Thinking about code like math

An intro to functional programming with Haskell

Slides by Sascha Strand

About this lab

- Meets after lecture and lunch on Tuesdays and Thursdays
- 20-30 minutes of lecture
- 50-60 minutes of lab time
- Faculty advisor Mark Jones, instructor of CS 557 Functional Languages
- Review lab outline.
- Random partners or choose groups?

Optionally for Credit

- Credit will be based on attendance
- Optionally make up 20% of your grade for the overall course
- Reducing the weight of the exams and assignments in the OS section
- You must commit this week to either grading scheme for the course
- To get credit for the lab, you're expected to attend and participate in all the labs. You can miss two labs and still get full credit. If you miss more than two you will not receive credit for the lab section.

Goals of this course

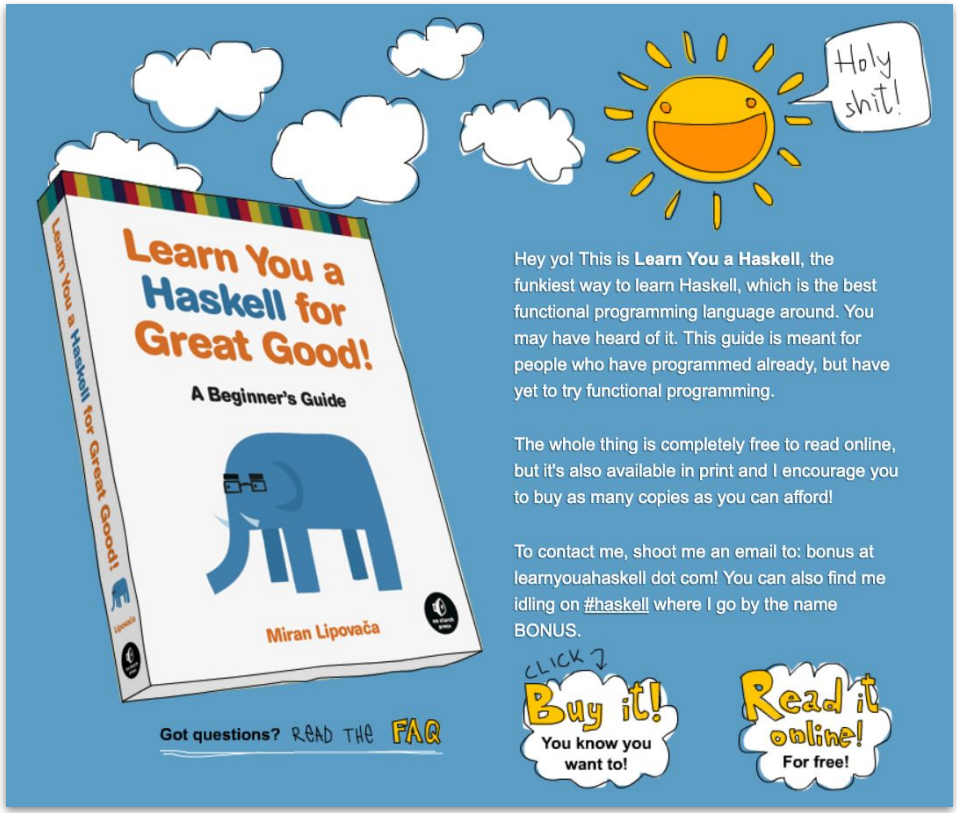
- Describe the key characteristics of the functional paradigm.
- Define algebraic data types to model useful domains.
- Identify and program:
  - Head recursion
  - Tail recursion
  - Mutual recursion
  - Multiple recursion
  - Infinite and halting recursion
- Write small Haskell programs to manipulate lists and trees.
- Use first-class functions as data values.
- Use higher order functions, including maps, folds, and composition
- Write a multithreaded map-reduce application over lists

Additional topics we can cover

CS 557, Functional Programming, contains additional topics we can cover here

- Write programs using simple infinite data structures.
- Explain and use polymorphic functions and data types, and interpret type error messages.
- Explain the differences between eager and lazy evaluation and the significance of each.
- Explain the basic implementation considerations for a functional language.

Reference material for the lab



Miran Lipovača's

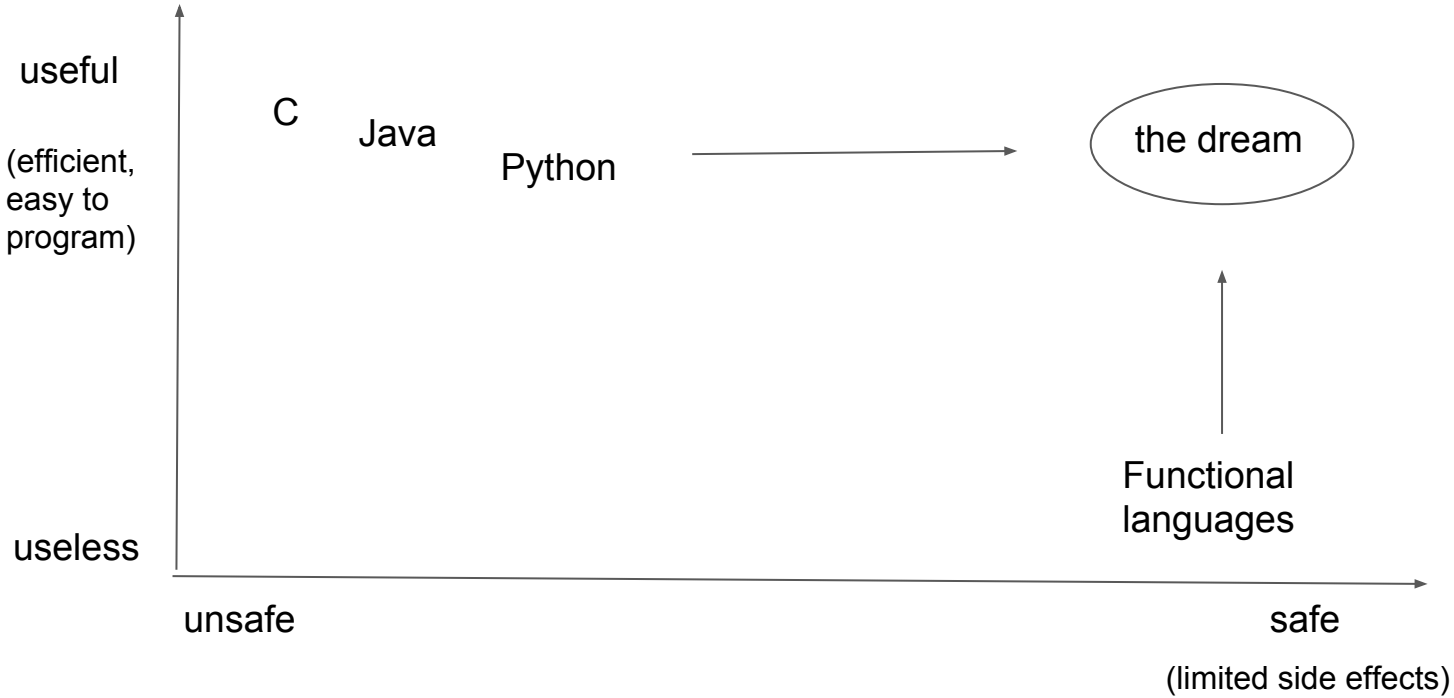
Learn You a Haskell for Great Good!

Available for free online at
http://learnyouahaskell.com

What is a functional language?

> “Functional programming is so called because its fundamental operation is the application of functions to arguments. A main program itself is written as a function that receives the program’s input as its argument and delivers the program’s output as its result. *Typically the main function is defined in terms of other functions, which in turn are defined in terms of still more functions, until at the bottom level the functions are language primitives.*”
>
> -- John Hughes
> *Why Functional Programming Matters* (1990)

Why use functional languages



From Simon Peyton Jones' Why Haskell is Useless

Why learn functional languages?

- Learn a new paradigm, see problems from this different perspective
- Learn a tool that’s gaining popularity to smoothly take on some particularly rough challenges in corners of the software world
  - Distributed systems
  - Browser-side (front-end) programming
  - High security, mission critical applications
- Master recursion in its natural environment

Why Haskell

- Haskell is a pure functional language.
- In an imperative language, you give the computer a series of instructions.
- Assignment instructions set values at memory locations.
- Control flow instructions change those values some number of times based on other values.
- In a functional language, you don’t tell the computer what to do, but rather what something is.
- This is what makes Haskell “declarative”.

Small declarative example

Haskell

```
fib 0 = 0
fib 1 = 1
fib n = fib(n-1) + fib(n-2)
```

C

```
int fib(int n) {
     if (n <= 1)
        return n;
     return fib(n-1) + fib(n-2);
  }
```

Why Haskell: Referential transparency

- No matter the state of the environment outside the function, if a function is called twice with the same parameters, it will return the same result.
- Once inside a function, we can’t interact with the outside world.
- This makes it generally easier to reason about and even prove the correctness of code.

Why Haskell: Lazy evaluation

- “Laziness” in programming languages is the result of two complementary strategies
  - Only run a function or generate a value when that value is needed (call-by-name)
  - Pass the intermediate result of a nested function to its caller as soon as it’s available (sharing)
- Call-by-name

  `let y = ((((0 + 1) + 2) + 3) + 4)`

  Stored just like is in memory until it needs to be evaluated.
- Sharing

  `map sqrt (map sqrt [16, 256, 65536])`

  Each intermediate result of the inner map is passed to the outer map

Why Haskell: Static Typing

- Like C, the compiler knows the type of every expression
- Before compilation happens, a type checker runs to determine if all your types align.
- Unlike C, the compiler performs *type inference* to determine the types without you needing to annotate them.
- The type inference system generally doesn’t require a type signature for a function, but including one is good practice.

Embracing a strong, static typing system

> “Well-typed programs can’t go wrong.”
>
> -- Robin Milner
> A Theory of Type Polymorphism in Programming

- Going wrong here means exhibiting undefined behavior
- Like crashing when the program does something the OS does not allow
- Technically, this claim requires a formalization of the type system Haskell does not have. You can crash Haskell programs.
- But the spirit of the quote conveys the value in working through type errors in a strong, static typing system before compile time.
- Sometimes described as “moving debugging work up front”

Intro summary

- All the slides and exercises for the lab will be on my course page: http://web.cecs.pdx.edu/~sastrand/
- Our lab reference manual: http://learnyouahaskell.com/
- Review course web page