Introduction to Functional Programming in Haskell

Outline

Outline

- Why learn functional programming?
- The essence of functional programming
- How to functional program
- Refactoring and reuse
- Type inference

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**

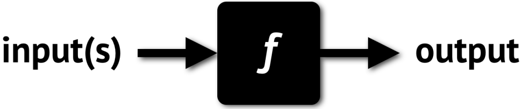
2. It will make you a better (imperative) programmer
   - forces you to think **recursively** and **compositionally**
   - forces you to **minimize use of state**

   …essential skills for solving **big** problems

3. It is the future!
   - more scalable and parallelizable (MapReduce)
   - functional features have been added to most mainstream languages

Outline

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:

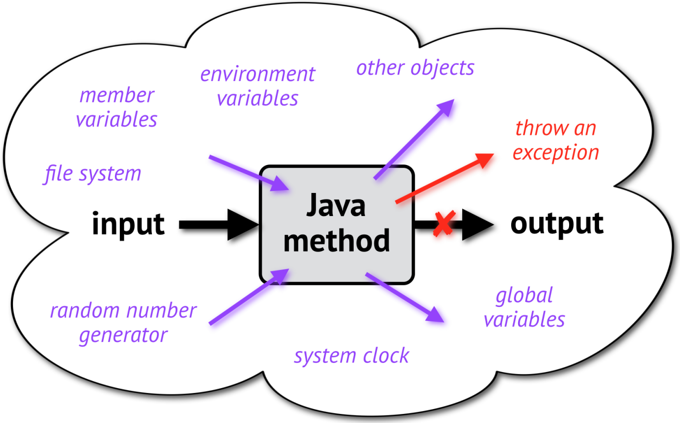
- mathematical functions $f(x) = x^2 + 2x + 3$
- encryption and compression algorithms

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

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

```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

a.k.a. **referent**

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

```
    length [1,2,3] + 4
⇒                3 + 4
```

what if `length` was a Java method?

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!

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

equations

```
    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? ✓

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

Exercise

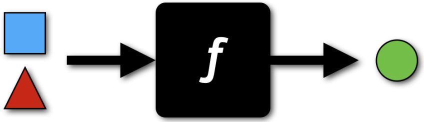
```
succ :: Int -> Int
succ x = x + 1

double :: Int -> Int
double x = x + x
```

Evaluate:

1. `double (succ (double 3))`
2. `(double . succ) 3`
3. `(succ . double) 3`

Outline

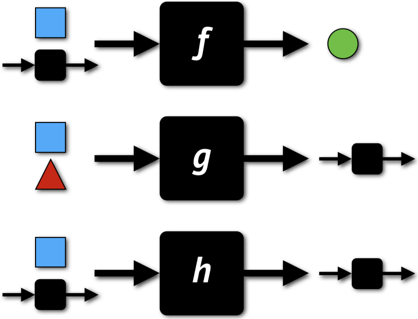
First-order functions



Examples

- `cos :: Float -> Float`
- `even :: Int -> Bool`
- `length :: [a] -> 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*

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

```
map f [2,3,4,5] = [f 2, f 3, f 4, f 5]
```

```
map even [2,3,4,5]
= [even 2, even 3, even 4, even 5]
= [True,False,True,False]
```

fold: *loop for aggregating elements in a list*

```
foldr :: (a->b->b) -> b -> [a] -> b
foldr f y []     = y
foldr f y (x:xs) = f x (foldr f y xs)
```

```
foldr f y [2,3,4] = f 2 (f 3 (f 4 y))
```

```
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

```
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)
```

```
(f . 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

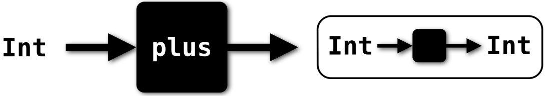
```
plus2  = succ . succ
odd    = not . even
second = head . tail
drop2  = tail . tail
```

Currying / partial application

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


Haskell Curry

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



Curried `plus 2 3`

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

Uncurried `plus (2,3)`

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

a pair of ints

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

Outline

Lazy evaluation

In Haskell, expressions are reduced:

- only when needed
- at most once

Supports:

- infinite data structures
- separation of concerns

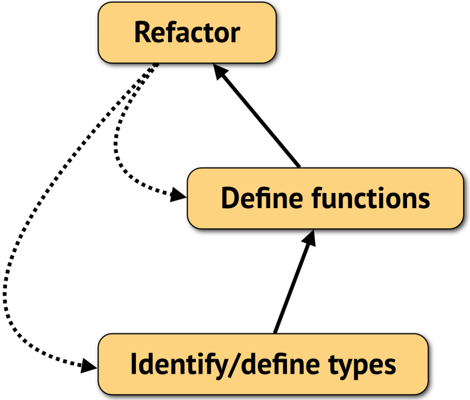
```
nats :: [Int]
nats = 1 : map (+1) nats

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

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

What is the running time of this function?

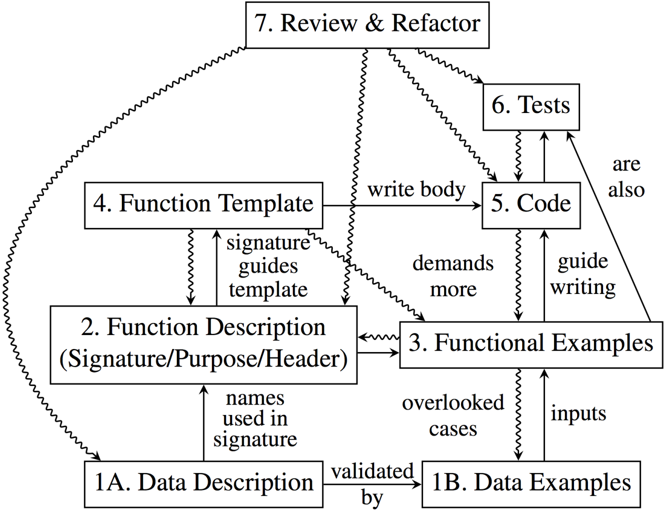
Outline

FP workflow (simple)



"obsessive compulsive refactoring disorder"

FP workflow (detailed)



Norman Ramsey, *On Teaching "How to Design Programs"*, ICFP'14

Outline

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

type name

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

cases

data constructor

types of arguments

Example: $2 + 3 + 4$ **`Plus (Lit 2) (Plus (Lit 3) (Lit 4))`**

FP data types vs. OO classes

Haskell

```
data Tree = Node Int Tree Tree
          | Leaf
```

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

Java

```
abstract class Tree { ... }
class Node extends Tree {
  int label;
  Tree left, right;
  ...
}
class Leaf extends Tree { ... }
```

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

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

Type parameters

type parameter

(Like generics in Java)

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

reference to type parameter

recursive reference to type

Specialized lists

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

Outline

Tools for defining functions

Recursion and other functions

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

Pattern matching

(1) case analysis

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

(2) decomposition

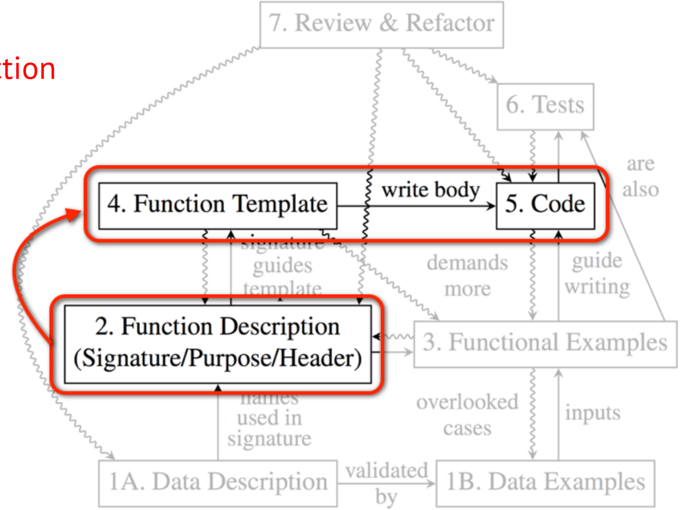
Higher-order functions

```
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**

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

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

Formatting function applications

Function application:

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

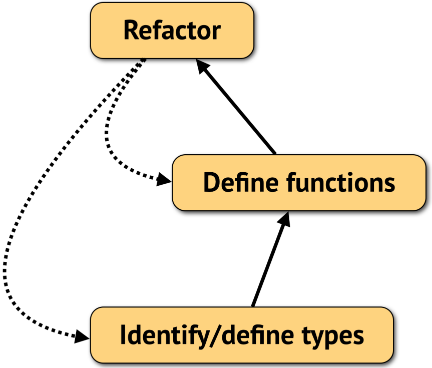
`f(x)`	`f x`
`(f x) y`	`f x y`
`(f x) + (g y)`	`f x + g y`

Use parentheses only to *override* this behavior:

- `f (g x)`
- `f (x + y)`

Outline

Refactoring in the FP workflow



"obsessive compulsive refactoring disorder"

Motivations:

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

Refactoring relations

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

- Eta reduction:

```
\x -> f x   ≡   f
```

- Map−map fusion:

```
map f . map g   ≡   map (f . g)
```

- Fold−map fusion:

```
foldr f b . map g   ≡   foldr (f . g) b
```

“Algebra of computer programs”

John Backus, *Can Programming be Liberated from the von Neumann Style?*, ACM Turing Award Lecture, 1978

Outline

What is a type class?

1. an **interface** that is supported by many different types
2. a **set of types** that have a common behavior

```haskell
class Eq a where
  (==) :: a -> a -> Bool
```

types whose values can be compared for equality

```haskell
class Show a where
  show :: a -> String
```

types whose values can be shown as strings

```haskell
class Num a where
  (+) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  ...
```

types whose values can be manipulated like numbers

Type constraints

```
class Eq a where
  (==) :: a -> a -> Bool
```

List elements can be of any type

```
length :: [a] -> Int
length []     = 0
length (_:xs) = 1 + length xs
```

List elements must support equality!

```
elem :: Eq a => a -> [a] -> Bool
elem _ []     = False
elem y (x:xs) = x == y || elem y xs
```

use method ⇒ add type class constraint

Outline

Type inference

How to perform type inference

If a literal, data constructor, or named function: write down the type – you're done!
Otherwise:

1. identify the top-level application $e_1\ e_2$
2. recursively infer their types $e_1 : T_1$ and $e_2 : T_2$
3. T_1 should be a function type $T_1 = T_{arg} \to T_{res}$
4. unify $T_{arg} =^? T_2$, yielding type variable assignment σ
5. return $e_1\ e_2 : \sigma T_{res}$ (T_{res} with type variables substituted)

If any of these steps fails, it is a **type error**!

Example: `map even`

Exercises

Given

```
data Maybe a = Nothing | Just a
gt   :: Int -> Int -> Bool                      not  :: Bool -> Bool
map  :: (a -> b) -> [a] -> [b]                  even :: Int -> Bool
(.)  :: (b -> c) -> (a -> b) -> a -> c
```

1. `Just`
2. `not even 3`
3. `not (even 3)`
4. `not . even`
5. `even . not`
6. `map (Just . even)`