Introduction to Functional Programming in Haskell
Outline

Why learn functional programming?

The essence of functional programming
  What is a function?
  Equational reasoning
  First-order vs. higher-order functions
  Lazy evaluation

How to functional program
  Functional programming workflow
  Data types
  Type-directed programming
  Haskell style
  Refactoring (bonus section)
Outline

Why learn functional programming?

The essence of functional programming

How to functional program
Why learn (pure) functional programming?

1. This course: strong correspondence of core concepts to PL theory
   - abstract syntax can be represented by algebraic data types
   - denotational semantics can be represented by functions

2. It will make you a better (imperative) programmer
   - forces you to think recursively and compositionally
   - forces you to minimize use of state

…essential skills for solving big problems

3. It is the future!
   - more scalable and parallelizable
   - functional features have been added to most mainstream languages
Programming is not about data. It’s about transforming data …
That’s why I think functional programming is a natural successor to object-oriented programming. In OOP, we’re constantly concerned about the state of our data. In functional programming, our focus is on the transformation of data. And transformation is where the value is added.

—Dave Thomas, The Pragmatic Programmer
Without understanding functional programming, you can’t invent MapReduce, the algorithm that makes Google so massively scalable.

—Joel Spolsky, Joel on Software
So that's the big deal about functional languages; and it is one big fricking deal. There is a freight train barreling down the tracks towards us, with multi-core emblazoned on it; and you’d better be ready by the time it gets here.

—Robert C. Martin, a.k.a. “Uncle Bob”
Outline

Why learn functional programming?

The essence of functional programming
  What is a function?
    Equational reasoning
    First-order vs. higher-order functions
    Lazy evaluation

How to functional program
What is a (pure) function?

A function is **pure** if:

- it always returns the same output for the same inputs
- it doesn’t do anything else — no “side effects”

In Haskell: whenever we say “function” we mean a **pure function**!
What are and aren’t functions?

Always functions:
- mathematical functions $f(x) = x^2 + 2x + 3$
- encryption and compression algorithms

Usually not functions:
- C, Python, JavaScript, … “functions” (procedures)
- Java, C#, Ruby, … methods

Haskell only allows you to write (pure) functions!
Why procedures/methods aren’t functions

- output depends on environment
- may perform arbitrary side effects
Outline

Why learn functional programming?

The essence of functional programming
- What is a function?
  - Equational reasoning
  - First-order vs. higher-order functions
  - Lazy evaluation

How to functional program
Getting into the Haskell mindset

![Image of a meditating person vs. a person running]

**Haskell**

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

**Java**

```java
int sum(List<Int> xs) {
    int s = 0;
    for (int x : xs) {
        s = s + x;
    }
    return s;
}
```

In Haskell, “=” means *is not change to!*
Getting into the Haskell mindset

Quicksort in Haskell

```haskell
qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort (filter (<= x) xs)
  ++ [x] ++ qsort (filter (> x) xs)
```

Quicksort in C

```c
void qsort(int low, int high) {
  int i = low, j = high;
  int pivot = numbers[low + (high-low)/2];
  while (i <= j) {
    while (numbers[i] < pivot) {
      i++;
    }
    while (numbers[j] > pivot) {
      j--;
    }
    if (i <= j) {
      swap(i, j);
      i++;
      j--;
    }
  }
  if (low < j) qsort(low, j);
  if (i < high) qsort(i, high);
}
void swap(int i, int j) {
  int temp = numbers[i];
  numbers[i] = numbers[j];
  numbers[j] = temp;
}
```
Referential transparency

An expression can be replaced by its value without changing the overall program behavior.

\[ \text{length} \ [1,2,3] + 4 \]

\[ = \]

\[ 3 + 4 \]

what if \textit{length} was a Java method?

\textbf{Corollary}: an expression can be replaced by any expression with the same value without changing program behavior.

Supports \textit{equational reasoning}
Equational reasoning

Computation is just substitution!

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

sum [2,3,4] 
⇒ sum (2:(3:(4:[[]))
⇒ 2 + sum (3:(4:[]))
⇒ 2 + 3 + sum (4:[])
⇒ 2 + 3 + 4 + sum []
⇒ 2 + 3 + 4 + 0
⇒ 9
Describing computations

**Function definition**: a list of **equations** that relate inputs to output
- matched top-to-bottom
- applied left-to-right

**Example: reversing a list**

**imperative view**: how do I rearrange the elements in the list?  
**functional view**: how is a list related to its reversal?

```haskell
reverse :: [a] -> [a]  
reverse [] = []  
reverse (x:xs) = reverse xs ++ [x]
```
Exercise

1. Evaluate: \( \text{double \ (succ \ (double \ 3))} \)
   
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>14</td>
<td>16</td>
</tr>
</tbody>
</table>

2. Prove, up to associativity of (+), using equational reasoning:
   
   \( \text{double \ (succ \ x)} = \text{succ \ (succ \ (double \ x))} \)
Exercise

1. Evaluate: $\text{foo (foo 0 0) 0}$
   
   - A: 0
   - B: 2
   - C: 3

2. Evaluate: $\text{foo 0 (foo 0 0)}$
   
   - A: 0
   - B: 2
   - C: 3
Outline

Why learn functional programming?

The essence of functional programming

  What is a function?
  Equational reasoning
    First-order vs. higher-order functions
  Lazy evaluation

How to functional program
First-order functions

Examples

- \( \text{cos} :: \text{Float} \rightarrow \text{Float} \)
- \( \text{even} :: \text{Int} \rightarrow \text{Bool} \)
- \( \text{length} :: [a] \rightarrow \text{Int} \)
Higher-order functions

Examples
- map :: (a -> b) -> [a] -> [b]
- filter :: (a -> Bool) -> [a] -> [a]
- (.) :: (b -> c) -> (a -> b) -> a -> c
Higher-order functions as control structures

**map**: loop for doing something to each element in a list

\[
\text{map} :: (a \to b) \to [a] \to [b] \\
\text{map} f [] = [] \\
\text{map} f (x:xs) = f x : \text{map} f xs
\]

\[
\text{map} f [2,3,4,5] = [f 2, f 3, f 4, f 5] \\
\text{map} \text{even} [2,3,4,5] \\
= [\text{even} 2, \text{even} 3, \text{even} 4, \text{even} 5] \\
= [\text{True}, \text{False}, \text{True}, \text{False}]
\]

**fold**: loop for aggregating elements in a list

\[
\text{foldr} :: (a\to b\to b) \to b \to [a] \to b \\
\text{foldr} f y [] = y \\
\text{foldr} f y (x:xs) = f x (\text{foldr} f y xs)
\]

\[
\text{foldr} (+) 0 [2,3,4] \\
= (+) 2 ((+) 3 ((+) 4 0)) \\
= 2 + (3 + (4 + 0)) \\
= 9
\]
Function composition

Can create new functions by **composing** existing functions

- *apply the second function, then apply the first*

**Function composition**

\[(f \circ g) x = f(g x)\]

**Types of existing functions**

- `not :: Bool -> Bool`
- `succ :: Int -> Int`
- `even :: Int -> Bool`
- `head :: [a] -> a`
- `tail :: [a] -> [a]`

**Definitions of new functions**

- `plus2 = succ . succ`
- `odd = not . even`
- `second = head . tail`
- `drop2 = tail . tail`
Currying / partial application

In Haskell, functions that take multiple arguments are **implicitly higher order**

```
plus :: Int -> Int -> Int
```

### Curried
```
plus 2 3
```

### Uncurried
```
increment :: Int -> Int
increment = plus 1
```

```
plus :: (Int,Int) -> Int
```

Haskell Curry
Outline

Why learn functional programming?

The essence of functional programming
  What is a function?
  Equational reasoning
  First-order vs. higher-order functions
  Lazy evaluation

How to functional program
Lazy evaluation

In Haskell, expressions are reduced:
- only when needed
- at most once

Supports:
- infinite data structures
- separation of concerns

\[
nats :: [\text{Int}] \\
nats = 1 : \text{map} \ (+1) \ nats
\]

\[
fact :: \text{Int} \rightarrow \text{Int} \\
fact n = \text{product} \ (\text{take} \ n \ nats)
\]

\[
\text{min3} :: [\text{Int}] \rightarrow [\text{Int}] \\
\text{min3} = \text{take} \ 3 \ . \ \text{sort}
\]

What is the running time of this function?
Outline

Why learn functional programming?

The essence of functional programming

How to functional program
  Functional programming workflow
  Data types
  Type-directed programming
  Haskell style
  Refactoring (bonus section)
“obsessive compulsive refactoring disorder”
FP workflow (detailed)

1A. Data Description

1B. Data Examples

2. Function Description (Signature/Purpose/Header)

3. Functional Examples

4. Function Template

5. Code

6. Tests

7. Review & Refactor

Norman Ramsey, On Teaching “How to Design Programs”, ICFP’14
Outline

Why learn functional programming?

The essence of functional programming

How to functional program

Functional programming workflow

Data types

Type-directed programming

Haskell style

Refactoring (bonus section)
Algebraic data types

Data type definition
- introduces new **type** of value
- enumerates ways to **construct** values of this type

Some example data types
- `data Bool = True | False`
- `data Nat = Zero | Succ Nat`
- `data Tree = Node Int Tree Tree | Leaf Int`

Definitions consists of …
- a **type name**
- a list of **data constructors** with **argument types**

Definition is **inductive**
- the arguments may **recursively** include the type being defined
- the constructors are the **only way** to build values of this type
Anatomy of a data type definition

```haskell
data Expr = Lit Int
  | Plus Expr Expr
```

Example: \(2 + 3 + 4\) \(\text{Plus (Lit 2) (Plus (Lit 3) (Lit 4))}\)
<table>
<thead>
<tr>
<th>Haskell</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{data Tree} = \texttt{Node Int Tree Tree} \mid \texttt{Leaf}</td>
<td>\texttt{abstract class Tree { ... }}</td>
</tr>
</tbody>
</table>
| \texttt{class Node extends Tree \{ \}
  \hspace{1em} \texttt{int label;}
  \hspace{1em} \texttt{Tree left, right;}
  \hspace{1em} \texttt{...}
  \texttt{\}} | \texttt{class Leaf extends Tree \{ \}} |

- separation of type- and value-level
- set of cases closed
- set of operations open

- merger of type- and value-level
- set of cases open
- set of operations closed

Extensibility of cases vs. operations = the “expression problem”
Type parameters

Like generics in Java

data List a = Nil | Cons a (List a)

Specialized lists

type IntList = List Int
type CharList = List Char
type RaggedMatrix a = List (List a)
Outline

Why learn functional programming?

The essence of functional programming

How to functional program

Functional programming workflow
Data types
Type-directed programming
Haskell style
Refactoring (bonus section)
Outline

Why learn functional programming?

The essence of functional programming
  What is a function?
  Equational reasoning
  First-order vs. higher-order functions
  Lazy evaluation

How to functional program
  Functional programming workflow
  Data types
  Type-directed programming
  Haskell style
  Refactoring (bonus section)
Tools for defining functions

Recursion and other functions

```haskell
sum :: [Int] -> Int
sum xs = if null xs then 0
        else head xs + sum (tail xs)
```

Pattern matching

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

(1) case analysis

(2) decomposition

Higher-order functions

```haskell
sum :: [Int] -> Int
sum = foldr (+) 0
```

no recursion or variables needed!
What is type-directed programming?

Use the **type** of a function to help write its **body**
Type-directed programming

Basic goal: transform values of **argument types** into **result type**

<table>
<thead>
<tr>
<th>If argument type is ...</th>
<th>If result type is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>• atomic type (e.g. Int, Char)</td>
<td>• atomic type</td>
</tr>
<tr>
<td>• apply functions to it</td>
<td>• output of another function</td>
</tr>
<tr>
<td>• algebraic data type</td>
<td>• algebraic data type</td>
</tr>
<tr>
<td>• use pattern matching</td>
<td>• build with data constructor</td>
</tr>
<tr>
<td>• case analysis</td>
<td>• function type</td>
</tr>
<tr>
<td>• decompose into parts</td>
<td>• function composition or partial application</td>
</tr>
<tr>
<td>• function type</td>
<td>• build with lambda abstraction</td>
</tr>
<tr>
<td>• apply it to something</td>
<td></td>
</tr>
</tbody>
</table>
Outline

Why learn functional programming?

The essence of functional programming

How to functional program
  Functional programming workflow
  Data types
  Type-directed programming
  Haskell style
  Refactoring (bonus section)
Outline

Why learn functional programming?

The essence of functional programming
   What is a function?
   Equational reasoning
   First-order vs. higher-order functions
   Lazy evaluation

How to functional program
   Functional programming workflow
   Data types
   Type-directed programming
   Haskell style
   Refactoring (bonus section)
Good Haskell style

Why it matters:
- layout is significant!
- eliminate misconceptions
- we care about *elegance*

Easy stuff:
- **use spaces!** (tabs cause layout errors)
- align patterns and guards

*See style guides on course web page*
Function application:

- is *just a space*
- associates to the left
- binds most strongly

\[ f(x) \]
\[ (f \ x) \ y \]
\[ (f \ x) + (g \ y) \]

Use parentheses only to *override* this behavior:

- \( f \ (g \ x) \)
- \( f \ (x + y) \)
Outline

Why learn functional programming?

The essence of functional programming

How to functional program
  Functional programming workflow
  Data types
  Type-directed programming
  Haskell style
  Refactoring (bonus section)
Refactoring in the FP workflow

Motivations:
- separate concerns
- promote reuse
- promote understandability
- gain insights

“obsessive compulsive refactoring disorder”
Refactoring relations

Semantics-preserving laws prove with equational reasoning and/or induction

• Eta reduction:
  \( \lambda x \rightarrow f \ x \equiv f \)

• Map–map fusion:
  \( \text{map } f \ . \text{map } g \equiv \text{map } (f \ . \ g) \)

• Fold–map fusion:
  \( \text{foldr } f \ b \ . \text{map } g \equiv \text{foldr } (f \ . \ g) \ b \)

“Algebra of computer programs”

John Backus, Can Programming be Liberated from the von Neumann Style?, ACM Turing Award Lecture, 1978
Strategy: systematic generalization

commas :: [String] -> [String]
commas [] = []
commas [x] = [x]
commas (x:xs) = x : " , " : commas xs

Introduce parameters for constants

seps :: String -> [String] -> [String]
seps _ [] = []
seps _ [x] = [x]
seps s (x:xs) = x : s : seps s xs

Broaden the types

intersperse :: a -> [a] -> [a]
intersperse _ [] = []
intersperse _ [x] = [x]
intersperse s (x:xs) = x : s : intersperse s xs
Strategy: abstract repeated templates

**abstract** (v): extract and make reusable (as a function)

```haskell
showResult :: Maybe Float -> String
showResult Nothing = "ERROR"
showResult (Just v) = show v

moveCommand :: Maybe Dir -> Command
moveCommand Nothing = Stay
moveCommand (Just d) = Move d

safeAdd :: Int -> Maybe Int -> Int
safeAdd x Nothing = x
safeAdd x (Just y) = x + y
```

Repeated structure:
- pattern match
- default value if **Nothing**
- apply function to contents if **Just**
Strategy: abstract repeated templates

Describe repeated structure in function

```haskell
maybe :: b -> (a -> b) -> Maybe a -> b
maybe b _ Nothing = b
maybe _ f (Just a) = f a
```

Reuse in implementations

```haskell
showResult   = maybe "ERROR" show
moveCommand  = maybe Stay Move
safeAdd x    = maybe x (x+)
```
Refactoring data types

data Expr = Var Name
  | Add Expr Expr
  | Sub Expr Expr
  | Mul Expr Expr

vars :: Expr -> [Name]
vars (Var x) = [x]
vars (Add l r) = vars l ++ vars r
vars (Sub l r) = vars l ++ vars r
vars (Mul l r) = vars l ++ vars r

eval :: Env -> Expr -> Int
eval m (Var x) = get x m
eval m (Add l r) = eval m l + eval m r
eval m (Sub l r) = eval m l - eval m r
eval m (Mul l r) = eval m l * eval m r
Refactoring data types

Factor out shared structure

data Expr = Var Name
     | BinOp Op Expr Expr

data Op = Add | Sub | Mul

vars :: Expr -> [Name]
vars (Var x) = [x]
vars (BinOp _ l r) = vars l ++ vars r

eval :: Env -> Expr -> Int
eval m (Var x) = get x m
eval m (BinOp o l r) = op o (eval m l) (eval m r)
where
    op Add = (+)
    op Sub = (-)
    op Mul = (*)