trying to help everyone become more thoughtful engineers, and we especially want to help everyone learn to think clearly about code. The most effective way I know to do this is through structured, supervised code reviews. And I promise that you will find it fascinating to see, in depth, how other students have tackled problems that you yourself are working on.

Code review is hard to do well, but good code reviews will teach you more about coding, more deeply, than you could learn in any other way. If you can think about code that other people have written, and if you can write your own code so that others can easily think about it, you will have acquired an invaluable skill.

I have done code reviews before, and we learned that it's cool to see how other people approach problems. We also learned that presenting our technical work in front of others can be intimidating. It gets easier with practice. A good semester of code review will teach you a lot, and the practice will help you significantly in your job interviews.

Code reviews may be difficult to get started, but if we lean on the strong collaborative culture that is established throughout most of the Tufts CS department, we should be able to craft a good experience for everyone. I'm counting on you.

How will I need to prepare before class?

Our time in labs and at code reviews is precious, and you'll need to hit the ground running.

- For a lab, you'll need to know the material in the background reading, both general background and (for many modules) a focused handout directed at that lab. Each module will tell you exactly what you need to do to get ready for lab.
- For small-group code review, you'll need to have answered the three questions with your Monday-night homework.
- If your team is presenting code for plenary review, *all* members of the team will need to agree on a plan and on high-value areas to present. Your team can agree on a presenter beforehand, but every contributor will need to be prepared to take over if needed.
- If you're serving as a panelist for a plenary code review, you'll need to think about what questions you want to ask and what code you wish to have presented. You'll have the option of seeing the code beforehand, but in most cases you'll have worked on the same problem, and that will be preparation enough.

How can I avoid working over Spring break?

According to the schedule laid down by the registrar, we have no lab on Thursday, February 23. (On that day, Tufts will run a

Monday schedule.) This annoyance can be handled in one of two ways:

1. We can stick to the registrar's schedule. That will leave a week in February with no module. We will use February 28 for general code review.

   If we choose this alternative, we will start module 9 (functions in the translator) on March 16, which is the Thursday before Spring break, we will turn in the code on Monday, March 27, and we will have the code review on March 28.

2. As an alternative, we can schedule a "stealth" lab for February 23 or 24. (I am willing to run two "stealth" labs: one on Thursday and one on Friday.) We would then have normal code review on February 28th, and we would have a module for every week in February.

   If we choose this alternative, module 9 will be complete on Tuesday, March 14. To make up for the stealth lab, I will cancel class on Thursday, March 16, which is the Thursday before Spring break. And we will use Tuesday, March 28 either for general code review or for planning module 12.

We will decide on an alternative on Thursday, February 9, which is the day after the drop date.

How will I work with my peers?

Interactions during labs and code reviews will take place as described above. Interactions around homework will depend on how you wish to work. The project is ambitious enough that only rare individuals are likely to complete an entire SVM *and* UFT working on their own. I encourage you to work with a partner. Working with larger groups is also permitted, although I'm not sure the project offers enough scope for a larger group.

The ground rules are as follows:

- Code may be shared freely. In particular, your SVM and your UFT may include components written by other students or by the course staff. Nobody is expected to implement every component by themselves. The only code that has to be your own work is code that you want to cite as evidence that you have learned something. Expectations for what might be cited will be clearly identified in each module.
- Pair programming is not required. You are welcome to divide work with your partner or partners in any way you want.
- Although your Tuesday-night reflection should be informed by discussions with your classmates and your programming partners, the actual words you submit must be entirely your own work. Any code you cite as evidence for learning must be either your own work or joint work with a pair-programming partner. (If something you've learned is supported only by your partner's work, you'll have to explain how it is that you learned something from someone else's code.)
- For every homework, and for labs after the first few weeks, you will have the option of working alone.
- You will also have the option of changing partners frequently, including having one partner for a lab and another partner for the associated homework. Or if you prefer, you can keep one partner for many weeks—even the whole semester.
- Any code written in partnership is community property: all partners may continue to use it throughout the term. All partners are also free to share the code with other classmates.

To help you decide with whom to work and how long, I recommend this heuristic: if you are having a great experience, keep doing what you are doing. If you are just having a good experience, try changing something. And if you ever have a bad experience, please come talk to me.

What about ChatGPT and Copilot?

You are free to use ChatGPT and Github Copilot for anything you want. If you use these tools thoughtfully, you can even earn depth points (explained below).

If you do use ChatGPT, please do highlight the results during code review. Let us know what the tool did well, how you figured out some effective prompts, and what you had to do to make the bot's code good enough to deploy.

How should I organize my work, especially with partners?

You have a lot of freedom. I recommend that you use it like this:

1. Each module assumes that you have working code from previous modules. But it doesn't have to be *your* code. Code from previous modules is rarely edited, so if you want to, you can usually plug in somebody else's code. I therefore recommend that you begin each module, before the Thursday lab, by consulting your partners and planning what code you'd like to keep from the previous week—and what you might like to replace with someone else's code.
2. Each module includes a lab, the results of which will be part of your homework. You can think of the Thursday lab as a well-supervised "start the homework" session. Prepare for the lab ahead of time, and plan to do it with a partner. (For the first few labs, you'll be placed with a partner or partners chosen by the lab instructor. For later labs, you will choose.)
3. Each module lists 8 or 9 things you should plan to learn, or skills you should plan to demonstrate, while doing the homework. These are the module's *learning outcomes*. At the end of the lab, identify which parts of the homework

seem essential to the learning outcomes. If you're working on a team, those parts should either be pair programmed or should be split in such a way that all team members get to claim learning outcomes. (For example, if the team needs to implement a dozen VM instructions, each member could independently implement a half-dozen instructions, and all could credibly claim learning outcomes associated with those instructions.)

4. Not every piece of code you write is going to contribute equally to learning outcomes. For example, in the first module, it's essential that you learn the basic technique for implementing VM instructions, but once you've done something like an `add` instruction, you're not going to learn a lot more from doing subtraction, multiplication, and division. But they still have to be implemented. Code like this, which is necessary for the project but is not necessarily essential to learning outcomes, can be implemented either in partnership or by one or another partner independently. If your implementation choices permit it, you can even crowdsource such code to the whole class.

How will I get software? How will I submit?

Software I provide will be available in the private git repository https://gitlab.cs.tufts.edu/cs106-staff/student-2023s/. In order to be granted access, you'll need your department account to be enabled for Gitlab. It is sufficient to sign into `gitlab.cs.tufts.edu` using the web interface.

I encourage you to maintain your own code in a `git` repository, but because `git` is such a usability train wreck, I don't want to mandate it. So you'll submit from the department servers using the venerable `provide`.

As for other software, C and Standard ML are already installed on the department servers. I do recommend that you install these languages on a machine of your own, however—the development experience is so much better.

If you are using Emacs, I highly recommend the `magit` package. It actually makes `git` pleasant to use—an outcome I thought was impossible.

How will we stay in touch outside of class?

Asynchronously, we will be on Slack. I've created an instance at https://cs106spring2023.slack.com/.

How will grades be determined?

Grades will be determined by a point system. The system is complicated, but it's meant to be transparent: at all times, you can be in control of and aware of your grade. Because the system is unusual, you deserve to be aware of the rationale behind it:

1. I want everyone to do the project.
2. I want everyone to contribute to code review.
3. I want everyone to get an A.

In addition, I want to simulate real-life conditions: if you're implementing a programming language, you won't have an instructor to tell you if the code is right. In fact, I don't really care if your code is right—I care that you do the project *thoughtfully*.

To meet all of my goals for the course, I've designed a custom grading system. I believe it meets the goals, but it's not simple. The grading system is based on points, which you can earn in three categories:

- You earn *project* points by checking off learning outcomes with each homework. There are 12 homeworks with a total of 100 learning outcomes, distributed roughly equally (8 or 9 learning outcomes per homework). After you submit a homework, you'll have 24 hours to think over what you've learned and talk over the experience with your classmates. You'll then submit a *reflection* that claims whatever learning outcomes you think you've earned.
- You earn *participation* points by contributing to code reviews and to the workshop.
- You earn *depth* points by going deeper into a topic. For example, if you write a parser for a concrete syntax like C or Python, you'll earn 10 depth points. Many modules will have depth-point opportunities.

Project points and participation points are essential, but to earn an A or an A-plus, you also have to demonstrate depth. The number of points you earn determines your grade; each grade has a minimum requirement in each category:

Grade	Project	Participation	Depth
A+	97	97	10
A	92	92	5
A-	90	90	—
B+	87	87	—
B	83	83	—
B-	80	80	—
C	70	70	—

If you earn more points in one category than another, you can trade them as follows:

- You can sell one project point to buy four participation points.
- You can sell five participation points to buy one project point.
- You cannot buy or sell depth points.

The costs are set up to make the project worth 80% of your grade and participation worth 20% of your grade. You don't

actually have to execute any trades; my software will trade points automatically in a way that maximizes your grade.

How do I earn project points?

In each module, you earn project points from your Tuesday-night reflection, which in turn is supported by your Monday-night homework. Each module has a section labeled "learning outcomes." In that section you will find 8 or 9 bullet points, each describing a thing you are hoping to do or hoping to learn. Here are some examples:

- C code builds successfully with a Makefile or compile script.
- You can identify VM instructions you might use in a language other than Scheme.
- When you've written a translator, you can identify the invariants that distinguish the source and target representations.

Each learning outcome—that is, each bullet—earns one project point. You'll have the learning outcomes in mind when you craft the code you'll submit on Monday. Then after 24 hours, on Tuesday night, you'll submit a reflection that lists the outcomes you've learned. You'll support each learned outcome with a short sentence or two, sometimes just a pointer to your code. The deadlines are designed so that learning outcomes can be discussed in Tuesday's code review.

Project points for a module are earned based on the homework submitted for that module and on the corresponding Tuesday-night reflection. Unless they are so marked, a project point can't be carried forward to a future module.

There are a total of 100 opportunities to earn project points. If it looks like it might be hard to get to 92 points, I may add opportunities to earn an additional 2 or 3 project points.

Some points can be earned more quickly than others, but if you earn all the available points for the course, I guess that it might take you around an hour and twenty minutes per point. That's a *very* rough estimate; our data from CS 105 show that the time students report spending typically varies by a factor of 4. You are part of a self-selected group, so I am not expecting that much variance, but a factor of 2 would be unsurprising.

How do I earn participation points?

Participation points are earned for activities you do every week, like small-group code review; for activities you do occasionally, like serve on a code-review panel; or activities you do once, like present your work at a workshop. Each category earns points up to a fixed maximum, as listed in this table:

40	The workshop at the end of the course (once)
15	Presenter at plenary code reviews (occasionally)
13	Panelist at plenary code reviews (occasionally)
12	Participation in peer code reviews (weekly)
10	Code-review reflections on homework (weekly)
10	Material responses to code review (occasionally)

The algorithm for earning participation points is where things get complicated:

- The final workshop is worth 40 points, broken down as follows: 20 for your presentation, 10 for engagement with other people's presentations, and 10 for your final, post-workshop reflection. That's 8% of your overall grade.
- If you *present at a plenary code review*, you earn 10 participation points. If you present a second time, you earn an additional 5 points.
- If you *serve on a panel at a plenary code review,* you earn 3 participation points. If you serve a second time, you earn an additional 2 points, and if you serve a third time, 1 more point. A moderator may earn an additional point, and a panel recordkeeper may earn up to 2 additional points.
- If you are *among the first to volunteer* to present or to serve on a panel, you will earn bonus participation points, up to a total of 5 additional points.
- Every time you *review a peer's code* in a small-group Tuesday session, you earn 1 point, up to a total of 12.
- Every time you submit a homework, we ask you to reflect on *what you are proud of*, *what you want help with*, and *what you would like the instructor to review (and why)*. Answering these questions earns 1 point per homework, up to a total of 10.
- When you submit a homework, you may point to *code that you materially changed* in response to code review. Material changes may include fixes for problems that are identified in code review, but they can also include any other improvement that is motivated by specific feedback given in code review. Code doesn't have to be bad to get better.

  Identifying a material response to code review earns you 3 participation points. If you thank your reviewer by name, that person also earns 2 bonus points. (They must not be a contributor to the code that was reviewed.) You can earn up to 10 points this way. And if you are thanked by others for your effective reviews, you can earn up to 10 bonus points that way.

Everyone can be confident of earning full credit for participation. If I've done the arithmetic correctly, you can earn 100 points through mechanisms that are totally under your control: If you show up every week, take your turn at plenary code reviews, and generally do what's expected, you'll earn 60 points. You'll earn 40 points for a successful workshop. That's a total of 100, without any bonus points. If you do earn bonus points, either through

early volunteering or by having your peers acknowledge you for help that you give them in code reviews, they add to your total.

How do I earn depth points?

Everybody is doing the same project, but not everybody has the same interests. Depth points offer you the opportunity to earn a top grade by demonstrating depth in an area *that you choose*. Depth points will be available in multiple modules, and when possible, depth points will be available for interests that you identify in the pre-course survey.

Depth points are earned by completing depth goals, which might look like this:

- Design a concrete syntax for vScheme that resembles C or Python, write a parser for it, and integrate the parser into your translator [10 points].
- Optimize the VM's loading of literal values and explain how the optimization is consistent with the formal semantics of loading [2 points].
- Say what would constitute a *complete* set of VM instructions for implementing vScheme [1 point].

Unless a depth goal is marked with an expiration date, depth points can be earned by completing the goal any time before exams begin. Unless otherwise marked, a depth goal is *not* limited to the module in which it appears.

To earn an A or A+ you need 5 or 10 depth points, which you can earn by completing a few small goals or one bigger goal. The point rewards are meant to be tuned so that earning a depth point takes about an hour.

What if I can't deliver my work on time?

The project is very cumulative, and I recommend that you don't get behind. To make it easy to submit your homework on time, I've set things up so you get points *not* for complete, working code, but for your ability to articulate *what you learned* from writing the code. If you've built a thing and you understand how it's put together, but it doesn't quite work yet, *go ahead and submit it on time*. You can debug it later.

That said, I'm offering a limited late policy somewhat similar to the late policies for CS 40 and CS 105:

- Each student is automatically issued *four* "extension tokens." By expending an extension token, you get a 24-hour extension on *both* deadlines associated with a single module: that's the Monday homework and the Tuesday reflection.
- *At most **one** extension token* may be expended on any single assignment.
- A homework submitted late with an extension token earns full project points, but *the submitters earn no participation points for the Tuesday code review*. That's only fair: if your work isn't submitted, we can't review it—and I need Tuesday morning to look at the submissions and plan code review.

  If for some reason your Monday homework is on time but your Tuesday reflection is 24 hours late, you can earn full project points and full participation points—but it's a terrible use of an extension token, so please don't let this happen.
- *When you are out of tokens, late homework will no longer earn points,* but it can still be submitted for feedback, and it can be shown at code reviews.

What if I'm too sick to submit work?

If Covid spikes again, you may get too sick to work. And the project is very cumulative—what are you going to do? Just let me know. It's a small class, and I want everyone to get an A. I will work with you to make that possible, in a way that is fair to you and also fair to your classmates.

This policy applies to any significant illness, not only to Covid-19.

How will the course end?

In lieu of a final exam, you will present your work to your peers and to a jury of outside experts. Presentations will take place during a half-day workshop, which will be take place between 2:30pm and 6:00pm on Tuesday, May 9.[1] After the workshop you will have 48 hours to respond to the jury and to submit one final reflection summarizing your learning outcomes from the course and from the workshop.

Acknowledgements

Matthias Felleisen, who has many years of experience doing code reviews with students, explained to me how his code reviews are structured; I have borrowed several of his practices. Jacob Matthews suggested why code review might be important.

[1] This is our scheduled exam block, plus some time on either end.