CIS 1904: Haskell

Haskell Design

Logistics

- HW0 due yesterday
- HW1 due next Monday, 11:59pm
- Today is the last day to register this course

Haskell Design

Why was Haskell built?

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

- Parsing
- Code analysis
- Web development
- Hardware design
- Data processing
- Metaprogramming
- Probabilistic programming
- etc.

Why was Haskell built?

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

- Parsing
- Code analysis
- Web development
- Hardware design
- Data processing
- Metaprogramming
- Probabilistic programming
- etc.

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 abstractions 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 abstractions emerged:

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

$\Bigg\}$ stay tuned for units on these

What is a functional language?

What is a functional language?

Functional programming: a *programming paradigm* organized around **applying and composing functions**

- Haskell is functional in the sense that it most strongly supports this paradigm
- Most languages (including Haskell) technically allow many paradigms but support some more strongly than others

Paradigm Comparison

Imperative programming

- Common in C
- Organized around *commands*

```
foo();
bar();
```

Object-oriented programming

- Common in Java
- Organized around *objects*

```
Foo x = new Foo();
x.bar();
```

Function programming

- Common in Haskell
- Organized around *functions*

```
f x = foo(bar x)
```

What is a functional language?

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

$$2 + 2 + 2 \longrightarrow 4 + 2 \longrightarrow 6$$

*out of scope of this class

[†]we will cover these in later units

What is a functional language?

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

In C:

```
int acc = 0;
for ( int i = 0; i < len; 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, easier to test and verify
  - Typically more concise
  - Helps avoid “spaghetti code”

In Haskell:

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

What is a functional language?

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

In Haskell:

```
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:
  - Can be less efficient than imperative code
  - Can be unintuitive for low-level applications

What is a pure language?

- No side effects, e.g.:
  - Printing
  - Reading from memory
  - Nondeterminism
  - Mutable state

If this definition seems somewhat fuzzy, it is!

Think of side effects as anything the code does beyond evaluation of a term to a simpler term, e.g., `2 + 2 + 2` just evaluates to `6` with no side effects.

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

In C, we can mutate variables:

```
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

In Haskell, mutating a variable is a side effect, so we have to make a new one:

```
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 enables real-world code with this restriction (hint: it’s monads).

What is a pure language?

Pros:

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

Cons:

- Often less efficient
- Steep learning curve for writing effectful programs, e.g. I/O

Note: functional code tends to be pure, but it does not always have to be.

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!

```
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:

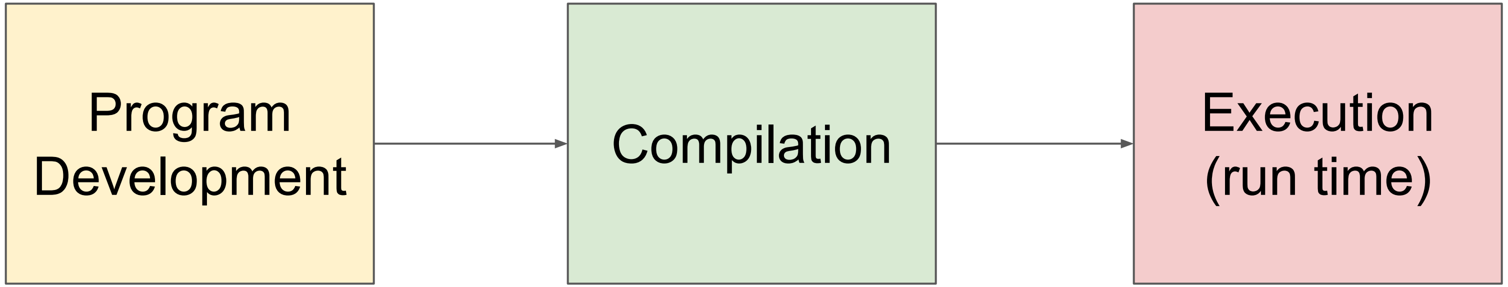
- Can be very efficient
- Allows for infinite or partially-defined data structures

Cons of laziness:

- Hard to reason about

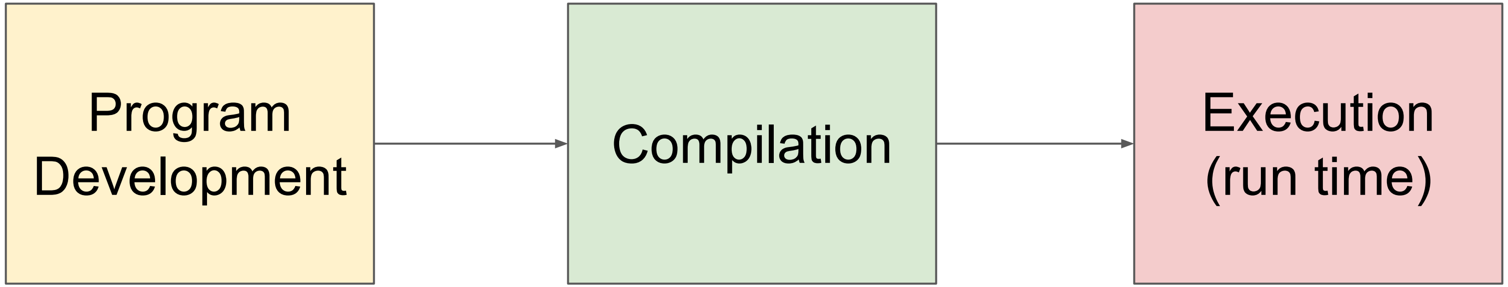
What is a statically-typed language?

- Static typing: types are known *at compile time*



What is a statically-typed language?

- Static typing: types are known *at compile time*



Execution often occurs many more times than compilation, so the more work we can shift to compilation time, the better.

Types give us information we can use to do that work at compile time.

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)

    ```
    let x : int = 1904
    ```
  - e.g., the **compiler** is able to infer types for many terms (also in OCaml)

    ```
    let s = “Haskell”
    ```
- facilitates analysis and optimization during compilation
- immediate feedback and refactoring support from IDE

  ```
  let y = s + 5
  Error: expression has type string, expected type int
  ```
- provides code documentation and design support

What is a statically-typed language?

- Dynamic typing: information we get from types is determined at *runtime*
  - e.g., as in Python
  - often quicker to write without annotations
  - sometimes gives the programmer more freedom

```
x = 1904     # type(x) returns <class 'int'>
x = "Haskell"   # type(x) returns <class 'str'>
```

What is a statically-typed language?

- Dynamic typing: information we get from types is determined at *runtime*
    - e.g., as in Python
    - often quicker to write without annotations
    - sometimes gives the programmer more freedom

```
x = 1904    # type(x) returns <class 'int'>
x = "Haskell"   # type(x) returns <class 'str'>


y = x + 5       # crashes at runtime
```

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 to help programmers understand the code
    - allows for type-driven development
- Cons of static typing
    - can be annoying to write all those types
    - improved correctness guarantees come with decreased flexibility

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`
    - The project folder is the one with a .cabal file
    - 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
```

If you do not see this suggestion after typing the above, something is wrong with your setup; please check with an instructor or TA.

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 has type Int”

`x = 3`   ← “x has value 3”

VSCode can often infer and suggest type signatures.

You should **always** include type signatures for top-level code, 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 give a variable a new value, 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
```

Function arguments do not need parentheses in general.

```
-- >>> f 1 (f 2 3)
-- 6
```

We add parentheses when we need to clarify precedence (i.e., that all of `f 2 3` is the second argument, not just `f`).

Syntax

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

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

For two-argument functions, we can write the name between the arguments if we add backticks `.

Syntax

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


g :: Int -> Int
g = f 1


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

We can partially apply functions; i.e., `f 1` is a function that takes an argument and adds it to `1`, so we can define the function `g` in terms of `f 1`.

Syntax

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

```
g = f 1
```

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

In ghci, we can use `:t` to get the type of an expression.
Other useful ghci commands include `:r` to reload the file(s) after changes and `:q` to quit the REPL.

Syntax

```
x :: Int
x = 3 + (-2)
```

← enclose negatives in parentheses to disambiguate

```
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

```
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
```

Haskell tries to match an argument against the first definition. If it does not match (i.e., it is not `0`), Haskell tries the next definition. Anything can match a variable, so that one will catch all remaining cases.

Syntax: Guards

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

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

Constructions of the form `| P` are called *guards*. Haskell will use the first definition for which `P` is satisfied, falling back on the `otherwise` case if no guards are satisfied.

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
```

We can mix pattern matching and guards!

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 _ = -1
```

_ is a wildcard, i.e., it matches anything. We use it in cases where we do not care what the argument is, i.e. we do not need a guard and we do not use the argument in the function body.

Syntax: Lists

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

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

The empty list needs a type annotation, because there is no way for Haskell to infer the type of the elements.

Syntax: Lists

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


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


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

We can add an element to the front of the list with the `:` operator, sometimes pronounced “cons”. We can also define lists directly, using square brackets and commas.

Syntax: Lists

```
hello1 :: String
hello1 = “hello”

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

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

A `String` is the same as a list of `Char`s.

Syntax: Lists

```haskell
intListLength :: [Int] -> Int

intListLength [] = 0

intListLength (_:xs) = 1 + intListLength xs
```

We can pattern match to distinguish the empty list case and the nonempty list case of a function over lists. In the second case here, we do not care what the head of the list is, so we use a wildcard for it.

Syntax: Lists

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

We can match the first two elements similarly.

Syntax: Composition

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

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

Syntax: Composition

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

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

```
2
```