# Course Introduction

**Dr. Mattox Beckman**

Illinois Institute of Technology
Department of Computer Science



## Table of Contents





## Welcome to CS 440!

Topics for discussion:

- Logisitics — instructor, grades, course objectives, lecture format
- What is a Language? — Models of computation, REP Loop
- Haskell
- How to suceed in this class.



## Me!

**Name** Mattox Beckman

**History** PhD, Fall 2003, University of Illinois at Urbana-Champaign

**Research Areas** Programming Languages, Mathematical Foundations of Computer Science

**Specialty** Partial Evaluation, Functional Programming

**Professional Interests** Teaching; Partial Evaluation; Interpreters; Functional Programming; Semantics and Types; Category Theory

**Personal Interests** Cooking; Go (Baduk, Wei-Qi, Igo); Theology; Evolution; Greek; Meditation; Kerbal Space Program; Home-brewing; ... and many many more ...

Teaching philosophy is available at
http://mccarthy.cs.iit.edu/mattox/static/teaching-philosophy.pdf

## My Responsibilities

My job is to provide an "optimal learning environment".

- Assignments will be clearly written and administered.
- Questions will be answered in a timely fashion.
- Objectives of lectures and assignments will be clearly communicated.
- Grades will be fair, meaningful, and reflect mastery of course material.
  - C grade means "can reliably recognize the correct answer"
  - A grade means "can reliably generate the correct answer"
- **If something's not going the way it should, tell me!**



## Your Responsibilities

- Check the course web page frequently.
  `http://mccarthy.cs.iit.edu/cs440`
- Subscribe to Piazza and have at least digest email.
- Do the homework assignments in order to learn them.
- Attend lectures if at all possible.
- **Take responsibility and initiative in learning material** — experiment!

**You are the one primarily responsible for your education.**



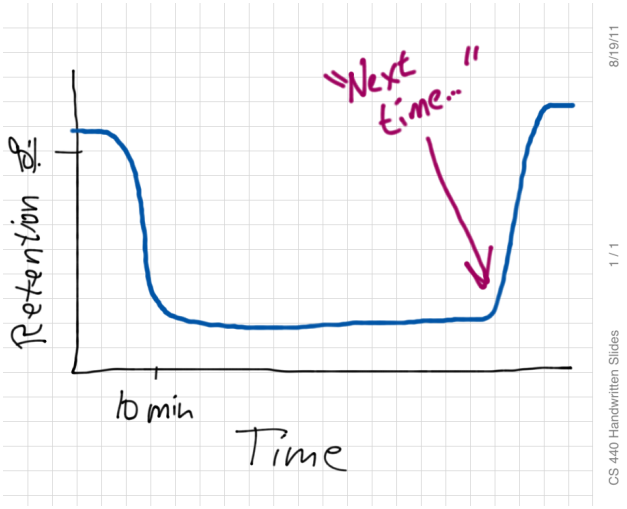
## Lectures

Speaking of lectures...

- The lecture is ancient technology; invented before the printing press.
- "Transforming lives, rehashing the past?"
- What usually happens during a lecture?



## Attention vs. Time



## Some observations about learning

- Traditional lectures are hard because:
  1. You have to be there at a certain time.
  2. ...and you have to be awake.
  3. ...and you can't "rewind" if you miss something.
  4. But at least you can ask questions! (If you're not shy.)
- Homeworks are hard because:
  1. What seemed obvious in lecture is not obvious later.
  2. You can't ask the professor for help until office hours (or until (if) they check their email).
  3. The one time you see the professor is during lecture, and then they are busy lecturing.
- Proposal: we're doing this backwards. Let's do it the right way instead.



## Reverse Lectures

### The Right Way

- Lectures will be screen-cast and made available on the course web site.
  - Usually 2–3 mini-lectures, about 10–20 minutes each.
  - Viewable on mobile devices.
  - Hard part: you do need to see them before the corresponding class period.
- During class:
  - Review time: "Any questions about the lectures?"
  - Activity. Work in groups of 2–3 people, reinforce lecture concepts, prepare you for exams. Activities are collectively worth 10% of your grade.
  - Homework. I don't think we'll have time for homework, but let me know if you're bored and I'll come up with something.
- This method is not common, but has been thoroughly tested, and **it works**.



## Contact Info


Instructor Mattox Beckman

Best Contact via email. I use inbox zero, but not on weekends.

Email Addresses `<beckman@iit.edu>` or `<mattoxbeckman@gmail.com>` They go to the same inbox. Don't spam them.

Office 110 SB

Office Hours 16:00–17:00 Tuesday, Thursday

Home Page http://mccarthy.cs.iit.edu/mattox

Teaching Assistant TBA




## Machine Problems

- Machine Problems — collectively worth 15%
- Designed to help you study for the exams, and to achieve major course objectives.
- Full collaboration allowed.
- Don't use the "perturbation method" of solving machine problems! We expect you to *understand* the solution and the process very well.
- Multiple MPs may be active at a time.
- Expect four to six assignments, and expect this number to change.

## Exams

- The purpose of an exam is to measure mastery of material.
- 1 midterm exam — worth 35%.
  - Held during class.

    Dates Thursday, June 25.
- Final exam — worth 35%
  - Date: Thursday, July 23.
  - Cumulative
  - Nice British System



## Grade Guarantees

The course will not be graded on a curve or by ranking. Instead, we have the following grade guarantees:

- 85% A
- 70% B
- 55% C
- 40% D



## Table of Contents





## Why study languages?

- *pai sei*
- Blub — see *Beating the Averages* by Paul Graham.
- Language Families

## Themes

### The Big Idea

**A Programming Language is an Implementation of a Model of Computation**

The course has three major themes:

1. Languages
   *What is a language? What kinds of things can we say in a language?*
   This covers a lot of areas.
2. Parsing
   *How do we get the computer to read what we said?*
3. Interpreting and Compiling
   *How do we get the computer to do what we said?*



## Four Fundamental Models

**A programming language is a model of computation.**

### Models

- von Neumann Machine
- Lambda Calculus (or term rewriting)
- Message Passing
- Unification



## von Neuman Machines

- Based on the physical implementation of computers: a CPU, a memory core, and a straw.
- Computation is performed by populating the memory with initial values and manipulating (destructively updating) them.
- Languages in this family are called *imperative*:

  "A program is a list of instructions for the computer to follow."

- This style is very popular!
- Example languages: C, C++, Pascal, Fortran, Forth, Assembly
- Hybrid languages: Java, Python, Ruby, etc.
- Low level of abstraction.



## Example: Assembly Language

- A program is a *series of control signals for a CPU*
- Data consists of integers, memory address, and IEEE floating point
- Based on the architecture of the CPU
- Usually not very expressive
- Examples: Assembly languages, microcode
- Abstraction: what's that?

```
1 .LFB0:  pushq   %rbp
2         movq    %rsp, %rbp
3         movl    %edi, -4(%rbp)
4         movl    -4(%rbp), %eax
5         addl    $1, %eax
6         leave
7         ret
```

## Object Oriented Languages

- Use *Message Passing* as a model of computation.
- A program is a *list of commands to be executed*
- Data consists of *objects* which can send and receive messages. — an object is a "function with state"
- Examples: Smalltalk
- Hybrids: C++, Java
- Model: More advanced abstraction; inheritance, finer-grained abstraction.

```
class Square {
public:
  int x,y,h,l;
  Square() { x = y = h = l = 0; }
};
```



## Functional Languages

- Use *functions* as a model of computation.
- A program is an *expression to be evaluated*
- Data consists of low level types and "higher order types"—functions.
- Examples: Lisp, Scheme, ML

```
let twice f x = f(f(x))
let inc x = x + 1
let x = inc 5
let y = twice inc 5
```



## Logic Programming Languages

- Advertised Model: "pure logic". If we can't prove it, we assume it's not true.
- Real Model: Unification — Find solution that satisfies multiple constraints.
- A program is a *logical predicate to satisfy*
- Data consists of a set of assertions about what we know to be true.
- Example: Prolog

```
-  human(socrates).
-  mortal(X) :- human(X).
-? mortal(Who).
Who = socrates
```



## Scripting Languages

- These don't neatly fit into a model of compuation.
- Usually imperative, often a hybrid (python and ruby are good examples)
- The *intent* of the language is what defines it.
- Rapid prototyping, easy access to system (or software package) resources.

```
lines = sys.stdin.readlines()
num = start
for i in lines:
   print "%0d %s" % (num,i),
   num = num + 1
```

## So, what should you learn?

- Understand major classes of programming languages: techniques, features, styles.
- How to select an appropriate language for a given task.
- How to read a formal specification of a language and implement it.
- How to write a formal specification of a language.
- Four Powerful Ideas:
  1. Recursion
  2. Abstraction
  3. Transformation
  4. Unification

The emphasis is on learning the theory, knowing why the theory is valuable, and using it to implement a language.



## How am I going to learn it?

There are two common approaches to teaching a PL course.

- Approach 1: "Language of the Month Club"
  - Lots of time spent on syntax, fundamentals tend to get lost.
  - You'll forget them all anyway.
- Approach 2: "Host Language"
  - Learn one language, use it to write interpreters for all the other languages.
  - You actually get to see how a language is put together.



## Table of Contents





## How to Pick an Implementation Language

- You all know a lot about Imperative/OO languages.
- Few or none of you know anything about functional languages.
- Functional languages are becoming increasingly important:
  - Roughly four times the programmer productivity.
  - Parallel computation
- Our main language is Haskell.

## Features of Haskell

- It's an advanced higher order functional language.
- Has a very modern, concise syntax.
- Has automatic type inference with parametric polymorphism.
- Used a lot in research.
- Extremely well suited for writing languages.



## Demo of Haskell

Follow along on your own computer if you can. We will go over...

- Variables
- Basic Types
- Simple Functions
- Lists



## Haskell Files

To create a Haskell file, create a file in your favorite editor. The extension is `.hs`. You can compile the file with the command

```
ghc foo.hs
```

or use it interactively by starting the interactive compiler.

```
$ ghci
GHCi, version 6.12.1: http://www.haskell.org/ghc/  :? for h
Prelude> :load foo.hs
[1 of 1] Compiling Main             ( foo.hs, interpreted )
Ok, modules loaded: Main.
*Main>
```

Another meta-command is `:edit`, and can save a lot of time when using the interactive environment.



## Global Variables

To create a global variable in Haskell, simply write

```
1 x = 10
```

the same way you would write it in mathematics. If you are using the interactive compiler, you will use the `let` keyword.

```
Prelude> let x = 10
```

## All Definitions are Recursive

This variable is *recursive* and *global*. It can be used from anywhere else in the program, even before it was defined!

```
y = x + 1
x = 5 + 5
```

This is actually because Haskell is a *lazy* language; it does not evaluate anything until it is absolutely necessary.



## Local Variables

You can create a variable with limited scope by using the `let` and `in` keywords. The basic syntax is

`let` *var* = *value* `in` *expr*

### Example

```
let x = 10
 in x + x
```

The part after the `in` keyword is called the *scope* (lifetime) of `x`.



## Multiple Lets

You can declare multiple variables in one `let` expression. Note that `let` expressions are mutually recursive.

```
xx = let x = 10 + y
         y = 20
         z = x + y
     in x + y + z
```

- Note the use of indentation to delimit scope!
- Can you figure out the value of `xx`?



## Where

A related keyword is `where`. It allows the body of the expression to come first, and the auxiliary variables to come after.

### Example

```
xx = x + y + z
  where x = 10 + y
        y = 20
        z = y + x
```

## Nesting Scopes

Sometimes you will see two variables defined that have overlapping scopes. In that case, a variable always is bound by the nearest enclosing scope.

```
x = 20
yy = x + let x = 10
             y = 20 + x + let x = 5
                          in x + 2
             z = x + y
         in x * z
```

Can you identify the scopes of the different variables?



## Named Functions

- The syntax for a named function declaration is also very simple.

```
inc x = x + 1
```

- The function does not modify $x$! It returns the *value* $x + 1$.
- Function calls are written by juxtaposition. if you write `inc 10` you will get 11 back. Function calls bind more tightly than any other operation.

```
double x = x * 2
a = double 10 + 5      -- 25
b = double $ 10 + 5    -- 30
c = double (10 + 5)    -- 30
```

- The $ operator is low precedence



## Lists

- Functional programming language people LOVE lists.
- A *list* can take two forms:
  - It can be an empty list, or
  - it can be an element, together with another list.
- Empty lists are written `[]`
- Non-empty lists are written `x : xs`
  - $x$ is the *head* of the list.
  - $xs$ is the *tail* (or *rest*) of the list.
- The elements of a list must all have the same type. (*homogeneous*)



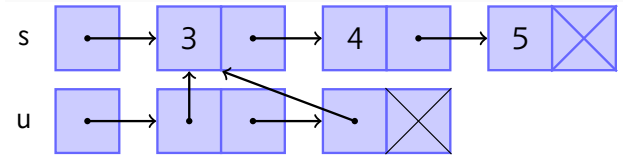
## Built-in Linked Lists

```
x = 3 : 4 : 5 : []
head x -- returns 3
tail x -- returns [4,5]
```


```
x = [3,4,5]
```

## More List Examples

```
1 s = [3,4,5] -- s :: [Int]
2
3 u = [s,s]   -- u :: [[Int]]
```






## Functions on Lists

Because lists are recursive, functions that deal with lists tend to be recursive.

```
1 mylength :: [a] -> Int
2 mylength [] = 0
3 mylength (x:xs) = 1 + mylength xs
4
5 mylength s -- would return 3
```

We will discuss recursion in depth next time.



## Simulating Stacks with Lists

Consider this code.

```
1 x = [1,2]
2
3 foo xx = length (bar (3:xx)) +
4          length xx
5
6 bar (x:xs) = xs
```

- The `3:xx` effectively pushes a 3 onto a stack.
- Once `bar` returns, the 3 is 'popped off' of `xx`.
- Also, `bar` itself pops the stack and returns it.



## Cheery Facts about Variables

- Variables can *never* have their values changed once assigned!
- Haskell *infers* the type of all expressions.
  To see the type Haskell assigned an expression, you can use the `:type` (usually abbreviated as `:t`) operation in the interactive compiler.

```
Prelude> :t inc
inc :: Integer -> Integer
Prelude> :t plus
plus :: (Num a) => a -> a -> a
```

- An arrow indicates a function type.
- A lower case letter indicates a generic type.
- The double arrow indicates a type class.

# Table of Contents



John Backus.
Can programming be liberated from the von neumann style? a functional style and its algebra of programs.
1997.
ACM ID: 1283933.

Paul Graham.
Beating the averages, April 2003.
http://www.paulgraham.com/avg.html.

George Orwell.
Politics and the english language.
*Horizon*, 13(76):252—265, April 1946.
http://www.resort.com/~prime8/Orwell/patee.html.

Introduction and Logistics Course Haskell Bibliography