This week:

- Introduction/motivation/pep talk
- Basics of Haskell
Prerequisite

- Knowledge of basic functional programming
  - e.g. Scheme, Ocaml, Erlang
  - CS 1, CS 4
  - "permission of instructor"
- Without this, course will be pretty hard
"Any programming language that doesn't change the way you think about programming is not worth learning."

-- Alan Perlis
Why learn Haskell? (1)

- Pound for pound, Haskell has more novel concepts than any programming language I've ever seen
  - and I've seen 'em all
- Very powerful and innovative type system
- Extremely high-level language
- Will make you smarter
- Fun to program in!
Why learn Haskell? (2)

- Very elegant and concise code:

  ```haskell
  quicksort :: (Ord a) => [a] -> [a]
  quicksort [] = []
  quicksort (x:xs) =
      quicksort lt ++ [x] ++ quicksort ge
  where
      lt = [y | y <- xs, y < x]
      ge = [y | y <- xs, y >= x]
  ```

- Works for any orderable type
What Haskell is good at

- Any problem that can be characterized as a transformation
- Compilers
- DSLs (*Domain-Specific Languages*)
- Implementing mathematical/algebraic concepts
- Theorem provers
What Haskell is not good at

- Any problem that requires extreme speed
  - unless you use Haskell to generate C code
- Any problem that is extremely stateful
  - e.g. simulations
  - though monads can get around this to some extent
What is Haskell, anyway?

- Haskell is a programming language
  - duh
- A functional programming language
  - you all know what that is
- A lazy functional programming language
- Has strong static typing
  - every expression has a type
  - all types checked at compile time
What is Haskell, anyway?

- Named after Haskell Curry
  - pioneer in mathematical logic
  - developed theory of combinators
    - S, K, I and fun stuff like that
Laziness (1)

- Lazy evaluation means expressions (e.g. function arguments) are only evaluated when needed.
- As opposed to strict evaluation, where arguments to a function are always evaluated before applying the function.
- What does this mean in practice?
Laziness (2)

- Lazy evaluation can do anything strict evaluation can do
  - and will get the same answer
- Lazy evaluation can also do things strict evaluation cannot do
- Seems like a minor point, but...
- Has a profound impact on the way programs are written
Laziness (3)

- Example:
  ```haskell
  let f x = 10
  f (1/0)
  ```

- In strict language, this causes an error
- In lazy language this returns 10
  - 1/0 is never evaluated, because it wasn't needed
- Big deal, right?
Laziness (4)

- Finite list of integers:
  ```
  let one_to_ten = [1..10]
  ```
  Can do this in either lazy or strict language

- Infinite list of integers:
  ```
  let positive_ints = [1..]
  ```
  Can only do in lazy language
Laziness (5)

- What can we do with this?

```ocaml
define positive_ints = [1..]
define one_to_ten = take 10 positive_ints
```

- Now the first ten `positive_ints` are evaluated
  - because we needed them to compute `one_to_ten`

- The rest are still in unevaluated form

- Details of this are handled by the system
Why lazy evaluation?

- Allows many programs to be written in a more elegant/concise manner than would otherwise be the case
- Can be costly (wrap closures around each expression to delay evaluation)
- Means evaluation order cannot be specified
  - because we don't know which arguments of a function call will be evaluated ahead of time
Lazy evaluation is a "side effect" (pun intended) of having a pure functional language.

Scheme, Lisp, Ocaml are impure functional languages.
- Also support side-effecting computations.

Pure functional languages support "equational reasoning".

Means substitution model of evaluation holds:
- Recall CS 4
- No messy environment model to worry about.
Equational reasoning means programs are much easier to reason about
- e.g. to prove correctness
- Functions are "referentially transparent"
  - i.e. they're black boxes
  - a given input will always produce same output
  - large classes of bugs that cannot happen!
- No side effects!
No side effects!

- No side effects means:
  - No assignment statements
  - No mutable variables
  - No mutable arrays
  - No mutable records
  - No updatable state at all!
  - "How do you guys live like this?"

- Need alternative ways of doing things
Haskell vs. Scheme/ML

- Haskell, like Lisp/Scheme and ML (Ocaml, Standard ML), is based on Church's lambda (\(\lambda\)) calculus.
- Unlike those languages, Haskell is pure (no updatable state).
- Haskell uses "monads" to handle stateful effects
  - cleanly separated from the rest of the language
- Haskell "enforces a separation between Church and State"
Functional data structures are automatically persistent

Means that can't change a data structure

- but can produce a new version based on old version
- new and old versions co-exist
Persistence eliminates large classes of bugs...

... but also means that many standard data structures are unusable
- arrays, doubly-linked lists, hash tables

Persistent data structures
- singly-linked lists, trees, heaps

Can be less efficient
- but generally no worse than log(n) hit
Pure functional programming

- Pure FP is kind of a programming "religion"
- Requires learning new ways to do things, new disciplines
- Rewards:
  - fewer bugs
  - greater productivity
  - higher level of abstraction
  - more fun!
End of pep talk

- We'll see concrete examples of all these vague points as we go along
- Now, on to practical matters...
Using Haskell

- Haskell is a compiled language like C, java, or ocaml
- Compiler we'll use is **ghc**
  - the Glorious Glasgow Haskell Compiler
  - state-of-the-art, many language extensions
  - mostly written in Haskell (some C)
- Initially, mainly use interactive interpreter
  - **ghci** (for "ghc interactive")
ghci

- **ghci** is a very useful learning/debugging tool
- But can't write everything in ghci that could be written in a Haskell program
  - e.g. definitions of new types
- Better approach: write code in files and load into ghci, then experiment with functions interactively
Introduction to the language

- Now will give a whirlwind introduction to most basic features of Haskell
- Much will not be covered until future weeks
Topics

- basic types, literals, operators, and expressions
- type annotations
- aggregate types: tuples, lists
- `let` bindings, conditionals
- functions and function types
- patterns, guards
First: how to write comments?

-- This is a single-line comment.
-- So is this.
{- This is a
multi-line comment. -}
{- Multi-line comments
{- can nest! -}
unlike in most other languages. -}
Simple expressions

- **Literals:**
  - 0  5  (-1)  3.14159  'c'

- **Operators:**
  - 7  +  9

- **Function application:**
  - abs  (-4)
  - sqrt  4.0
Can annotate types using :: syntax:

- 10 :: Int

This declares that 10 is an object of type Int

All type names start with capital letter

Normally don't declare most types

- compiler infers them (type inference)
- usually annotate function signatures anyway
Common primitive types

- **Int** – fixed-precision integer
- **Integer** – arbitrary-precision integer
- **Float** – single-precision float point number
- **Double** – double-precision floating point
- **Char** – Unicode character
  - Char literals written between single quotes
  - 'l' 'i' 'k' 'e' ' ' 't' 'h' 'i' 's'
Common derived types

- **Bool** – boolean truth value
  - either *True* or *False*
  - actually an algebraic data type (next week)
- **String**
  - actually a list of *Chars*
Can ask *ghci* to determine a type for you:

```haskell
Prelude> :t 10
10 :: (Num t) => t

Prelude> :t (10 :: Int)
(10 :: Int) :: Int
```

Note that numerical types more complicated than you might think (more on this later)
Common aggregate types (1)

- Tuples – an ordered sequence of pre-existing types of a fixed length
- e.g. `(Int, Float, String)` is a tuple type
- `(42, 3.14159, "Hello, world!") :: (Int, Float, String)`
- Also a type which looks like an empty tuple:
  - `()`
    - Actually the sole representative of the `()` type, also called "unit" (but it's not a tuple!)
Common aggregate types (2)

- Lists – an ordered sequence of a single type of an arbitrary (non-negative) length
- Empty list: `[]`
- Lists of `Ints`:
  - `[1, 2, 3]`
  - could write as `1 : 2 : 3 : []`
  - `: ` here is the "cons" (list construction) operator
- List ranges: `[1..10], [1,3..10], [1..]`
Common aggregate types (3)

- List type names also written with `[]`
  - `[1, 2, 3] :: [Int]`
  - `['h', 'e', 'l', 'l', 'o'] :: [Char]`
  - "hello" :: [Char]
  - "hello" :: String

- n.b. `String` and `[Char]` are equivalent
Haskell has *let* expressions like in Scheme or Ocaml:

```
let
    y = x + 2
    x = 5
in
    x / y
```

- Really like Scheme *letrec* (mutually recursive)
- Not assignments!
Many expressions can be written with indentation to delineate boundaries
- sort of like python (but better)
- always an equivalent non-indented form
- "offside rule"
Example

```plaintext
let
  y = x + 2
  x = 5
in
  x / y
```

- same as:
  ```plaintext
  let y = x + 2; x = 5 in x / y
  ```
Conditional expressions

- `if a == b then "foo" else "bar"`
- Conditional expression must evaluate to a value
- Unlike e.g. C, where `if` expression used for side effects
  - We have no side effects!
- Both branches of conditional must evaluate to same type
Function types (1)

- Function types are written in the form \( a \to b \)
- where \( a \) and \( b \) are type names

\[
\text{sqrt} :: \text{Float} \to \text{Float} \\
(>) :: \text{Integer} \to \text{Integer} \to \text{Bool}
\]

- Actual types are slightly more general and complex
  - e.g. \( \text{sqrt} :: (\text{Floating} \ a) \Rightarrow a \to a \)
Functions of multiple arguments have types like this:

\[ a \rightarrow b \rightarrow c \]

Not a syntax trick!

Functions are automatically curried

Function of two args \( a \) and \( b \), returning \( c \)

- is actually a function of one arg \( a \)
- which returns a function of one arg \( b \)
- which returns \( c \)
Operators and functions

- Operators are actually functions with special syntax
- Can convert operator into 2-arg function by surrounding with parentheses
  \[ 2 + 2 \text{ same as } (+) \ 2 \ 2 \]
- Can also convert 2-arg function into operator by surrounding with backquotes
  \[ 101 \ `\text{mod}` \ 2 \text{ same as } \text{mod} \ 101 \ 2 \]
Defining functions (1)

- Simple function definition:
  \[ \text{add} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \]
  \[ \text{add} \ x \ y = x + y \]

- \text{add} is a function of two \text{Int} arguments returning an \text{Int}
  - really a function of one \text{Int} argument returning?
  - a function of one \text{Int} returning an \text{Int}
Not the same as:

\[
\text{add :: (Int, Int) \to Int}
\]

\[
\text{add (x, y) = x + y}
\]

This \text{add} is a function of one argument, which is a tuple of two \text{Ints}

- still returns an \text{Int}
Patterns (1)

- Names on LHS of an equation are actually patterns
- Args of function matched against formal args by pattern matching
- Can have multiple equations, each with different pattern to match

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

- N.B. Use recursion for loops (if needed)
Patterns (2)

- Patterns are tried starting with first equation
  - if that doesn't match, then second equation, etc. etc.
- Patterns may include
  - constants like 0 or []
  - names like n
  - structures like lists or tuples
  - more things you'll see as we proceed
Pattern guards

- A pattern can include guards to specify non-structural aspect of thing to be matched
- Guards must have type `Bool`
- Guards tried in order until one returns `True`

```
abs :: Integer -> Integer
abs 0 = 0
abs x | x < 0 = -x           -- ' | ' starts a guard
       | otherwise = x
```
List patterns

sum :: [Int] -> Int
sum [] = 0
sum (x:xs) = x + sum xs

foo :: [Integer] -> String
foo [1,2,3] = "Hey!"
foo [4,x,7] = if x > 0 then "Whoa!" else "Hi!"
foo [] = "Nothing"
foo z = "Something else"
Tuple patterns

bar :: (Integer, String) -> String
bar (0, "hello") = "world"
bar (0, x) = x
bar (x, "foo") = "foo"
bar (x, y) = "who cares?"

- Lists and tuples get destructured during pattern matching if necessary
Next week (1)

- Algebraic datatypes
- Polymorphic types
- @ patterns and _ patterns
- case expressions
- Lambda expressions
- Operator slice notation
Next week (2)

- Useful list functions and the Prelude
- List comprehensions
- Type synonyms
- The `IO` monad and input/output
- Compiling stand-alone programs