CIS 1904: Haskell

Logistics

- HW0 due yesterday
- HW1 due next Wednesday, 11:59pm
  - Reminder: do not use Copilot or ChatGPT on the homework.
    You are welcome to use Hoogle, but do not search solutions elsewhere.
- TA office hours
- Registration

Haskell Design

Why was Haskell built?

These days, Haskell is used in industry as well as academia.

- Facebook’s spam filters
- GitHub’s Semantic library

Why was Haskell built?

These days, Haskell is used in software engineering as well as research.

- Facebook’s spam filters
- GitHub’s Semantic library

Originally, Haskell was designed to be an open-source research language.

Why was Haskell built?

Programming language researchers wanted to study design of languages that are:

- Functional
- Pure
- Lazy
- Statically typed

Why was Haskell built?

Programming language researchers wanted to study design of languages that are:

- Functional
- Pure
- Lazy
- Statically typed

Along the way, key features emerged:

- Typeclasses
- Monads
- Higher-order programming
- Algebraic data types

Why was Haskell built?

Programming language researchers wanted to study design of languages that are:

- Functional
- Pure
- Lazy
- Statically typed

Along the way, key features emerged:

- Typeclasses
- Monads
- Higher-order programming
- Algebraic data types

stay tuned for units on all of these

What is a functional language?

- Functional programming: a *programming paradigm*
  - built around **applying** and **composing functions**, often using **recursion**
  - functions are “first-class”, i.e., they can be passed around like values
    - Haskell has *higher-order* functions: functions which take other functions as arguments

e.g. `map` takes a function as a regular argument

```
map (+1) [1,2,3]

-> [2,3,4]
```

What is a functional language?

- Functional programming: a *programming paradigm*
  - built around **applying** and **composing functions**, often using **recursion**
  - functions are “first-class”, i.e., they can be passed around like values
    - Haskell has *higher-order* functions: functions which take other functions as arguments

e.g. `map` takes a function as a regular argument

```
map (+1) [1,2,3]

-> [2,3,4]
```

What is a functional language?

- Designed to “look like math”
  - Evolved out of lambda calculus
  - Built around structures from math (functors, monoids, etc.)
    - Evolved alongside category theory
  - Focuses on **evaluating terms**, not executing instructions

```
2 + 2 + 2
-> 4 + 2
-> 6
```

What is a functional language?

- Some alternatives to functional programming:
  - Imperative programming (common in C)
  - Object-oriented programming (common in Java)
  - Many languages, e.g. OCaml, are *multi-paradigm*

What is a functional language?

- Pros:
  - Easier to reason about
  - Close to the logic of what the program does
  - Typically more concise
  - Helps avoid “spaghetti code”

```
int acc = 0;
for ( int i = 0; i < lst.length; i++ ) {
   acc = acc + 3 * lst[i];
}
```

What is a functional language?

- Pros:
  - Easier to reason about
  - Close to the logic of what the program does
  - Typically more concise
  - Helps avoid “spaghetti code”

```
sum (map (3 *) lst)
```

What is a functional language?

- Pros:
    - Easier to reason about
    - Close to the logic of what the program does
    - Typically more concise
    - Helps avoid “spaghetti code”

```
sum (map (3 *) lst)
```

This type of reasoning that considers whole data structures and state spaces at once is sometimes called **wholemeal programming.**

What is a functional language?

- Cons:
  - Often less efficient
  - Can be unintuitive for low-level applications

What is a pure language?

- No side effects:
  - Printing
  - Reading from memory
  - Nondeterminism
  - Mutable state
  - Anything except evaluating a term down to a simpler term

```
int x;
x = 3;
x = 4;
```

What is a pure language?

- No side effects:
  - Printing
  - Reading from memory
  - Nondeterminism
  - Mutable state
  - Anything except evaluating a term down to a simpler term

```
x = 3
y = 4
```

What is a pure language?

- No side effects:
  - Printing
  - Reading from memory
  - Nondeterminism
  - Mutable state
  - Anything except evaluating a term down to a simpler term

We will see later how Haskell manages this (it’s monads).

What is a pure language?

Pros:

- Easier to understand and maintain
- Ease of equational reasoning can make it easier to refactor
- Easier to reason about correctness, especially in concurrent programming

Cons:

- Often less efficient
- Unintuitive for effectful programs, e.g. I/O

What is a lazy language?

In a lazy language, programs do not get evaluated until their results are needed!

```
foo :: Int → Bool
foo x = True
```

What happens if we call `foo` 217^{81}?

What is a lazy language?

In a lazy language, programs do not get evaluated until their results are needed!

```haskell
foo :: Int → Bool
foo x = True
```

What happens if we call `foo` 217^{81}?

- In most languages, we will have to calculate 217^{81} before returning `True`.
- In Haskell, we just return `True` right away!

What is a lazy language?

In a lazy language, programs do not get evaluated until their results are needed!

```
foo :: Int → Bool
foo x = True

crash :: Int
crash = error “crash”
```

What happens if we call `foo crash`?

What is a lazy language?

In a lazy language, programs do not get evaluated until their results are needed!

```
foo :: Int → Bool
foo x = True

crash :: Int
crash = error “crash”
```

What happens if we call `foo crash`?

Haskell again just returns `True`!

What is a lazy language?

Pros of laziness:

- Saves evaluation time
- Allows for infinite or partially-defined data structures

Cons of laziness:

- Hard to reason about

What is a statically-typed language?

- Static typing: types are known *at compile time*
  - e.g., the programmer annotates terms with their types (as in OCaml)
  - e.g., the compiler is able to infer types for many terms (also as in OCaml)

```
myList :: [Int]
myList = [1, 2, 3]
```

- Dynamic typing: types are determined at *runtime*
  - e.g., as in Python

```
[True, 1, “Hello”]
```

What is a statically-typed language?

- Pros of static typing:
    - helps compiler with analysis and optimizations
        - more bugs get caught at compile time, not runtime
    - immediate feedback makes development interactive
    - refactoring support from IDE
    - provides documentation
    - type-driven development
- Cons of static typing
    - annoying to write all those types
    - sometimes makes for more convoluted programs

Key theme: Abstraction

- Avoid repeated code
- Structure code in a way that conveys the logical ideas behind it

Haskell Basics

How do I run a Haskell program?

Option 1: Use the terminal

- In the terminal, navigate to the project folder and run `stack ghci`
  - You can run `stack ghci <filename>` if you only want to compile one file
  - This starts a Read-Eval-Print-Loop
- Next, you can type an expression and ghci will evaluate (interpret) it for you
  - e.g., typing `main` will run the `main` function, if present
  - e.g., typing `1+3` will print out `4`

How do I run a Haskell program?

Option 2: In VSCode

```
-- >>> 1 + 3
```

How do I run a Haskell program?

Option 2: In VSCode

`Evaluate…`

```
-- >>> 1 + 3
```

How do I run a Haskell program?

Option 2: In VSCode

```
Refresh...
-- >>> 1 + 3
-- 4
```

How do I run a Haskell program?

Note: Haskell compiles the whole file at once!

This means you can define functions “out of order”.

```
foo’ :: Int -> Bool
foo’ = foo

foo :: Int -> Bool
foo x = if x > 0 then True else False
```

That said, please try to organize files to maximize legibility.

Syntax

```
x :: Int
x = 3
```

← “x has type Int”
← “x has value 3”

VSCode can often infer and suggest type signatures.
You should always include type signatures, whether you write them yourself or accept a VSCode suggestion. They provide helpful documentation.

Syntax

```
x :: Int← “x has type Int”
x = 3          ← “x has value 3”
```

This is variable *definition*, not assignment. x is not temporarily holding the value 3, it *is* 3, like how variables are used in math.
If we try to run the following code, we get an error:

```
y :: Int
y = 3

y = 4
```

Syntax

```
f :: Int -> Int -> Int
f x y = x + y
```

Syntax

```
f :: Int -> Int -> Int
f x y = x + y

-- >>> f 1 2
-- 3
```

Syntax

```haskell
f :: Int -> Int -> Int
f x y = x + y

-- >>> f 1 2
-- 3


-- >>> 1 `f` 2
-- 3
```

Syntax

```
f :: Int -> Int -> Int
f x y = x + y


g :: Int -> Int
g = f 1


-- >>> g 2
-- 3
```

Syntax

```
f :: Int -> Int -> Int
f x y = x + y
```

```
g = f 1
```

```
ghci> :t g
g :: Int -> Int
```

Syntax

```
x :: Int
x = 3 + (-2)
```
← enclose negatives in parentheses

```
y :: Int
y = 19 `div` 3

z :: Double
z = 8.7 / 3.1
```

Syntax

```
b1 :: Bool
b1 = not (False || (True && False))

b2 :: Bool
b2 = 'a' == 'a'

b3 :: Bool
b3 = (16 /= 3) && ('p' < 'q')
```

Syntax

```haskell
f :: Int -> Int
f x = if x /= 0 then x else -1
```

Haskell does have if/then/else, but it is usually not idiomatic.

Instead, Haskell provides two options:

1. Pattern matching
2. Guards

Syntax: Pattern matching

```haskell
f :: Int -> Int
f x = if x /= 0 then x else -1
```

```haskell
f :: Int -> Int
f 0 = -1
f x = x
```

Syntax: Guards

```
f :: Int -> Int
f x = if x > 0 then x else 0
```

```
f :: Int -> Int
f x
    | x > 0 = x
    | otherwise = -1
```

Syntax: Pattern Matching and Guards

```
f :: Int -> Int
f x = if x > 0 then x else 0
```

```
f :: Int -> Int
f x | x > 0 = x
f x = -1
```

Syntax: Pattern Matching and Guards

```haskell
f :: Int -> Int
f x = if x > 0 then x else 0
```

```haskell
f :: Int -> Int
f x | x > 0 = x
f _ = -1
```

Syntax: Lists

```
someNums :: [Int]
someNums = [1,2,3]
```

```
noNums :: [Int]
noNums = []
```

← we need the annotation to know what this is a list of

Syntax: Lists

```
moreNums :: [Int]
moreNums = 0 : someNums

-- >>> moreNums
-- [0,1,2,3]

-- >>> 0 : (1 : [])
-- [0,1]
```

Syntax: Lists

```
hello1 :: String
hello1 = “hello”

hello2 :: [Char]
hello2 = [‘h’, ‘e’, ‘l’, ‘l’, ‘o’]

-- >>> hello1 == hello2
-- True
```

Syntax: Lists

```
intListLength :: [Int] -> Int
intListLength [] = 0
intListLength (_:xs) = 1 + intListLength xs
```

Syntax: Lists

```
sumEveryTwo :: [Int] -> [Int]
sumEveryTwo [] = []
sumEveryTwo (x:[]) = [x]
sumEveryTwo (x:y:zs) = (x + y) : sumEveryTwo zs
```

Syntax: Composition

```
f :: [Int] -> Int
f xs = intListLength (sumEveryTwo xs)
```

What is `f [1,2,3]`?

Syntax: Composition

```
f :: [Int] -> Int
f xs = intListLength (sumEveryTwo xs)
```

What is `f [1,2,3]`?

`2`

Exercises!

Please complete Exercises.hs.

We will come around and take attendance.