Lesson 1

PROLOGUE

May 16, 2023

1. Administrivia
2. The Philosophy of Functional Programming
3. Types, Expressions, Values
4. Declarations

1 - Administrivia

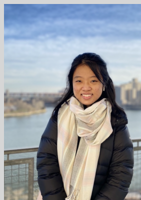
I'm an alum who graduated last year with a degree in computer science from CMU.

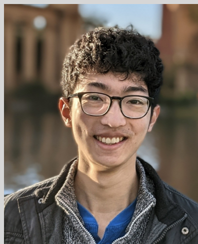
I TA'd 150 for three years while I was at CMU, and I currently work a full-time job as a software engineer at a security company called Semgrep, doing functional programming[1] for program analysis.

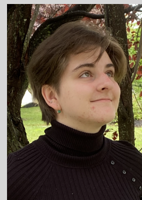
This means I am uniquely equipped as someone who works in the industry, to say that **functional programming is really useful**.

Note **I am not a professor**.

[1]Using a language very similar to the one you will learn in this class.





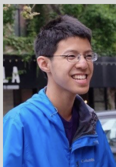


Nancy

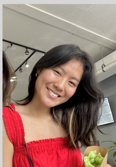
Brandon Wu

Sonya

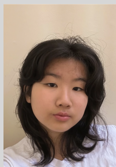
Kaz



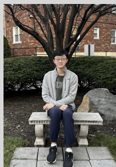
Deya



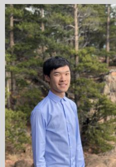
Caroline



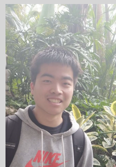
Stephen



Brandon



Michael



- Homework every week – due on Tuesdays or Thursdays (see the lecture schedule!)
- Turn in assignments via Gradescope
- Receive assignments via Canvas
- Piazza for any questions
- If you are not in all of { Gradescope, Piazza, Canvas } please let me know
- Slides, course policy, and other information at the course website here

The full summer schedule can be found here (click me!)

The grading policy for this semester will follow a *choose-your-own-adventure* format.

There are two tracks of grades that we will support, the "Lecture Track" and the "Homework Track".

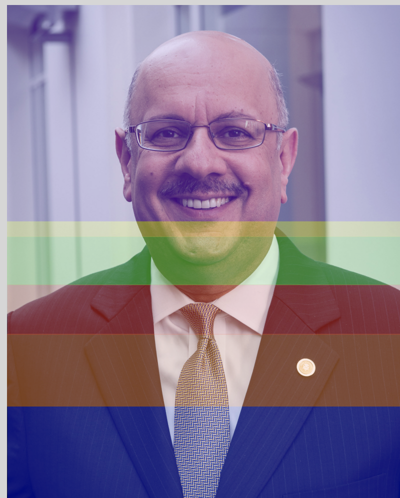
The Lecture Track is for students who would like to earn points via lecture attendance. Homework will count for 42% of the final grade, and lecture attendance will be 3%.

The Homework Track is for students who would rather not be graded for lecture attendance. Homework will instead count for 45%, and there will be no grading on lecture attendance.

Grading scheme selection will occur via Piazza after the first week.

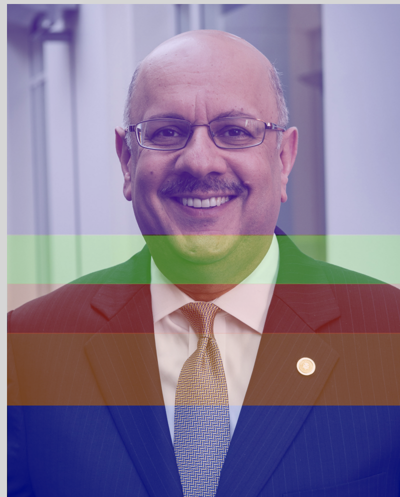
(visualized via % of Farnam)

- Homework: 42%
- Lecture Attendance: 3%
- Lab Attendance: 10%
- Midterm 1: 10%
- Midterm 2: 15%
- Final: 20%



(visualized via % of Farnam)

- Homework: 45%
- Lab Attendance: 10%
- Midterm 1: 10%
- Midterm 2: 15%
- Final: 20%



This semester, we are implementing a house system.[2]

The class has been divided into three labs of roughly 20 students, which each forms a "house". Each house will be eligible to earn points on the merits of:

- Asking good questions on Piazza
- Giving good answers on Piazza
- Answering questions during lecture

Twice a semester, the house with the most points will earn boba for their entire lab.

[2] Yes, like Harry Potter.

It is better to learn by exercising your mind actively, rather than passively absorbing information in a lecture setting.

To facilitate this, we will have active learning exercises during lecture. There will be a short (roughly 5 minute) quiz in the middle of every single lecture, that **will not count towards your grade**. Instead, it will simply be used by TAs to judge where your understanding may be lacking.

In particular, these quizzes are a whole-house effort! Every house will take the quiz together, and whichever house scores the highest cumulatively earn points for their house, and be crowned that lecture's victor, along with earning other prizes.

This means that winning the house competition is a whole-house effort! It's in your interest to discuss with others during these quizzes, and fill in gaps in yours and others' understanding.

Your health is important.

The things you will learn in this class are important, but remember that **it's just a class**. If you're sleeping 4 hours a night, skipping meals, or otherwise compromising your health for the sake of this class, **something is wrong**.

Things that are required:

- Sleeping
- Eating
- Socializing (for most people)
- Paying taxes

Things that are not required:

- Finishing 150 homework at the expense of these four things[3]

[3] Especially the last one.

Take care of yourself, and if you're experiencing difficulties, don't hesitate to reach out to myself or a TA you trust. For more severe cases, please go to http://www.cmu.edu/counseling/

My job is to teach you, not to ruin your life. If you're struggling, **talk to me**. Let's make a plan to help you succeed.

2 - The Philosophy of Functional Programming

What is functional programming?

Well, hold up. What is programming?

Programming is the act of **instructing a computer on how to achieve some kind of operation.**

Programming is **inherently a communicative act.**

Instructing is the key word. Good communication exists, so what is good programming?

Functional programming is **an improvement on our ability to communicate as programmers**.

Good programming should be **descriptive**.

Edgar Dijkstra: Go To Statement Considered Harmful

Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing

dynamic progress is only charac
call of the procedure we refer.
we can characterize the progres
textual indices, the length of
dynamic depth of procedure cal

```
template <typename T>
void goto_sort( T array[], size_t n ) {
        size_t i{1}

        first: T current{array[i]};
        size_t j{i};

        second: if ( array[j - 1] <= current ) {
                goto third;
        }

        array[j] = array[j - 1];

        if ( --j ) {
                goto second;
        }

        third: array[j] = current;

        if ( ++i != n ) {
                goto first;
        }
}
```

Good programming should be **modular**.

Logically distinct parts should be separated, for separate maintenance and reuse.

You should be able to think about a single area of a codebase without needing to concern yourself with unrelated logic.

Good programming should be **maintainable**.

Programs should be written with future maintainability in mind.

This goes hand-in-hand with descriptivity and modularity, but code which is written expressively, and has future expansion in mind, will be easiest to maintain.

Let's look at an example of non-functional code in Python.

```python
count = 0

def increment():
  global count
  count += 1

  return count
```

What does `increment()` return?

The answer: **it depends**. On the first call, it returns `1`, and on the second call, `2`, and so on.

This demonstrates what we call **state**.

Suppose you are a master chef at a 5-star restaurant.

An imperative program is like a fully crowded kitchen with no rules.

- Everyone uses the same ingredients and the same cookware.
- **Each cook is an individual actor that can mess with the others**, if care is not taken. The health of the kitchen depends on each individual chef.

A functional program is like a kitchen where each cook has their own working space.

- Everyone has their own pots, pans, and ingredients, and they only share things when they finish producing their individual parts.
- **Each cook only interacts when sharing finished results**. This means it is impossible for one cook's actions to mess up another's cooking.

Programming, Imperatively

Computation by **modifying the computer's state**

```
x := 2;
x + x
```

Programming, Functionally

Computation by **reduction of expressions to values**

```
2 + 2
```

In stateful programs, we use commands like `x := 2` to *change the world*, to be one where `x + x` is 4.

This doesn't stop another part of the program from changing that later!

In functional programs, we apply simplifying rules to expressions like `2 + 2`, to obtain the value of `4`.

These expressions are **disjoint**, in that evaluation of one expression is unrelated to the evaluation of another.

In stateful programs, understanding a program entails not only understanding what the code does, but knowing the entire history of the program up until that point.

Functional programming **avoids modification of state**.

Def A function is *pure* if it does not have any observable side effects, and always returns the same outputs, given the same inputs.

A large amount of problems in computer science are of a pure nature. This means that they give the same outputs for the same inputs. (For example, finding the shortest path through a graph, computing the nth prime number, or compressing a file)

An important motivation behind functional programming will end up being that we should prefer to solve pure problems with pure components. In other words, don't introduce state when it's not necessary!

Functional programming can be characterized by three theses, which will be a recurring theme throughout the semester.

1. Recursive Problems, Recursive Solutions
2. Programmatic Thinking is Mathematical Thinking
3. Types Guide Structure

In this class, **almost every single function you write will be recursive**.

Many problems in computer science lend themselves to a recursive formulation. These are naturally solved by recursive solutions.

We will see that lists, trees, and other important structures employ a simple recursive description.

Programs are simpler and more understandable when viewed in a mathematical lens.

Functional programming allows reasoning about programs and their subcomponents in the same way that you would reason about a mathematical expression.

Furthermore, mathematical analysis of code grants techniques to reason about things like time complexity, parallel time complexity, and program correctness. Functional programs are very amenable to proofs of correctness!

We're not just in the business of writing code, but *correct* code!

Types Guide Structure

Before, we described programming as an explanatory, communicative process.

We also stated how *descriptivity* and *maintainability* are key goals for good programming.

Functional programming places a great emphasis on *types*, which serve the purpose of documenting the purpose of code, and restricting the range of behaviors that a program is allowed to exhibit.

In this way, types guide the structure of a program, by providing clean interfaces for how different parts should interact, and what it should be allowed to do.

5-minute break!

3 - Types, Expressions, Values

In this class, we will be using a functional programming language called Standard ML (SML).

Mantra In Standard ML, **computation is evaluation**.

Evaluation of what, though? The most fundamental unit of an SML program is called an **expression**.

Def An **expression** is the building block of an SML program. These may or may not evaluate to a value.

Def A **value** is a **final answer**, that cannot be simplified further.

Examples of values include `2`, `"hi"`, and `true`.

Examples of expressions include `2`, `2 + 2`, and `4 * 5`

To evaluate an expression like `(2 + 3) * 4`, we apply simplifying rules. So we get:

$$
\begin{aligned}
\texttt{(2 + 3) * 4} &\Longrightarrow \texttt{5 * 4} \\
&\Longrightarrow \texttt{20}
\end{aligned}
$$

Def We use the $\Longrightarrow$ symbol to denote **stepping** (or **reducing**) of expressions, which means to simplify an expression by one step.

Def We use the $\Longrightarrow^*$ symbol to denote the application of the $\Longrightarrow$ relation an arbitrary number of times, usually until completion.

So the expression `5 * 4` *steps to* the expression `20`, and the expression `(2 + 3) * 4` $\Longrightarrow^*$ `20`.

We call the previous slide the **computation trace** of the expression `(2 + 3) * 4`.

The goal of a computation trace is to produce a value.

If we know that the expression `e` eventually reduces down to value `v`, we might say that `e` reduces to `v`, or write `e` $\hookrightarrow$ `v`. We then say that `e` is **valuable**.

So `(2 + 3) * 4` $\hookrightarrow$ `20`.

We said before that the goal of a computation trace is to produce a value, but not all expressions do!

What value does the expression `1 div 0` reduce to?

The answer: **there is no such value!** Division by zero is undefined, and in Standard ML, raises an exception.

Evaluating an expression has three possible behaviors:

- Reducing to a value
- Raising an exception
- Looping forever

In Standard ML, the string concatenation operator is `^`.

So we would say that `"hi" ^ "there"` $\Longrightarrow$ `"hithere"`.

But what does `"1" ^ 50` step to?

Some programming languages might try to make sense of this expression.

Standard ML will not.

Def A **type** is a specification of the behavior of a piece of code. It **predicts** what a program is allowed to do.

We write `e : t` to say that the expression `e` has type `t`, so we could write `1 + (2 * 3) : int`.

For instance, something with type `int` must produce a number, if it reduces to a value.

Similarly with something of type `string`.

What can we say about the runtime behavior of `"1" ^ 50`? It's not clear, so the expression `"1" ^ 50` **does not have a type**.

How does Standard ML know the type of an expression?

It follows **typing rules** to determine this. For instance:

Def The typing rule for + is: `e1 + e2 : int` if `e1 : int` and `e2 : int`

Take the expression `1 + (2 + 3)`.

Then, we know `1 + (2 + 3) : int` if `1 : int` and `2 + 3 : int`.
We know `1 : int`.
Then, we know `2 + 3 : int` if `2 : int` and `3 : int`.
We know `2 : int`.
We know `3 : int`.
So `1 + (2 + 3) : int`.

What about for `"1" ^ 50`?

Def The typing rule for `^` is: `e1 ^ e2 : string` if `e1 : string` and `e2 : string`

Take the expression `"1" ^ 50`.

Then, we know `"1" ^ 50 : string` if `"1" : string` and `50 : string`.
We know `"1" : string`.
However, it is not true that `50 : string`, because `50 : int`.

So `"1" ^ 50` does not have a type, and we say it is an **ill-typed expression**.

SML is a **statically typed** language, meaning that all typing rules are applied **before the program is ever run**.

Def We say that a piece of code or a program which obeys all the typing rules **type-checks**, or is **well-typed**.

Key **Ill-typed programs are not evaluated.**

4 - Declarations

We've so far discussed types and values, but we haven't introduced the machinery we need to actually work in a programming language! We need some way to declare variables and functions.

In SML, we can declare functions using the `fun` syntax:

```
fun double (n : int) : int = n + n
```

This comprises of a few parts:

- the `fun` keyword that signals the start of the declaration
- the name of the function (`double`)
- the arguments to the function, annotated with type (`n` and `int`)
- the return type of the function (`int`)
- the body of the function `n + n`

To use the function we just defined, we have to **apply** it. This is done via placing the function expression directly adjacent to the argument that it is meant to take in.

So for instance, instead of writing `double(2)` like we would in some programming languages, we would write `double 2`. This is known as **function application**.

So we would have that `double 2` $\Longrightarrow$ `4`.

Since `double` is a function which takes in an `int` and returns an `int`, we would refer to it as having the **function type** `int -> int`.

How do we know that `double : int -> int`, however? Is it just because we annotated it as taking in `int` and returning `int`?

That isn't true, however, because the following declaration fails to type-check, and thus will not be executed by Standard ML:

```
fun double (n : int) : string = n + n
```

In reality, SML is checking the type of `double` to make sure that it is consistent with the annotations that we state.

```
fun double (n : int) : int = n + n
```

To check the type of `double`, SML will take our word for what the input type of `n` is, here.

It will then try to produce a type for the body of the function, *given that* `n : int`. If that matches up with the return type as stated, then the function is well-typed.

In this case, we see that if `n : int`, then by our previous logic `n + n : int`, which matches the return type. When we change it to `string` however, that becomes different, so the declaration is rejected as ill-typed.

How do we know that `double 2` is well-typed?

Def The typing rule for function application is that `e1 e2 : t2` iff `e1 : t1 -> t2` and `e1 : t1`, for some types `t1`, `t2`.

In essence, this is saying applying a function only returns a type `t2` if the function has type `t1 -> t2`, and it is given an input of type `t1`.

In this case we know `double 2 : int` if `double : int -> int` and `2 : int`.
We know `double : int -> int`, because of our previous reasoning.
We also know that `2 : int`.

So `double 2 : int`, the return type of `double`.

Now we need to cover how to declare variables. The key word of interest here is `val`.

```
val favoriteCourseNumber : int = 150
```

It functions similarly to a function declaration, but there are now no arguments to type-annotate.

We will have more to say about variable declarations in the next lecture.

Thank you!