# CS 115
# Functional Programming

*Lecture 1*: March 28, 2016

## Overview, philosophy, basics

# Today

- Course overview and policies
- Motivation (course philosophy)
- Introduction to Haskell

# Course overview

# Life milestones

- Birth
  - High school graduation
  - College graduation
  - Getting your dream job
  - Marriage
  - Kids
- Learning to program in a statically-typed functional programming language

# Course policies

- 9 credits, graded
- No midterm or final
- One "setup" assignment, worth 2 marks
- 6 regular assignments, graded from 0-3 as in CS 1
- 1 mark for filling out final survey
- Maximum number of marks: 21

# Course policies

- Grades:
  - A: 19-21
  - B: 16-18
  - C: 14-15
  - D: 12-13
  - F : < 12

# Adding the class

- If you aren't registered but want to add the class, no problem, *BUT:*
- You *must* fill out the CSman signup sheet before I will sign your add card!
  - (Registered students need to do this too!)
- If you don't fill out the signup sheet, you will not be able to submit assignments, and late penalties (0.5 marks/day) are in effect!

# Assignments

- Like CS 1:
  - grades are 0 (no good), 1, 2, 3 (near-perfect)
  - multiple sections
  - grade is *minimum* of section grades
  - one week rework
  - submitted through csman

# Web site

- On `moodle.caltech.edu`
- Password is `freemonad`

# Textbooks

- None required, some recommended:
  - Thinking Functionally with Haskell, by Bird
  - Haskell, the Craft of Functional Programming, 3rd ed. by Thompson
  - Real World Haskell, by O'Sullivan, Goertzen, Stewart
  - Purely Functional Data Structures by Okasaki
  - Learn You a Haskell For Great Good by Lipovaca

# Textbooks

- New online textbook:
  - Haskell Programming From First Principles
  - `haskellbook.com`
- I haven't read it but I hear it's good

# Textbooks

- I will adapt/steal material from these but will not follow them particularly closely
- Other readings may come up too (course readings, papers, blog posts)

# Course outline

- First half:
  - Basic functional programming
  - Evaluation, induction, proving correctness
  - Core Haskell
  - "Thinking functionally"

# Course outline

- Second half: Monads
  - Theory:
    - notions of computation
    - monad laws
  - Applications:
    - computations that may fail (`Maybe` monad)
    - computations that return multiple values (list monad)
    - computations that may fail in multiple specific ways with error recovery (`Error` monad)
    - computations that do input/output (`IO` monad)
    - computations that manipulate state (`State` monad)
    - imperative programming in Haskell

# Course philosophy/reason for being

# What is wrong with programming?

- Some things that are wrong are:
  - Too many bugs (too difficult to write correct programs)
  - Too much code (code is at too low a level)
  - Too hard to exploit concurrent and parallel programming

# What is wrong with programming?

- Functional programming (FP) may offer a solution to these problems
  - FP code typically has far fewer bugs than non-FP code ("If it compiles, it's very likely to be correct")
  - FP code typically at a much higher level than non-FP code (fewer lines of code to say the same thing)
  - FP code naturally lends itself to parallelization

# What is Functional Programming?

- Difficult to define precisely
- "You know it when you see it"
- Some common threads, several axes of variation

# What is Functional Programming?

- Thread 1: *Functions are data*
  - Functions can be passed as arguments to other functions
  - Functions can be returned as return values of functions
  - Functions can be created on-the-fly
- *N.B.* By this standard, many non-FP languages (*e.g.* Python) would qualify as functional languages

# What is Functional Programming?

- Thread 2: *State mutation is discouraged or forbidden completely*
  - Emphasis on using immutable data structures (singly-linked lists, trees) instead of mutable ones (arrays, hash tables)
  - Use of recursion for looping instead of counting up or down a state variable
  - Use of helper functions with extra arguments instead of mutable local state variables

# Problem(s) with mutation

- State mutation is a very fertile source of bugs
  - *e.g.* aliasing
  - references to objects behave differently than copies of objects
  - "off-by-one" errors in loops
- State mutation makes it harder to have a mathematical theory of programming
  - must model the locations where data kept
  - semantics are time-dependent

# Advantages of mutation

- Many programming problems are most naturally expressed in terms of mutating state variables
  - *e.g.* simulations
- State mutation maps well onto current microprocessor designs
  - imperative code can thus run very efficiently
- Many familiar data structures and algorithms absolutely require the ability to mutate state
  - *e.g.* see *any* standard algorithms textbook

# Programming paradigms

- Different programming "paradigms" are largely distinguished by the way they handle mutation
  - *Imperative*: allow mutation with no restrictions
  - *Object-oriented*: allow mutation internally in objects only (in response to a method call)
  - *Functional*: discourage mutation
  - *Purely functional*: disallow mutation entirely!
- This illustrates how important the "mutation problem" has been in the evolution of programming languages

# "Functional style"

- Learning to write programs without mutation is one of the hardest aspects of learning functional programming
  - like learning to program from scratch all over again
- Many functional languages (*e.g.* Scheme, Ocaml) allow mutation, allowing programmers to "cheat" and fall back on imperative habits if they want to
- Pure functional languages make this much harder, forcing you to learn *functional style*

# Other FP features

- Some (but not all) functional languages have features such as:
  - strong static type systems
  - powerful type definition facilities
  - type inference
  - interactive interpreters
  - support for monads
  - support for concurrency
  - support for parallel programming

# Survey of FP languages

- Lisp: Original FP language (1958!). Dynamically-typed, AI orientation, macros, fast compilers, "industrial strength"
- Scheme: Modern, "cleaned-up" Lisp, stronger FP orientation, hygienic macros
- Clojure: Lisp for the JVM, very functional, strong concurrency orientation
- Erlang: Dynamically-typed, concurrent FP language; emphasis is on massive concurrency using message-passing

# Survey of FP languages

- Scala: Hybrid OO/FP language, runs on JVM, statically-typed, complex but powerful type system
- Standard ML: Statically-typed functional language with imperative programming support (mutable references and arrays)
- Ocaml: Similar to Standard ML, OO extensions, fast compilers and fast code
- Haskell...

# Haskell

- A *non-strict*, *purely functional* language
- Non-strict ("lazy"): expressions are never computed unless their values are needed
- *Purely functional:*
  - no mutable values (except with monads)
  - simple computational model (substitution model)
  - easy to reason about code correctness

# Haskell

- Other features of Haskell:
  - Statically typed, compiled language
  - Very advanced type system
  - Generic programming using *type classes*
  - Imperative programming (and more!) using *monads*
  - Simulate OO features using *existential types*
  - Can even simulate dynamically-typed languages (with the `Typeable` type class)!

# Why Haskell?

- Functional programming is a new way to think about programming (a new programming *paradigm*)
- To learn a new programming paradigm, it is useful to study the purest instance of it
- Almost all other FP languages let you "cheat" and program non-functionally
- Haskell doesn't, so you *must* learn to program functionally
  - though monads allow "controlled cheating"

# Why Haskell?

- Much of the cutting-edge work in functional programming is being done in Haskell
- New FP abstractions are coming up all the time, usually first in Haskell
  - arrows
  - applicative functors
  - monoids
  - iteratees
  - functional dependencies / type families
  - *etc.*

# Why Haskell?

- Haskell is also a *practical* programming language
  - very advanced compiler (`ghc`)
  - interactive interpreter (`ghci`)
  - fast executables
  - large libraries
  - very helpful and rapidly-growing user community

# Personal observations

- Functional programming tends to spoil you as a programmer (hard to go back to non-FP languages)
  - Quote: *"Haskell is bad, it makes you hate other languages."*
- When you get used to working at a high level, with strong type systems to check your work, it's hard to give that up
- Functional languages are more fun!

# Beginning of details

# About Haskell

- Haskell is a compiled language
- Can also be run using an interpreter
  - with some restrictions
- Compiler we'll use: **`ghc`** (Glasgow Haskell Compiler)
  - state-of-the-art, many language extensions
- Interpreter we'll use: **`ghci`** (ghc interactive)
  - part of the **`ghc`** program
- Debugger: integrated into **`ghci`**

# Haskell as a calculator

- We'll work mostly with `ghci` at first
- Start up `ghci`…

```
% ghci
[... some descriptive text ...]
Prelude>
```

- Enter expressions at the prompt, hit `<return>` to evaluate them

```
Prelude> 2 + 2<return>
4
```

- Woo hoo!

# Haskell as a calculator

```
Prelude> [1..10]
[1,2,3,4,5,6,7,8,9,10]
Prelude> sum [1..10]
55
Prelude> foldr (*) 1 [1..10]
3628800
```

- `[1..10]` is a list from 1 to 10
- Function calls (like `sum`) don't require parentheses around arguments

# Haskell code in files

- Haskell source code files have names that end in **`.hs`** and (by convention) start with a capital letter (*e.g.* **`Foo.hs`**)
- Files normally define a *module* of Haskell code
- Start file **`Foo.hs`** like this:

```
module Foo where
...code goes here...
```

- (More sophisticated module declarations exist)

# Comments

- Single-line comments start with `--` and go to the end of the line

```
-- this is a comment
```

- Multi-line comments start with `{-` and go to the matching `-}`

```
{- this is a
   multiline
   comment -}
```

- Multiline comments can nest!

# File/`ghci` interaction

- `ghci` is good for interactive experimentation/ testing of code
- Cannot enter arbitrary code into `ghci` (some limitations)
  - though newer versions of `ghci` are getting closer to supporting full Haskell language
- Best approach:
  - write code in source code files
  - load into `ghci`, test

# File/`ghci` interaction

- Example: file `Foo.hs`:

```
module Foo where
double :: Int -> Int
double x = 2 * x
```

- Load into `ghci` and test:

```
Prelude> :load Foo.hs
*Foo> double 10
20
```

# File/`ghci` interaction

- `:load` is an example of a `ghci`-specific command (not part of Haskell language)
  - instruction to the interpreter: load a particular file
- Can abbreviate this as `:l`

```
Prelude> :l Foo.hs
```

- When loading a module, the prompt changes to reflect the new module

```
*Foo>
```

- The `*` means that all definitions in the module `Foo` are in scope

# Function definitions

- Definition of the `double` function in `Foo.hs`:

```
double :: Int -> Int
double x = 2 * x
```

- The first line is the function's *type declaration*
- The `::` means "has the type:"
  - so `double` "has the type" `Int -> Int`
- `Int` is the name of the type of (machine-level) integers
- `->` means that this is a function which takes one `Int` argument and produces one `Int` result

# Function definitions

- Type declarations can be omitted:

```
double x = 2 * x
```

- The compiler will try to infer what the proper type should be (*type inference*)
- This will usually work, but it's almost always a better idea to write down the type declaration explicitly
  - good documentation
  - clear statement of programmer intent
- Inferred types are often more general than you might want, *e.g.*

```
double :: Num a => a -> a
```

# Function definitions

- The definition of the function `double`:

```
double x = 2 * x
```

- is an equation describing how to transform the input (`x`) into the output
- Haskell functions are written as a series of equations describing how all possible inputs are transformed into the outputs

# Types

- Consider:

```
double :: Int -> Int
double x = 2 * x
```

- Haskell is *strongly statically typed*
- All values have a type which is known at compile time
- Types are checked during compilation
  - errors mean code doesn't compile

# Types

- Consider:

```
double :: Int -> Int
double x = 2 * x
```

- `x` has the type `Int`
- The return value of the function has type `Int`
- double has the *functional type* `Int -> Int`
- double is a *value*, just like `x` is
- Functions are values in functional languages!

# Types

- You can use **`ghci`** to query the type of any value

```
Prelude> :load Foo.hs
*Foo> :type double
double :: Int -> Int
*Foo> :t double
double :: Int -> Int
```

- **`:t`** is short for **`:type`**

# Pattern matching

- Most functions have more than one equation:

```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

- `Integer` is the type of arbitrary-precision integers
- Given an input, Haskell selects the appropriate equation to use by *pattern matching*
- Left-hand sides of equations are patterns to match

# Pattern matching

```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

- Given a function call *e.g.* `factorial 3`:
  - Haskell tries to match with `factorial 0`
  - `0` doesn't match `3` (failure)
  - Then tries to match with `factorial n`
  - This will match if `n` is `3`
  - evaluates `3 * factorial (3 - 1)`, etc.

# Pattern matching

- Much more to say about pattern matching in subsequent lectures
- Pattern matching is a pervasive feature of Haskell programming
- Beginning programmers often under-utilize it in favor of more familiar approaches
- For instance…

# `if` expression

- More conventional way to write **`factorial`** function:

```
factorial :: Integer -> Integer
factorial n = if n == 0
                then 1
                else n * factorial (n - 1)
```

- Note: Haskell has indentation-sensitive syntax, sort of like Python but less rigid
- **`then`** and **`else`** must not be to the left of **`if`**

# `if` expression

```
factorial :: Integer -> Integer
factorial n = if n == 0
                then 1
                else n * factorial (n - 1)
```

- `if` has the form:
  - `if <test> then <expr1> else <expr2>`
- `<test>` must have type `Bool` (boolean)
  - whose values are `True` and `False`
- `<expr1>` and `<expr2>` must both have same type
- cannot leave out `<expr2>`

# Next time

- More Haskell basics
- Evaluation in Haskell