Welcome to 15-150! Today’s lecture was an overview that showed the highlights of everything you’re learning this semester, which also meant we didn’t show it in a whole lot of detail, so don’t panic if you didn’t understand a lot of the things I said. Starting tomorrow we’ll be going through the material at a much more reasonable pace. Today’s goal is just to see the main goals of the course and get a feel for what they mean and why they’re useful.

1 Functional Programming

In class, I tricked you into acting out a functional program to figure out how many of you had taken 15-122. This is our first course objective:

You will learn to write functional programs.

Imperative programming, which 34 of you studied last semester, is all about creating and altering state. Functional programming, by comparison, is all about studying transformations on data. As the name implies, these transformations are done by functions. Imperative programmers use things called functions as well, but I mean functions more in the sense of mathematical functions, like the factorial function:

\[
\text{factorial}(5) \equiv 120
\]

(I’ll get to what that symbol \( \equiv \) means later.)

We’ve used the factorial function to transform 5 into 120. This might sound like altering state, but it’s not: the number 5 still exists, which is good!

In a similar way, we can define a function that counts how many students in a set have taken 122:

\[
count(\{s_1,1,\ldots,s_{1,n}\},\ldots,\{s_m,1,\ldots,s_{m,n}\}) = \sum_{i=1}^{m} \sum_{j=1}^{n} \begin{cases} 
0 & s_{i,j} \text{ didn’t take 122} \\
1 & s_{i,j} \text{ took 122} 
\end{cases}
\]

*Based on drafts by Brandon Bohrer, Carlo Angiuli and others
In a functional program, the students themselves don’t change when we count how many of you took 122. If this had been an imperative program, we might have updated some variables in the process and would have to be careful if we wanted to make sure you weren’t destroyed during the counting process.

In syntax that’s more like what you would actually type into the computer, the function we ran on you was

\[
\text{count} : \text{students} \rightarrow \text{int}
\]

The word \textit{count} is the name of a function that we’re going to define. The phrase \textit{students} \rightarrow \textit{int} is a \textit{type}, which is a contract on the behavior of this function. The type \textit{students} \rightarrow \textit{int} should be read “\textit{count} is a function that transforms a collection of students into integer.”

The type doesn’t say in detail what the program \textit{does}, it just says it transforms students into an integer. If we want to indicate how it does this, we can use a comment:

\[
(* \text{REQUIRES: true} *) \quad (* \text{ENSURES: returns the number of students who took 122} *)
\]

\[
\text{count} : \text{students} \rightarrow \text{int}
\]

Depending on the situation, sometimes our specifications will be very formal and mathematical, sometimes they will be more informal. For those of you who took 15-122, it’s worth noting that unlike in C0, which actually runs code that you put in a @requires or @ensures comment, these are just regular comments, so you can include plain English in your comments when that’s the best way to document your functions.

Before we figure out how to implement \textit{count}, we need to explain how \textit{students} are represented. Well, you’re organized into rows, so let’s assume that the collection of all students is a \textit{sequence} of rows, which we can write in SML notation using a \textit{type declaration}:

\[
\text{type row} = \text{int sequence} \\
\text{type students} = \text{row sequence}
\]

Likewise, a row is a sequence of integers, which are either 0 (if you didn’t take 122) or 1 (if you did). (In general, I don’t like reducing people to numbers, but it’s convenient for this example.)

The notion of a sequence is something we will spend a lot of time on this semester; for now, you can think of it as an abstract, ordered collection of things, just like you students in the back row there. When I say it’s an abstract collection, this means that when we program with sequences, we don’t worry about the details of how they are implemented, but instead we reason from an abstract specification telling us how sequences are supposed to behave. Programming in the abstract is perhaps the single most important skill for writing large programs.

First, let’s assume we have a function
(* ENSURES: sum r is the sum of the numbers in r *)
sum : row -> int

How do we use these to count the whole class? We define a function

fun count (s : students) : int = ...

This declaration introduces a function with the above type, students -> int. How did we implement this function when we acted it out?

1. First sum each row
2. Then sum the column at the end, which after all is just another sequence of integers.

Here's how this looks in ML:

fun count (s : students) : int = sum (map sum s)

In this code, map is an operation that applies a transformation to a sequence, producing another sequence. We (1) map sum over the rows, to transform them into the sums of their rows, then (2) sum over all the rows.

**Computing by calculation**  An important idea that we will talk about this semester is computing by calculation. As an example, we're going to work through the computation you did at the start of lecture, except we're going to use a smaller example.

Suppose the class looks like

row 1: yes no yes
row 2: no yes yes

We can represent this as follows:

val row1 : row = seq [1 , 0 , 1]  
val row2 : row = seq [0 , 1 , 1]  
val classroom : students = seq [row1 , row2]

Here the notation seq[1, 0, 1] means we're creating a sequence with three elements: the numbers 1, 0, and 1 in that order. The notation val row1 : row = ... means we're defining a new value named row1, which has type row. For now we will always write the types of values for the sake of clarity, and expect you to do the same. In a week or two we will talk about when and why you can leave types out.

To count the whole classroom, we apply the function count to the classroom, as in

count classroom
This is an expression, or a program phrase. An expression, simply put, is any piece of code that can be computed.

We run a program by computing by calculation, just like in high-school algebra. To evaluate the expression count classroom, we can calculate as follows:

\[
\text{count classroom} \\
\quad = \text{count (seq [row1, row2])} \\
\quad = \text{sum (map sum (seq [row1, row2]))} \\
\quad = \text{sum (seq [sum row1, sum row2])} \\
\quad = \text{sum (seq [2, 2])} \\
\quad = 4
\]

Whenever I calculate the value of a program, you’ll see me use this symbol \( \equiv \) (or \( == \) in ASCII). This is a notion of equivalence for programs that we call extensional equivalence. Once I start talking about proofs, you’ll hear a lot more about extensional equivalence - the idea is that when we compute by calculation, it’s almost surprising how much we can say about the correctness of a program just by talking about this notion of which programs are equal to each other. This is one reason why it’s so much easier to do a rigorous correctness proof in a functional language than in an imperative language.

At the end of my computation I have the numeral 4, which is a value—the final result of a computation.

This example has illustrated the first goal of the course: you will learn to write functional programs.

## 2 Parallelism

Earlier, we took a long time trying to figure out how many of you had taken 122, because we had to go through each row in order. Maybe it would go faster if we could compute multiple rows at the same time. In fact, there isn’t really any reason we can’t. This is because the calculation we’re doing on each row is just a mathematical transformation and can’t affect the calculation on any other row. If we were doing this imperatively, we’d have to worry about changing values that another calculation needed.

This is the essence of parallelism. By parallelism, I simply mean “the ability to do many things at once”. Why is parallelism important? First, most computers these days are multi-core: they have many processors, which means they can do many things at once. Second, lots of processing is done by clusters: a bunch of computers working together. This is how Google processes the Internet. So the challenge is to take advantage of these computational resources: how do you write a program in such a way that there are many things that can be done at once?

One of the great things about functional programming is that, to make use of parallelism, we don’t really have to change the code we wrote earlier at all! We just need to use
a data structure that is implemented in a way that supports parallelism (as sequences are). Once you have specified a program in such a form, the mapping to concrete hardware (e.g. the two processors in your laptop or the undisclosed number of computers in a Google data center) is taken care of by the compiler. You think about the things you want done, the compiler thinks about the order in which to do them, and the result is still well-defined.

This brings us to the second objective of the course, which is really just a modification of the first.

You will learn to write parallel functional programs.

3 Complexity Analysis

The next goal of the course is:

You will learn to analyze the sequential and parallel running time of your programs.

Here are the kinds of questions we usually ask:

- How long does it take to count the class, if only one person can add at a time? This corresponds to execution on a single-processor machine, and is called the work of the algorithm.
  
  Answer: Well, there is one addition per student, plus the addition “down the side” of the classroom, but that row certainly has fewer students than the class as a whole. So the number of additions is roughly proportional to the number of students in the class.

  This is called the work of the computation.

- How long does it take to count the class, if infinitely many people can add at a time? Answer: We still have to wait for (a) the longest row to do its addition and (b) the “row” at the side to add up at the end. Thus the answer is roughly proportional to the length of the longest row plus the number of rows.

  This is called the critical path, because it is the limiting factor, even with as many processors as you need. It is determined by the data dependencies of the computation—e.g. you can’t add yourself to the running total until you know the running total. The length of the critical path is called the span of the algorithm.

- How long does it take to count if we have $p$ processors? We’ll get to this soon.

  Intuitively: try to divide the work $W$ evenly over $p$ processors, but you can never do better than the span $S$. 

5
Asymptotic complexity analysis allows you to predict how long it will take to run your code on really big inputs, without actually running it on them. It is one of the main tools used to choose between different algorithms for the same problem (sometimes the constant factors matter more, but usually it’s the behavior when the inputs get very large that matters).

4 Sum with a better span

Now that we know about work and span, let’s analyze the process of adding across a row. The work is the number of students in the row, because you need to add that many numbers. The span is the same: you wait for the sum of the people to the left of you before you do anything.

What if a row was really big?

If a row was really big, we could divide into pairs and add numbers for the pairs, then have someone from each pair add their numbers with someone from another pair, and so on...
To make it easier to follow the example, we’ll add up the sequence seq \( \{1, 2, 3, 4, 5, 6, 7, 8\} \) (this might be the column that results from counting across 8 rows). To visualize this process, we can draw the computation as a tree of additions:

```
+ 
 / \ 
/ \ 
/ \ 
+ +  
/ \ / \ 
/ \ / \ 
/ \ / \ 
+ + + +  
/ \ / \ / \ 
/ \ / \ / \ 
/ \ / \ / \ 
1 2 3 4 5 6 7 8
```

What is the work? There are still 7 additions that you need to do. What about the span? If we have enough processors, we can do it like this:
Step 1:

```
+  
/ \
/   \
+  +
/ \ /\  
+  +  +  
/ / \ /  
+(3) 7 11 15  
/ \ /  
1 2 3 4 5 6 7 8  
```

Step 2:

```
+  
/ \
/   \
/     
+  +  +  
/ /  
+  +  +  
/ /  
+(3) 7 11 15  
/ /  
1 2 3 4 5 6 7 8  
```

Step 3:

```
+  
/ \
/   \
/     
/       
+  +  +  
/ /  
+  +  +  
/ /  
+(3) 7 11 15  
/ /  
1 2 3 4 5 6 7 8  
```

8
3 steps for 8 items: the span is logarithmic in the number of students in the row. This is much better than linear when things get large.

This way of summing up the tree in another common pattern of computation, and is represented by a function reduce, which breaks a sequence up into a tree and then places a binary operator (in this case +) at each node in the tree.

We can implement sum using reduce. Since this implementation of sum sums the numbers up in a tree shape, we'll name it sumtree, as opposed to just sum for the version we acted out at the beginning of class.

\[
\text{fun sumtree } (s : \text{int sequence}) = \text{reduce add 0 s}
\]

5 Reasoning

The fourth goal of the course is

You will learn to reason mathematically about the correctness of functional programs.

For example, if we ask the question “is adding up the numbers one by one (we’ll call this \text{sum}) the same as adding them up using a tree (which we will call \text{sumtree})?”, we can answer it with a theorem:

\textbf{Theorem 1.} For all rows \(r\), \text{sum } r and \text{sumtree } r compute to the same integer.

Soon, you’ll learn techniques for proving this theorem formally. For now, let’s get some intuition: why should this be true? Returning to the example of \text{seq}[1,2,3,4,5,6,7,8], we see that \text{sum} computes

\[1 + (2 + (3 + (4 + (5 + (6 + (7 + (8 + 0)))))))\]

whereas \text{sumtree} computes

\[((1 + 2) + (3 + 4)) + ((5 + 6) + (7 + 8))\]

Thus, the basic reason is that \text{addition is associative} \((x + y) + z = x + (y + z)\), with \text{unit 0} \((x + 0 = x)\).

More formally, we can do a proof by \text{structural induction on the sequence } r. Don’t worry if you don’t understand this term yet; you will by the end of the week.

Why do you need to be able to reason about your code like this? For one thing, this will help you write better software. Thinking through invariants this way will make it easier to write code. For example, like we saw with \text{sum} and \text{sumtree}, it’s often the case that there is a simple, obviously correct, but inefficient algorithm for a problem; and also a more complex algorithm that’s more efficient. Proving them equivalent \text{mathematically justifies your program optimization}. You could do this through intuition and testing, but those aren’t perfect. This way, you know your code is correct.
It’s also possible that someday you might have to write *safety-critical systems*, where you really want to know that your program doesn’t make a mistake. For example, airplanes these days are basically just flying computer programs. Most of us want the code running on airplanes to be correct, so it would make us feel better if we knew that someone had proven that code correct. The techniques used in proving this kind of code correct are a bit out of scope for an introductory course, but the point is that the techniques you’re learning now are the foundation for proofs that can actually save lives.

## 6 Course Overview

To summarize, you will learn to:

- Write parallel functional programs
- Analyze their sequential and parallel time complexity
- Reason mathematically about their correctness

The thesis of the course is that functional programming is a good way to do all of this: it makes it easy to express programs (cleanly, elegantly), analyze their time complexity, and verify their correctness. And moreover that this all applies to parallel programs: you can easily specify work that can be done in parallel, and analyze a program’s parallel complexity, and prove correctness about parallel executions (indeed, in this course, it will (almost) always have the same result as a sequential execution!).

### 6.1 Logistics

See the course website ([http://www.cs.cmu.edu/~15150/policy.html](http://www.cs.cmu.edu/~15150/policy.html)) for the course policies we covered in class. Make sure you have an account on Piazza. We’ll be going over the rest of the course infrastructure in lab on Wednesday.