Lesson 22

FINALE

August 8, 2023

Lesson 22

FINALE

August 8, 2023

Lesson Plan

λ

1. Rewind
2. Course Themes
3. Saying Goodbye

1 - Rewind

It's been a long semester.

12 weeks have soared by. We've made it through fire alarms, compiler bugs, and Andrew machine outages, and although it's been tough, we made it through nonetheless. You've survived homeworks, you've survived midterms, and now, we're nearing the end.

Now, it's time to review everything that we've learned. In the style of Michael Erdmann, I'm going to do so by going backwards – by starting at the end, and going back to the beginning.

Let's play back everything that we've learned.

Lesson 21

PROGRAM ANALYSIS

August 3, 2023

Programs are inherently recursive.

Expressing something which has as much rich, semantic information as a program can be as simple as a recursive datatype declaration in SML. Determining something about a program can be as simple as recursive function on a tree-like structure.

Program analysis is the final frontier of use cases for functional programming.

Lesson We can make a real difference, by using clean foundations and principled methods to achieve **real impact.**

Lesson 20
COMPILERS

August 1, 2023

Functional programming was made for compilers.[1]

The same tools that we use every day, the same magic that powers the principles of computer science that you study, are not so magic after all. Pure functions, safe code, and tree transformations are simple principles that you've learned over the course of the semester, but come to a head in one of the most fundamental applications in computer science.

Safety, elegance, expressivity. When writing a compiler, that's what you need.

Lesson Nothing can't be understood, if you can break it down to its most fundamental principles. The same foundations which make up a simple `treesum` function run in the DNA of the most complicated systems in the world.

[1] Technically, it was made for AI. But that's a different, longer story.

Lesson 19

IMPERATIVE PROGRAMMING

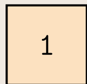
July 27, 2023

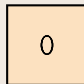
Eventually, we learned that functional programming doesn't need to be such an absolute concept!

We learned about **ref cells**, which involve a type `'a ref` of mutable boxes which store values of type `'a`. We learned about the following primitives:

```
val ref  : 'a -> 'a ref
val !    : 'a ref -> 'a
val :=   : 'a ref * 'a -> unit
```

which let us create, access, and manipulate boxes within our store:





r1 r2 r3

Mutability introduces many footguns when it comes to comprehending programs and debugging code, but it's all about how you use it. We can make mutability work for us, so long as we **only opt-in to it**, and we respect our principles of safety, readability, and elegance.

We are not beholden to certain ideas, just because we think that we should be. We invented abstractions to make them work for us, not the other way around.

Lesson Mutability is not so bad, so long as you have the **choice**.

Lesson 18

LAZY PROGRAMMING

July 25, 2023

Lambdas allow us to suspend computations in the form of a **thunk**, or a value of type `unit -> t`.

This offers us fine-grained control over exactly when and where computations happen. We can then use these to write **lazy** programs, which delay computations until they absolutely need to happen.

We saw this through the `'a stream` datatype, which defines a type of **maximally lazy** data structures, that do not compute any elements until they are explicitly **expose**d.

```
datatype 'a stream = Stream of unit -> 'a front
and 'a front = Nil | Cons of 'a * 'a stream
```

We can easily define **coinductive** streams via recursive functions without base cases, as in the case of the stream of natural numbers:

```
fun natsFrom n = Cons (n, fn () => natsFrom (n + 1))
val nats = natsFrom 0
```

Finite, infinite, we can express all of these ideas within Standard ML, via this simple idea. It all just comes out of mindful weaponization of our understanding of evaluation order.

Lesson Repeated application of simple ideas can lead to something great.

Lesson 17

SEQUENCES

July 20, 2023

Sequences are a signature for parallel-friendly, abstract data structures which are better for **bulk operations on data**.

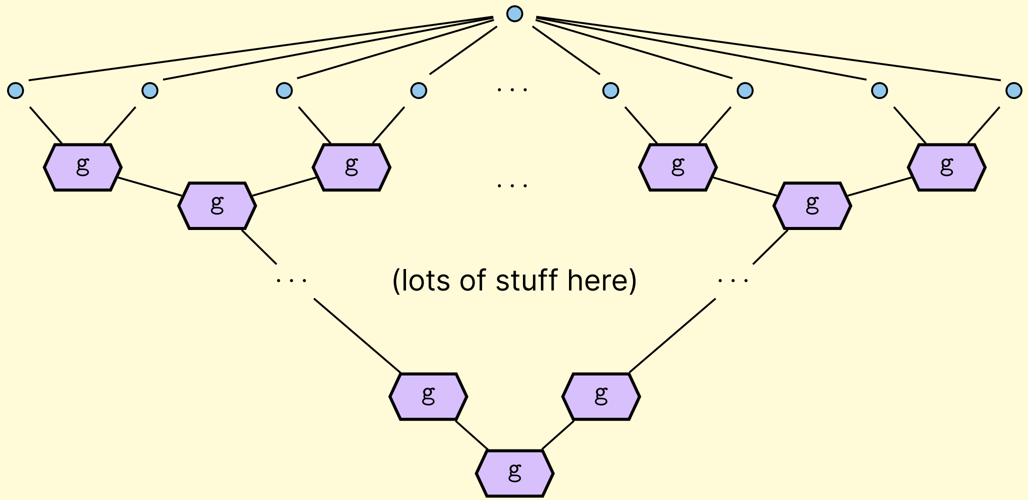
As an abstract type, we render them symbolically with the notation

$$\langle x_1, x_2, ..., x_n \rangle$$

and think of them conceptually as immutable arrays, where we can act on each element individually.

We saw that while they offer mostly the same operations as lists, they fit different use cases, in particular for when we are working with large collections of data, and don't need to perform any sequential operations.

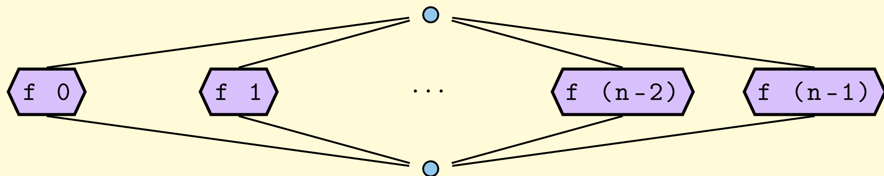
We also saw that, given a simple mathematical idea of associativity, we could present a brand new way to implement something like `foldl`, leading to `reduce` via a divide-and-conquer approach!


. . .
g
g
g
g
g
g
. . .
. . .
(lots of stuff here)
. . .
g
g
g

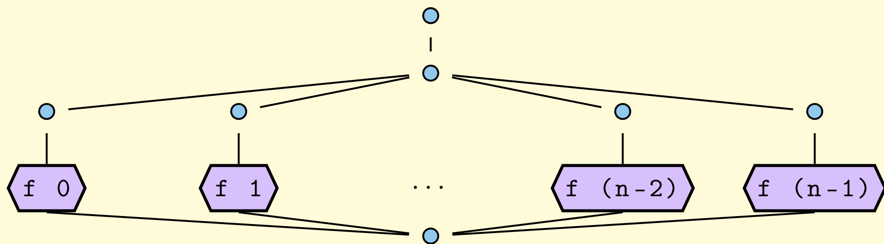
We also saw that we had this idea of **cost graphs**, which represent the cost of some computation by composing other cost graphs either parallel or sequentially.

Seen in this way, we can recover the cost of any higher-order function on sequences by merely composing these cost graphs together.

For instance, take the cost graph for `tabulate`:



From this cost graph, we can derive the cost graph of the expression `Seq.tabulate (fn i => f (Seq.nth (S, i))) (Seq.length S)` as:



This follows from simply "gluing together" the cost graphs of the `f` function, and the constant cost of `nth`, replacing them for the `f` nodes in the cost graph of `tabulate`, as well as adding the constant cost from `length`.

Lesson Functional programs are parallel-friendly, and can admit easy composable bulk operations.

Lesson 16

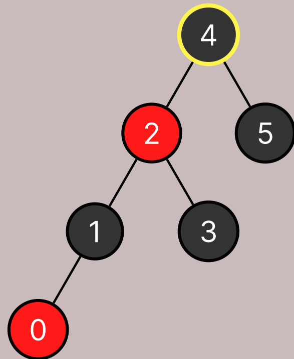
RED-BLACK TREES

July 13, 2023

We learned about **red-black trees**, a **self-balancing binary tree** with three invariants:

- the tree is a binary search tree
- the number of black nodes on any path to an `Empty` node is the same (black height invariant)
- a child of a red node must be black (red child invariant)

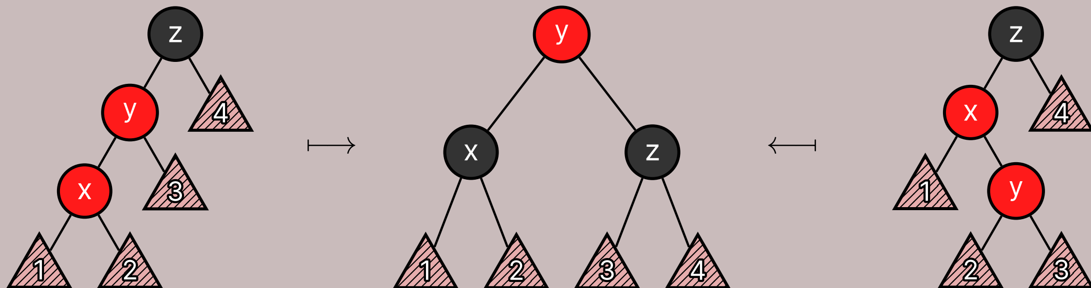
We saw that by a strategy of briefly breaking, and then restoring, our invariants, we could ensure that we always produced valid red-black trees through insertion.



Black height invariant: ✓

Red children invariant: ✓

This was carefully witnessed via our usage of **rotations** to restore our invariants locally, but by pushing inconsistencies to our recursive callers.



Lesson Abstract types and enforced invariants can help us achieve safer and more intuitive code. Put simply, knowing less is sometimes more.

Lesson 15
FUNCTORS

July 11, 2023

We learned about **functors** as maps from modules to modules, which allow us to neatly mix and match different parts of our codebase together.

In particular, we used it in combination with **typeclasses**, which allow us to attach values to particular types, for instance when choosing instances of a comparison function to go with a type.

This lets us modularize our code on other code, in a more powerful way than HOFs!

```
signature ORD =
  sig
    type t
    val compare : t * t -> order
  end
```

```
structure IntOrd =
  struct
    type t
    val compare = Int.compare
  end
```

We successfully used this to define a generic dictionary functor, which could be used to define any kind of dictionary library. All we needed to do was provide an instance of `ORD`, and the functor could automate away the rest of the logic for us.

```
functor MkDict (Key : ORD) :> POLY_DICT =
struct
  structure Key = Key

  type 'a t = (Key.t * 'a) tree

  val empty = Empty

  (* ... *)
end
```

Lesson Good software should be composable in terms of other software.

Lesson 14

STRUCTURES AND SIGNATURES

July 6, 2023

In this lecture we learned about **modules**, which allow us to organize and separate our code into distinct **namespaces**.

Most powerfully, we could use **signatures** to specify the types and definitions that we wanted exported from any given module, meaning that interfaces between software components can be typed and verified.

```
signature INTSET =
  sig
    type t

    val empty : t
    val insert : int -> t -> t
    val remove : int -> t -> t
    val mem : int -> t -> bool
  end
```

With **opaque ascription**, we can hide the definition of certain types, causing abstract types to be unknown to a user of a given library.

This gives us enormous power when it comes to maintaining invariants and making software conceptually simpler to deal with.

Lesson Software is easier to understand, easier to work with, and better overall when you can separate your interfaces cleanly.

Lesson 13

REGULAR EXPRESSIONS

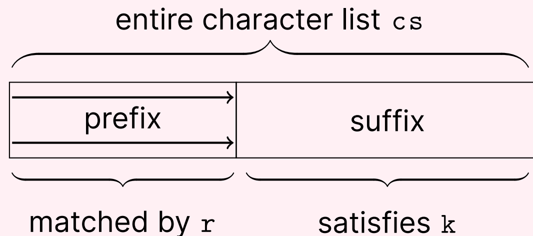
July 3, 2023

Regular expressions are a recursive datatype representing a mathematical object that corresponds to a certain set of strings.
Via simple operators, we can mix and match regular expressions to denote certain **languages** of interest:

Construct	Language matched
$L(c)$	$\{c\}$
$L(0)$	$\{\}$
$L(1)$	$\{\epsilon\}$
$L(r_1 + r_2)$	$L(r_1) \cup L(r_2)$
$L(r_1 r_2)$	$\{s_1 s_2 \mid s_1 \in L(r_1),\ s_2 \in L(r_2)\}$
$L(r^*)$	$\{s_1 ... s_n \mid$ for $n \geq 0$, when $\forall i, s_i \in L(r)\}$

In SML, this gets even more interesting with the implementation of the `match` matcher, which recursively decomposes on a regular expression to match strings in the language, given a regular expression `r` and a "suffix continuation" `k`.

We saw that we could reason about the matcher in terms of a kind of **proof by picture**, by thinking about the picture of matching a prefix and suffix that satisfy the specification.



Lesson Reasoning by specification or intuition is significantly more powerful than reasoning via stepping through code itself.

Lesson 12
EXCEPTIONS

June 22, 2023

We learned how to define **exceptions**, which are constructors of the **extensible type** `exn`, which can have arbitrarily many constructors added to it during runtime.

```
datatype exn = Match | Bind | Div | Fail of string | ...
```

We saw how we could use exceptions as escape hatches during exceptional cases for functions, as well as ways to jump to **handlers** that can then continue on with the program.

This manifested in **exception-handling style**, which resembles CPS, but replaces a failure continuation with exceptions.

```
exception NotFound

fun searchEHS p Empty = raise NotFound
  | searchEHS p (Node (L, x, R)) =
      if p x then
        x
      else
        (searchEHS p L) handle NotFound => searchEHS p R
```

Lesson It's OK to take quick, less-maintainable shortcuts, so long as you are judicious with their usage. Put simply, there's an **exception** to every rule.

Lesson 11

CONTINUATION-PASSING STYLE

June 20, 2023

We derive **continuation-passing style** by just considering what happens when we pass the result of a recursive call forward via piping into a lambda. So we might translate it as:

```
fun treesum Empty = 0
  | treesum (Node (L, x, R)) =
      treesum L + x + treesum R
```

First we replace all recursive calls with a placeholder variable.

```
fun treesum Empty = 0
  | treesum (Node (L, x, R)) =
      resL + x + resR
```

Then we obtain the placeholder variables via just making recursive calls, and piping into an explicit lambda.

```
fun treesum Empty = 0
  | treesum (Node (L, x, R)) =
      treesum L |> (fn resL =>
      treesum R |> (fn resR =>
      resL + x + resR))
```

Observe that this is just extensionally equivalent to our original `treesum` function.

This is not yet CPS, but what happens if we, instead of piping into a lambda, pass the lambda into the function itself?

Now, we derive CPS from very simple ideas! All we need to do is to make our recursive calls explicit.

Lesson Complicated things can be made simple once you have an algorithm for it.

```
fun treesumCPS Empty k = 0
  | treesumCPS (Node (L, x, R)) k =
      treesumCPS L   (fn resL =>
      treesumCPS R   (fn resR =>
      resL + x + resR))
```

```
fun treesumCPS Empty k = 0 |> k
  | treesumCPS (Node (L, x, R)) k =
      treesumCPS L (fn resL =>
      treesumCPS R (fn resR =>
      resL + x + resR |> k))
```

15-150: PRINCIPLES OF FUNCTIONAL PROGRAMMING - SUMMER 2023
Lesson 10
COMBINATORS AND STAGING
June 16, 2023

As a precursor to our eventual knowledge of streams and laziness, we found that we could control the evaluation of SML code via being particular about where it appeared relative to a given function's arguments.
So that this function:

```
fun foo x y =
  horrible_computation x + y
```

can instead be staged as such:

```
fun foo x   =
  let
    val res = horrible_computation x
  in
    fn y => res + y
  end
```

We then learned about piping operators like `|>`, which represent an ultimate form of higher-order programming, that lets us manipulate the format of any code to our liking.

```
remove (wait 2 (insert trayOfMozzarellaSticks (heat oven 400)))
```

```
heat oven 400
|> insert trayOfMozzarellaSticks
|> wait 2
|> remove
```

Lesson Pretty privilege is OK when it applies to code.

June 13, 2023

Here, we learned about **currying**, which is simply a function which takes in "multiple arguments", by returning a function which takes the rest of the arguments.

```
(* add : int * int -> int *)
fun add (x, y) = x + y

(* addC : int -> int -> int *)
val addC = fn x => fn y =>
(* syntactic sugar! *)
fun addC x y = x + y
```

We saw that, just by adjusting our perspective, something as simple as functions being able to return something of function type would have a massive influence on our style of programming.

We also saw the role that higher-order functions have in terms of being able to capture the very design patterns of our code.

So then, seen in this light, functions like `sum` and `concat` have the same "DNA" – the information content of their code follows the same pattern.

```
fun sum [] = 0
  | sum (x::xs) = x + sum xs

fun concat [] = ""
  | concat (x::xs) = x ^ concat xs
```

Higher-order functions are an essential tool in functional programming that are pivotal to many of the things we studied later.

HOFs ultimately ended up opening the first conceptual door to much greater things, such as `match`, such as CPS, and such as many of the functions we could write on streams and sequences. Through it, we see that great things come out of simply thinking of functions as values.

Lesson Writing code is good. Writing code which writes code is better.

Lesson 8
POLYMORPHISM

June 8, 2023

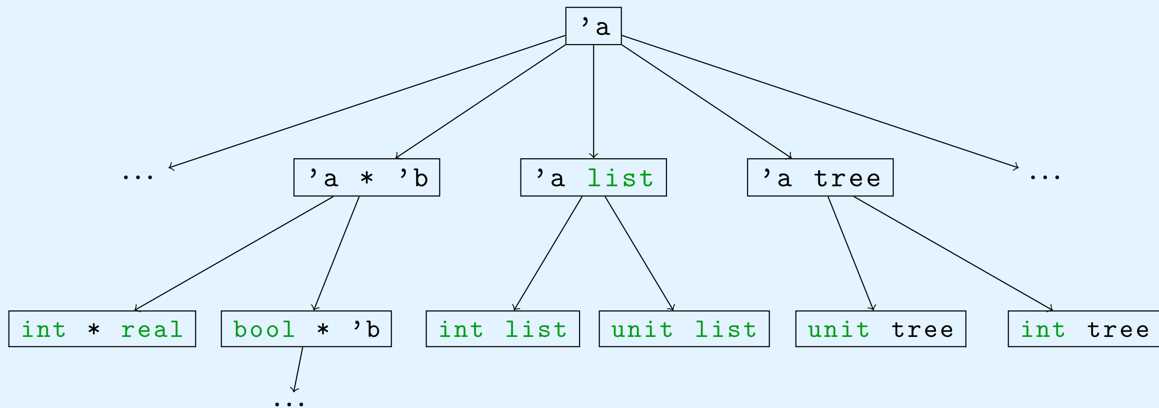
Before we could learn about higher-order functions, however, we needed to have a more expressive type system, to talk about functions of potentially arbitrary type.

Staging[2] that lecture was the idea of **polymorphism**, which expanded our vocabulary of types to include **type variables** like `'a`, `'b`, and `'c`, which allowed us to talk about types which may be instantiated at more specific type.

This ended up giving us a vast amount of flexibility, by allowing us to write generic functions that can be used a type-safe way, by merely varying the type with the context of its use.

We also saw that we could execute **typing traces** by collecting constraints on arguments to functions, which are given an initial, unrestricted type variable, and then coming up with its **most general type**.

[2] haha



Lesson More advanced type structure leads to concrete benefits in code. In other words, **types guide structure**.

Lesson 7
SORTING AND PARALLELISM

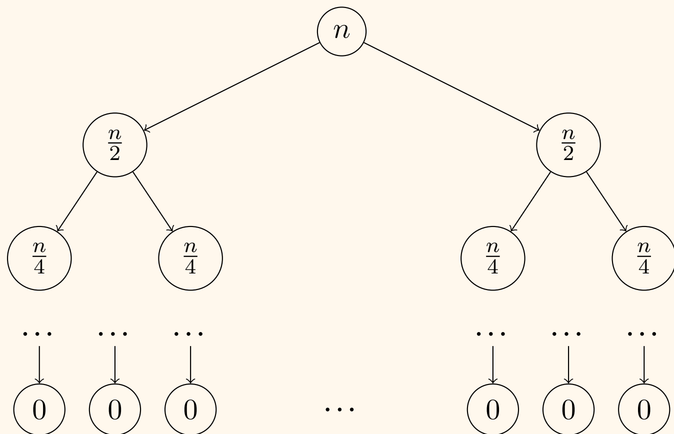
June 6, 2023

Back at this point in the course, we were more concerned with the mathematics and formal parts of analyzing our code.

We learned about the **tree method**, which allows us to solve recurrences that make more than one "recursive call" per level. We think of the tree method as simply inducing a "call tree" which denotes all of the calls being made by the recursive function, annotated with the size of the input and the nonrecursive work at each node.

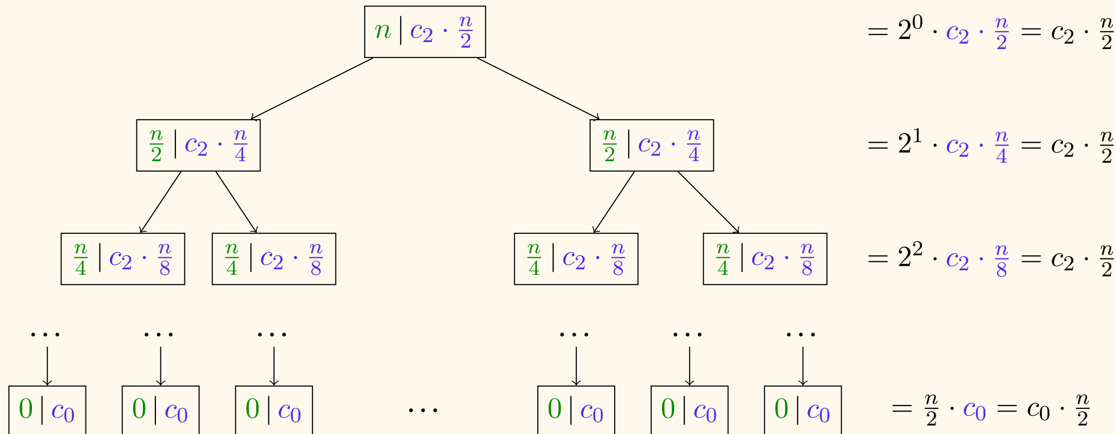
This ended up being a pivotal tool in analyzing such recursive functions, which came out of simply thinking about how we would sum the nonrecursive work at each level of the call tree, and then applying some simple mathematical facts.

For instance, we might draw a call tree labeled with the size of each node for the recurrence where eachh $W(n)$ term expands to contain $2W(n)$:



Or, for a specific example, we might also do this for the `inord` function:

$$W_{\texttt{inord}}(n) = c_1 + W_{\texttt{@}}(\frac{n}{2}) + 2 \cdot W_{\texttt{inord}}(\frac{n}{2})$$



We then learned about `span`, the idea of the *parallel cost* of our code, where we assume that we have an infinite number of processors.

Due to intrinsic **data dependencies** in code, having infinitely many processors doesn't necessarily solve all our problems, but it means that we can achieve a smaller cost bound for certain kinds of problems. This is because we take the **max** over certain parallelizable operations, rather than sequentially taking the **sum**.

We saw this in a use case for **merge sort**, where we could achieve a $O(\log n)$ span via parallelism, which also admitted an extremely simple implementation.

```
fun split ([] : int list) : int list * int list = []
  | split [x] = [x]
  | split (x::y::xs) =
      let
        val (A, B) = split xs
      in
        (x::A, y::B)
      end

fun merge ([] : int list, R : int list) : int list = R
  | merge (L, []) = L
  | merge (x::xs, y::ys) =
      if x < y then
        x :: merge (xs, y::ys)
      else
        y :: merge (x::xs, ys)

fun msort ([] : int list) : int list = []
  | msort [x] = [x]
  | msort L =
      let
        val (A, B) = split L
      in
        merge (msort A, msort B)
      end
```

Lesson Complicated things admit a simple recursive implementation, which also gives way to a simple recursive mathematical analysis.

Lesson 6

ASYMPTOTIC ANALYSIS

June 1, 2023

But before we could learn about parallel complexity, we learned about generally ascertaining the mathematical run-time of some function.

We learned how to do this by writing a **recurrence** for the abstract units of cost incurred by a given recursive function. We generally had to write our recurrences in terms of some notion of size of the input. For instance, the `treesum` function:

```
fun treesum Empty = 0
  | treesum (Node (L, x, R)) = treesum L + x + treesum R
```

This ends up producing the recurrence, in terms of the number of nodes of the tree n,

$$W_{\texttt{treesum}}(0) = c_0$$
$$W_{\texttt{treesum}}(n) = W_{\texttt{treesum}}(n_l) + c_1 + W_{\texttt{treesum}}(n_r)$$

where n_l and n_r are the number of nodes in the left and right subtrees, respectively.

Lesson Mathematical analysis can make even the most convoluted of things understandable.

Lesson 5
TREES

May 30, 2023

Here, we learned about **trees** and other custom datatype declarations, called **algebraic datatypes**.

We saw that structural induction could be carried out on any arbitrary recursive datatype, even including things which did not look like lists or induction!

Our previous view of induction was a narrow one, but we would later learn that induction is more of a spectrum, ranging over a wide variety of types and data structures.

In particular, we found that when inducting on a non-traditional structure, like a tree, the **proof follows the code**. If a constructor has two recursive sub-components (such as `Node`), then the proof gets two inductive hypotheses.

Writing code is like writing a proof. Decouple the two in your mind when writing code that is meant to be correct – the process of writing the code should be akin to proving that it is correct.

Lesson An impoverished view of programming fits problems to a small set of base types. A rich view of programming fits types to problems.

Lesson 4

STRUCTURAL INDUCTION AND TAIL RECURSION

May 25, 2023

Structural induction was taught as an upgrade of normal induction, where the structure of a list could be observed as similar to the structure of the natural numbers.

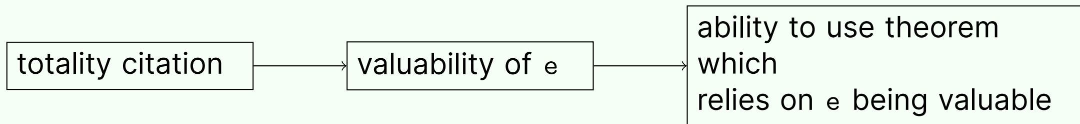
Just like how we had n and $n + 1$, we drew an analogy to the structure of a list as `xs` and `x::xs`.

The idea of **parse, don't validate** was introduced, which consists of **expressing your information through types** when possible, rather than flattening it down to something as simple and without depth as a boolean. We saw this in the translation of the function `take`:

```
fun take (n : int, L : int list) : int list =
  if isEmpty L orelse n = 0 then
    []
  else
    let
      val x = hd L
      val xs = tl L
    in
      x :: take (n - 1, xs)
    end
```

```
fun take (n : int, L : int list) : int list =
  case (n, L) of
    (0, _)      => []
  | (_, [])     => []
  | (_, x::xs) => x :: take (n - 1, xs)
```

Here, we also started learning about the importance of **totality** in proofs. We learned about the necessity of totality as a tool to get at valuability, so that we could step through code that relied on some form of eager evaluation.



Lesson **Parse, don't validate!** Express your data through types when you can.

Lesson 3

INDUCTION AND RECURSION

May 23, 2023

But before we could get to the idea of structural induction, we first had to talk about normal **induction** on the natural numbers.

We see now that induction on the natural numbers can be thought of as just an instance of structural induction on the following datatype:

```
datatype nat = Zero | Succ of nat
```

When it comes down to it, induction should be hard-wired into your brain. **Base case, induction hypothesis, inductive step, repeat.**

We also learned about the **"recursive leap of faith"** method for writing a recursive function, which simply requires a leap of faith that your function works, to write a function which does the right thing.

Lesson We call the method of solving an infinite amount of problems in finite space, "induction", or "recursion", and they are the same thing.

15-150: PRINCIPLES OF FUNCTIONAL PROGRAMMING - SUMMER 2023
Lesson 2
EQUIVALENCE, BINDING, AND SCOPE
May 18, 2023

Back during this lecture, we were still getting a hold of the basics of SML!

We learned about **extensional equivalence**, which turned out to be a pivotal notion in our understanding of programs, and of our understanding of how to write *correct* programs.

To facilitate this idea, we introduced the concept of **binding**, which differs from other programming languages which have **assignment**, in that the value of a variable never changes, when it is bound. That variable can only be **shadowed** with a different, unrelated binding that shares the same name.

This ultimately introduced the idea of **immutability**, or simply not allowing values to mutate wildly in a single context. This means that code reads the same as math – values are values.

Lesson Binding is not assignment! We can gain many benefits from simply adopting an immutable style.

Lesson 1

PROLOGUE

May 16, 2023

And then, at the start of everything, we had our first lecture.

This lecture introduced the Standard ML language, it introduced the idea of having a strong type discipline, and it made a few promises.

On the very first day, we set out with an idea of what we wanted programming to be. We also laid out our **Three Theses**, which have appeared so far throughout the slides, and throughout the course.

Lesson Functional programming is an improvement on our ability to program. It is a refinement on our ability to communicate, as programmers.

How have we kept on on those promises, and foreshadowings?

2 - Course Themes

On the first day, I posed this question to you. What is programming? What is good programming? What should good programming be?

Let's revisit those answers.

Programming should be:

- descriptive
- modular
- maintainable

Programming should be **descriptive**.

At this point in the course, we've seen many examples of this. Writing signatures that describe the interfaces of our code is one way of making our code more descriptive.

Giving our code concrete, well-specified invariants that we can think of, when programming is another way of writing descriptive code.

More generally, having powerful language constructs like higher-order functions, parametric polymorphism, and algebraic datatypes makes our programming incredibly expressive, and able to describe a wide array of problems.

Programming should be **modular**.

What better way to validate this claim than our discussion of literal modules? We saw that modules can be used to make composable software, or software that can be defined in terms of other, well-specified software components.

If we had a library for `IntSet`, refactoring the internal implementation is incredibly easy, when we hide the details via opaque ascription. We saw that outside callers literally cannot depend on things which are hidden behind the interface, making for modular code.

A strong typing discipline lends itself to modular code as well. Each expression and each variable has its own type, which gives it well-specified semantics and a well-specified way to interact with other parts of a modular codebase.

Programming should be **maintainable**.

More generally, the principle of extensional equivalence, or as I call it, the refactoring lemma, means we can **always** swap equivalent code for equivalent code. This is enormously powerful when it comes to refactoring and maintaining code.

More generally, a strong typing discipline means that maintaining code is less likely to run into errors. You can't accidentally swap an expression of type `int` for one of type `int tree`, at least not easily.

Functional code is not just nice to look at – terser, more understandable code will ultimately lead to more maintainability. You cannot maintain code that you do not understand.

What about the three theses that we learned?

Those were:

- **Recursive Problems, Recursive Solutions**
- **Programmatic Thinking is Mathematical Thinking**
- **Types Guide Structure**

Recursive Problems, Recursive Solutions is about not letting recursion be the bogeyman.

We've dealt with recursion all semester – by this point, we're pros at it. Recursion is a technique that isn't something to be feared or excluded, it's a fundamental technique that is applicable in many scenarios.

Through proper command of thinking with specifications and thinking with invariants, recursion becomes second nature. Ultimately, the recursive nature of linked lists and tree-like structures benefits a recursive approach.

Programmatic Thinking is Mathematical Thinking is about the fact that computer scientists were mathematicians first.

Before we can write any serious code, we have to be able to think about it in analytical code. Before we can write code which solves problems, we need to adopt a problem-solving mindset. It just so happens that math is the language of stating and solving problems.

Work and span, induction, extensional equivalence, and adopting formal mathematical specifications are all different ways that math shows up in our code. Embrace it – it can only make your ability to understand and solve problems better.

Types Guide Structure is about letting types dictate your thinking.

Types are the language of programs, and the blueprints that software architectures are built upon. The exchange, construction, destruction, and interplay of data are all processes that are codified via types.

We saw how powerful concepts like currying, HOFs, CPS, laziness, and polymorphism can come out of simply adjusting our type structure slightly. Seen in this way, we truly do let types guide the structure of our programs, and produce better code for it.

In addition to those, however, over the course of the semester I isolated some of the common sayings or idioms that tended to crop up during each lesson.

Some lessons are planned. Others must be discovered. These ones are the latter.

- Be Clever by Being Dumb
- Self-Defence Against Yourself
- We Are in the Business of Writing Correct Code
- Do It Better

I don't think I necessarily ever said this one out loud, but I was thinking it.

Some people think I like functional programming because I am smart. These people are incredibly wrong. I like functional programming because I am extremely prone to stupid mistakes, and without good code to support me, I would never get anything done.

At this point in the course, we've discussed the type-checker, and all the ways that it serves as both our best friend and our worst enemy. I am squarely in the "best friend" camp, because without the typechecker at my side 90% of the code I write would be nonsense.

Learning how to program (or computer science in general) is a massive rush of learning new skills and techniques, fancy names, and cool buzzwords that seem to clue you in on a brand new world.

In my experience, learning how to program well has entailed learning how to cut back on that instinctive desire to do things in a "cool" or "clever" way.

Cool is cool. Clever is clever. But clever is not maintainable. Nobody wants to read your "clever" code which requires five different assumptions to understand.

Write code which speaks for itself. Write code that is simple and expressive, and gets the job done in the least confusing way possible. To me, that is what functional programming stands for.

This has been a class on learning how to program well.

The main way that we can learn how to program well is to learn how to live with ourselves, as programmers. We are often our own worst enemies, writing code which is inunderstandable merely hours later, or leaving footguns in our code that future versions of ourself have to deal with.

To that end, the first step in learning to program well is to learn how to defend yourself against yourself. Writing safe, correct code requires that you be able to prevent yourself from making mistakes, because every programmer makes mistakes.

So let's prevent type errors, let's prevent dynamic errors instead of static ones, let's prevent ourselves from having to write messy code which we will have to deal with later. Making mistakes is OK, let's just minimize the chances.

Something I *have* said multiple times is that we are not in the business of writing code, we are in the business of writing *correct* code.

Some people are in the business of writing code. They look for code output and productivity measured in terms of silly metrics that have nothing to do with actual impact.

That's cool, but we're here to write correct code. We're here to do things right. Because nobody wants software which doesn't work. We want to make sure that our code is right, so we have things like extensional equivalence, reasoning via specification, and type safety to guide us towards that.

Do it, but do it better. Or, alternatively, we can do better.

Self-improvement is a process of first being open to improvement. Knowing how to program is good, but a theme throughout this course has been learning the numerous ways that we can improve our ability to program.

Knowing how to do something is cool, but you need to always be on the lookout for how to improve, if you want to really be good at something.

Instead of rewriting functions over and over, we can use HOFs to encapsulate common designs. Instead of running into errors at runtime, we can catch them statically. Instead of having to use redundant representations of types, we can design our own to fit our problem.

There's always better.

Before I go, I also wanted to give you a parting note on tribalism.

I vary back and forth on this, but the core idea is that throughout this course, we have been emphasizing this idea that "Functions are values", which is meant to signal the importance of being able to think of functions as themselves first-class objects which can be manipulated like any other.

This is in contrast to our rival class, which espouses "Functions are pointers". I don't need to tell all of you that this is a point of contention within the school, between these two beliefs.

My core parting gift to all of you is simply that **Functions are**.

As in, it doesn't matter.

I can, when properly baited into it, get as heated as anyone else on the idea of "functional programming vs imperative programming" or "functions are pointers vs functions are values". This is usually just for comedic effect.

In truth, **it doesn't matter**. These things are contextual. For our purposes, for teaching you this brand new thing called functional programming, of course I'm going to say that functions are values. It's a core belief of ours.

But this is just one context out of the many that people might find themselves in. This is a note on empathy.

It’s not OK to language bash. It’s not OK to judge someone based on the paradigm they employ, or the way that they like to program, or whether they prefer tabs versus spaces.

SML is a tool, just like any other. There are use cases where it is good, and there are use cases where it is not so good. That’s not at odds with my belief that functional programming is of the utmost importance to you. I think that it is pivotal that you obtain the learnings that you have so far in the course, but it **doesn’t imply being a jerk about it**.

Functional programming is not a static, well-defined thing. So it’s not worth wasting the energy compartmentalizing the world into things that are in the in-group or not. The world could just stand to be a little more functional.

Functional programming is exactly as I said on the first day. It's a mindset, a habit, a style. All programming is, or can be, functional, it's just a matter of whether you think about it explicitly or not.

Safety, simplicity, expressivity. I think these are some of the guiding three tenets behind the things which we have learned this semester.

On the first day, I promised that functional programming was a refinement on our way to program. I still steadfastly believe that. I hope that, now, you see it too.

Programing is just something linguistic. Functional programming gives us the outlook and tools necessary to make that communication better.

I'll borrow another idea here when I say that **code is art**.

Code can be beautiful.

Code can explain an idea.

Code can change how you think.

This is the first chapter of the rest of your life. Now, you have the knowledge that you need to succeed, and you can never go back.

3 - Saying Goodbye

I have many people I must acknowledge.

Teaching is not a one-person job. It's the combined efforts of myself, the rest of the course staff, and, by proxy, everyone who has ever taught me how to teach. Many people have shown up in this class, whether you knew so or not, even without showing their faces.

The list is too long for me to possibly state, but I can try anyways.

Acknowledgements

λ

I TA'd this class for many years before getting the privilege to return as an instructor. Through years of 150, I met friends, mentors, and lifelong connections.
These people influence me in every second that I teach, in ways both large and small. I have some specific ones in mind, but those people know who they are.

Aditi · Brandyn · David · Emma · George · Hannah · Harrison · Helen L. · Henry · Isabelle · Michael Zhang · Julia G. · Kalvin · Kai · Mckenna · Minji L. · Miranda · Matthew · Nikhita · Samarth · Shyam · Sue · Tim · Ashwin · Ariel · Brian · Disha · Ethan · Eunice · Gabriel · Isabel · Jacob · Kaz · Kevin · Keshav · Minji K. · Nathan · Naomi · Alexander · Siddharth G. · Mia · Avery · Agam · Cam · Eshita · Lili · Rahjshiba · Soumil · Siva · Cooper · James · Len · Leah · Abhi · Arthi · Andrew · Eric · Jon · Justin · Keiffer · Megha · Mason · Christina · Ryoha · Ryan · Runming · Samiksha · Siddharth P. · Stefan · Steven · Suhas · Surabhi · Thea · Will · Advait · Ananya · Allen · Anna · Ayush · Brandon · Eric · Ekemini · Juhi · Jimmy · Julia S. · Jonathan · Laura · Megan · Michelle · Nancy · Nicole · Pratik · Rachel · Sanjana · Dhruti · Sam · Sonya · Tarun · Xinyu · Yosef · Helen H. · Zach · Caroline · Deya · Michael Zhou · Stephen

My TAs, for caring, for persevering, and for always having the interests of the students at heart.

The connections, mentors, and lifelong friends that I have made over the years of teaching this class, some of which I left behind so that I could come here.

My friends who supported and kept me sane throughout this summer.

Dilsun Kaynar, for her pivotal support in running this course.

Semgrep, for not just supporting my decision in coming here to teach, but for going above and beyond in their sponsorship of the course.

Michael Erdmann, for imparting to me a modicum of his compassion.

Bob Harper, for showing me the power of passion.

Jacob Neumann, without which these slides would not exist.

Anil Ada, for the alternate grading schemes and drawing boxes on exams.

Ryan O'Donnell, for showing me how to start off a lecture with style.

Mor Harchol-Balter, for frequent handing out of candy during live lecture.

Suhas Kotha, for showing me the power of Hi-Chews.

Brian Railing, for the importance of in-class exercises in student learning.

Pat Virtue, for showing me it is possible to consider the individual.

I hate goodbyes.

I was never good at saying goodbye.

I spent days upon days figuring out what to write for this section, and couldn't find the right way to describe what I was feeling.

See, the problem is that I spent a lot of time premeditating these slides. I premeditate what I say, and the jokes that I make. But how can you possibly premeditate something as important as this? So, you know what? I'm not going to premeditate. I'm going to just say what I think. For once I'm going to just let myself say something off the cuff and be actually honest with you, no tricks, no jokes. Because my goodbye should be genuine, and unplanned, and... what are you looking at? Fuck.

Jokes aside.

My lectures usually take the form of stories. To me, every lecture I've given has been a very specific story, with lessons and tribulations and a beginning and end. I like to think that I've just been telling you stories over the course of this entire semester.

But what is the story of this lecture?

This is the story of me.

This story begins with my graduation. For me, it was an ending, but not a closure.

This story ends with a closure, of my time at CMU, my time with this class, and my time with this thing I gave my all for four years, called 150. This story ends with me.

How did I get here?

I always wanted to be a professor.

For years, my dream was to teach. It first came out of a pretty unfounded reason, which was that my dad, and his dad, were both professors. I thought it would be cool to be a professor.

But my senior year, I decided not to pursue a Ph.D. I decided that I could instead go to industry, and leave academia behind me. This led me to a lot of great things, don't get me wrong, but something like teaching 150 ever again would be closed to me. I made my peace with that.

Until one day, I messaged Tom Cortina, and asked him if I could teach, and somehow ended up getting this opportunity.

So I guess another part of making it hard to say goodbye is – how do you say goodbye to a dream? What do you do when your time comes to an end?

What is the theme of this lecture?

Something Worth Learning

But no, that doesn't quite make sense.

Something Worth Teaching

But actually, no. That doesn't make sense. The story is inconsistent.

More powerful than something worth learning is something worth teaching. Something worth telling people about. An idea worth spreading.

But what does it mean for something to be something worth teaching? I hear stories from some of you, telling others about the things which you have learned in this class, but I don't necessarily expect any of you to become teachers, or to want to teach functional programming to others.

For me, something worth teaching is obvious. That's my life. That's what I care about. I came here because 150 is something that is worth teaching to me.

Something I wanted you to carefully consider, when writing this presentation, is what I really want you to get out of this course. Why are we here? Why are you taking this class?

I'm here to teach you functional programming, and ostensibly you are here to learn functional programming. But that is not the end of the story.

Functional programming is a proxy for success. It's a stepping stone on the way towards that kind of goal, in some measure, but you can't forget the original goal that it set out to accomplish.

Why are you here?

I want to conduct a social experiment.[3]

Put your heads down, and think about why you are here. Think about your goals in life, think about your dreams. It doesn't need to be, and in fact most likely is not, anything related to functional programming.

Think about what you really want. Caveat, it's not allowed to be anything grades or academic related. Nobody is born dreaming of getting good grades – it's just a proxy for some other goal that we truly want.

[3] This sentence has never ended badly.

Something worth teaching means several things. For one, it means the thing which drives you, the thing which gets you out of bed, the thing you would leave your life behind to have the chance to do.

This is your something worth teaching. This is something that is worth it, at the end of the day. Chances are, it's different than mine, and that's OK.

I hope that one day, in the future, you achieve that. And more than that, I hope that what you've learned in this class somehow, in any way, helps you towards achieving that.

Don't be afraid to make an impact. Don't be afraid to give your 110%, because if there's anything that I can teach you out of this course, it's that passion makes the difference. Passion makes the journey worth it.

It might feel disingenuous for me to come up here and tell you to find something worth devoting your life to, because I'm so lucky or because it's so incomparable. But as my eleventh grade math teacher put it when I asked him, don't be mistaken for a second just because I'm your instructor, that I'm smarter or more capable. I just know more.

What I do have, and what I do feel qualified to speak on, is that I have a lot of passion for what I do. I have passion for my work, I have a passion for life, and I have a passion for teaching. And that's what shines through. It's what's gotten me here. And if you can find that same passion — then it will get you to where you need to go, too.

Something worth teaching means three things.

It represents my journey, over the past half a year, of coming here to teach, and understanding why it was so crucial that I needed to be here, why this was the most important thing I could have been doing.

The other meaning is for you. It's about you finding your something worth teaching. It's about finding your mission, aspirations, dreams, what gets you out of bed in the morning, and putting your all into it.

And finally, the thing thing is you. You, as a class. Because you have been something worth teaching.

No more stalling. This is the end.

This is goodbye to my time as an instructor, goodbye to CMU, and goodbye to 150. Time to hang the jacket up.

I will probably never teach again. But that's OK. Teaching all of you has been the most important thing I could do, because these ideas are worth teaching. Someone will continue after me.

I hope that this class has been something worth learning. I hope that this class has been something worth teaching.

Thank you. From the bottom of my heart.

Please do keep in touch.

wu.brandonj@gmail.com
https://brandonspark.github.io/
@onefiftyman
LinkedIn