COSC 4780 - Principles of Programming Languages - Spring 2022

Lecture 1: Language Refreshers

Finley McIlwaine

University of Wyoming

fmcilwai@uwyo.edu

January 20, 2022

Questions

Any trouble on homework #0?

Review

Last time we did a very quick and dirty tour of everything that we'll be covering in the course.

Today is the first day of the rest of the course, and we're starting at square one: How do we program in these languages (Java and Haskell)? We'll do our refreshers for each of these languages side by side, and then conclude by programming a solution to a problem in both languages.

Tools

How do we actually write and run programs in these languages? Writing programs is one thing, running them is another. There's flexibility in how we write these programs, but not much flexibility in how we actually run them.

Both processes use what we'll generally refer to as *developer tools*. These are applications that either make the actual writing of programs more enjoyable (IDE software, formatters, etc.), or actually allow us to run our programs (compilers and interpreters).

Tools

If you use the recommended method of setting up our tools for this course (Docker devcontainer, homework #0), then you have all the tools we need including those that aren't necessary but will make writing programs much easier.

Let's compare the methods: raw text editing in `TextEdit` (like Notepad for Mac) vs. full IDE-like experience in VSCode. Obviously we know which one will be better, but just for demonstration purposes.

Tools

Live example:

Writing programs with and without tools

Runnable Programs

Okay, now we know how we'd like to write these programs. We also should now have an idea what a runnable program looks like in each language.

In Java:

```
public class Main {
    public static void main(String[] args) {
        ...
    }
}
```

In Haskell (admittedly exceptionally minimalist):

```
main = ...
```

Language Features

Let's compare the languages side by side:

Feature	**Haskell**	**Java**
Paradigm	Purely functional	Unabashedly imperative/OO
Type system	Strong static typing	Weaker static typing
Data types	Algebraic data types	Classes
Pattern Matching	"First class"	`instanceof`/visitor
Polymorphisms	Parametric, ad-hoc	Parametric, ad-hoc, subtype

That's a lot of mumbo jumbo. By the end of this lecture, hopefully we have fuzzy notions in our heads of what all of this means.

Paradigms: Purely Functional vs. Imperative/OO

Word by word:

- *Purely*: Pure functions are those which when given equivalent inputs will produce equivalent outputs. All functions in Haskell are pure, besides those that do IO, but even that's still sort of pure.
- *Functional*: Functional style of computing is one where the primary method of computation is by the application of functions to arguments. Functional languages are those which not only support but also encourage a functional style of programming.
- *Imperative*: Imperative style of computing is one where the primary method of computation is by running a sequence of statements that operate on values stored in a shared read/write memory.
- *Object Oriented*: Software designed around encapsulation of data and procedures in "objects".

Paradigms: Purely Functional vs. Imperative/OO

Many of us are already comfortable with imperative, object-oriented programming. Many of us also wince at the thought of using Haskell ever again. Consider this class an opportunity for some exposure therapy.

(Purely) functional programming languages are weird and confusing, and Haskell is no exception. This weirdness doesn't come from the raw ideas of applying functions to arguments, but from the extreme levels of abstraction that programs in these languages typically use for even simple tasks.

Paradigms: Purely Functional vs. Imperative/OO

Live example:

Get a string from standard input and print it to standard output.

Paradigms: Purely Functional vs. Imperative/OO

In Java:

```
public class Main {
    public static void main(String[] args) throws IOException {
        String input = new String(System.in.readAllBytes());
        System.out.print(input);
    }
}
```

Warm and fuzzy, familiar. Not much abstraction at play.

Paradigms: Purely Functional vs. Imperative/OO

In Haskell:

```haskell
main :: IO ()
main = do
    input <- getContents
    putStrLn input
```

Confusing, offensive, derogatory... hostile?... Elegant????

We couldn't fully understand all the things going on this little code snippet until over halfway through 3015 last semester! That's not because applying functions is hard, but because this is turning the abstraction knob up to 11 (relative to Java).

Type Systems: Strong vs. Weak Static Typing

Both Haskell and Java are statically typed. Statically typed languages do type checking at compile time. The purpose of static type checking is to guarantee certain erroneous or ill-defined programs are never executed.

In the table above, I said Haskell uses *strong* static typing while Java uses *weaker* static typing. While those words are commonly used to describe type systems, their definitions are fuzzy and context-dependent. Here, I use them to mean Java's type system generally makes much weaker guarantees about the behavior of Java programs in contrast to Haskell's.

Type Systems: Strong vs. Weak Static Typing

Live example:

Breaking Haskell vs. Breaking Java

Type Systems: Strong vs. Weak Static Typing

Programming in Java's type system:

Type Systems: Strong vs. Weak Static Typing

Programming in Haskell's type system:

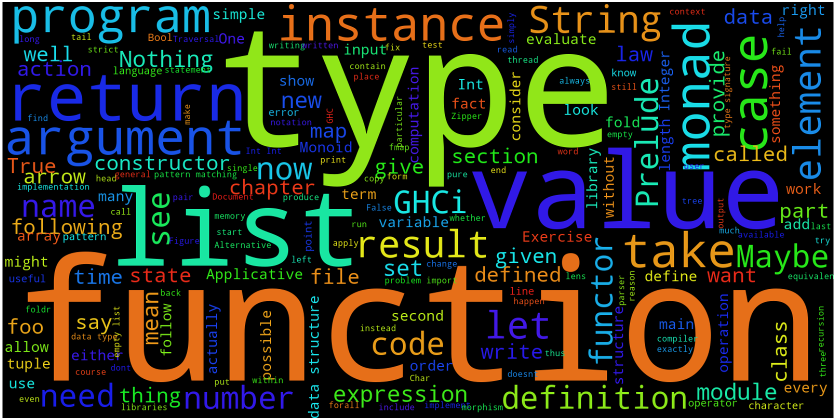
Data Types, Pattern Matching

This is the most important comparison we will make, especially in the context of this course. Types in Haskell are *algebraic data types*, while Java uses classes.

The differences this makes in practice will have the biggest impact on the ways you complete the assignments in this course.

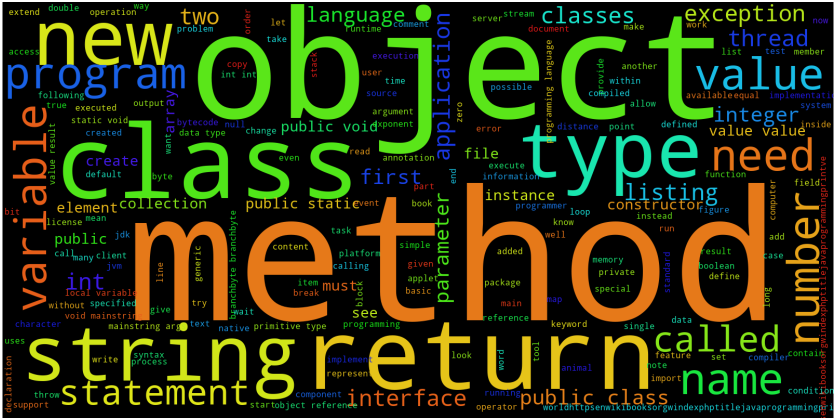
Data Types, Pattern Matching

Before we jump into an example: I made a word cloud of the entire Haskell Wikibook:



Data Types, Pattern Matching

And a word cloud of the entire Java Programming Wikibook:



Type Systems: Strong vs. Weak Static Typing

Live example:

Data types and pattern matching in Haskell vs. Java

Polymorphisms

The polymorphisms in each language won't play a big role in the course. They are important, and they'll be helping in the background, but we don't need to examine them up close at the moment.

All the code from this lecture will be included on the course page for this lecture. Also, some interesting code (that we can cover if we have time) is include in files titled `CrashCourse.java` and `CrashCourse.hs`. The Haskell code goes all the way to implementing a monad instance for the Maybe type. I attempted to match the Java code function-by-function with the Haskell code, but the wheels fell off at the applicative instance.

The End