Introduction to Functional Programming in Haskell
Outline

Haskell basics

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

Haskell basics

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)
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

Haskell basics

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)
Getting into the Haskell mindset

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

```
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

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

Quicksort in 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.

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

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

Supports equational reasoning.
Equational reasoning

Computation is just substitution!

\[
\begin{align*}
\text{sum} & : [\text{Int}] \rightarrow \text{Int} \\
\text{sum} \ [\ ] & = 0 \\
\text{sum} (x:xs) & = x + \text{sum} \ xs
\end{align*}
\]

\[
\begin{align*}
\text{sum} \ [2,3,4] & \\
\Rightarrow & \quad \text{sum} \ (2:(3:(4:[])))) \\
\Rightarrow & \quad 2 + \text{sum} \ (3:(4:[]))) \\
\Rightarrow & \quad 2 + 3 + \text{sum} \ (4:[]) \\
\Rightarrow & \quad 2 + 3 + 4 + \text{sum} \ [] \\
\Rightarrow & \quad 2 + 3 + 4 + 0 \\
\Rightarrow & \quad 9
\end{align*}
\]
Describing computations

**Function definition**: a list of *equations* that relate inputs to output

**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**: Use equational reasoning to compute the reverse of the list \([2,3,4,5]\)
Outline

Haskell basics

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)
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*

\[
\text{map} :: (a \to b) \to [a] \to [b]
\]

\[
\text{map } f \; [] = []
\]

\[
\text{map } f \; (x:xs) = f \; x : \text{map } f \; xs
\]

\[
\text{map } f \; [2,3,4,5] = [f \; 2, f \; 3, f \; 4, f \; 5]
\]

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

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

\[
\text{foldr} :: (a\to b\to b) \to b \to [a] \to b
\]

\[
\text{foldr } f \; y \; [] = y
\]

\[
\text{foldr } f \; y \; (x:xs) = f \; x \; (\text{foldr } f \; y \; xs)
\]

\[
\text{foldr } (\to) \; 0 \; [2,3,4] = (\to) \; 2 \; ((\to) \; 3 \; ((\to) \; 4 \; 0)) = 2 + (3 + (4 + 0)) = 9
\]
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.

```haskell
plus :: Int -> Int -> Int
increment :: Int -> Int
increment = plus 1
```

Curried
```
plus 2 3
```

Uncurried
```
plus (2,3)
```

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

Haskell Curry
Outline

Haskell basics

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)
Lazy evaluation

In Haskell, expressions are reduced:

- only when needed
- at most once

Supports:

- infinite data structures
- separation of concerns

```haskell
nats :: [Int]
nats = 1 : map (+1) nats

fact :: Int -> Int
fact n = product (take n nats)

min3 :: [Int] -> [Int]
min3 = take 3 . sort
```

What is the running time of this function?
Outline

Haskell basics

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)
Functional programming workflow

Refactor

Define functions

Identify/define types

“obsessive compulsive refactoring disorder”
FP workflow (detailed)

1A. Data Description → 1B. Data Examples
   validated by

2. Function Description (Signature/Purpose/Header)
   names used in signature

3. Functional Examples
   inputs
   overlooked cases
   more
   guide writing

4. Function Template
   signature guides template
   demands

5. Code
   write body
   are also

6. Tests

7. Review & Refactor

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

Haskell basics

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)
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

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

Example: $2 + 3 + 4 \rightarrow \text{Plus \ (Lit \ 2) \ (Plus \ (Lit \ 3) \ (Lit \ 4))}$
Type parameters

(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

Haskell basics

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

\[ \text{sum} :: [\text{Int}] \rightarrow \text{Int} \]
\[ \text{sum} \ x_\text{s} = \begin{cases} 0 & \text{if null } x_\text{s} \text{ then } \theta \\ \text{head} \ x_\text{s} + \text{sum} (\text{tail} \ x_\text{s}) & \text{else} \end{cases} \]

Pattern matching

\[ \text{sum} :: [\text{Int}] \rightarrow \text{Int} \]
\[ \text{sum} \ [] = 0 \]
\[ \text{sum} \ (x:x_\text{s}) = x + \text{sum} \ x_\text{s} \]

(1) case analysis

(2) decomposition

Higher-order functions

\[ \text{sum} :: [\text{Int}] \rightarrow \text{Int} \]
\[ \text{sum} = \text{foldr} \ (+) \ 0 \]

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

If argument type is …

- **atomic type** (e.g. Int, Char)
  - apply functions to it
- **algebraic data type**
  - use pattern matching
    - case analysis
    - decompose into parts
- **function type**
  - apply it to something

If result type is …

- **atomic type**
  - output of another function
- **algebraic data type**
  - build with data constructor
- **function type**
  - function composition or partial application
  - build with lambda abstraction
Outline

Haskell basics

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

Use parentheses only to \textit{override} this behavior:
- $f\ (g\ x)$
- $f\ (x + y)$
Outline

Haskell basics

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)
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:
  \[ x \to f \ x \equiv f \]

- 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]
vary (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 = (*)