The only valid measurement of code quality: WTFs/minute
Outline

• Good Haskell style
• What is refactoring?
• Strategies for refactoring
• Emphasizing function composition
Good Haskell style

Why it matters:
• layout is significant!
• expunging misconceptions
• we care about elegance

Easy stuff:
• use spaces (layout)
• align patterns and guards

See course web page for links to style guides
Formatting function applications

Function application . . .

• is just a space
• associates to the left
• binds most strongly

Use parentheses only to override this behavior:
Use pattern matching

pop :: [a] -> (a, [a])
pop xs = if not (null xs) then (head xs, tail xs) else error "empty"

pop :: [a] -> (a, [a])
pop xs = case xs of (y:ys) -> (y,ys) [] -> error "empty"

pop :: [a] -> (a, [a])
pop (x:xs) = (x, xs)
pop [] = error "empty"
Prefer pattern guards

```
elem :: Int -> Tree Int -> Bool
elem _ Leaf         = False
elem x (Node y l r) =
  if x == y then True
  else if x < y then elem l x
  else elem r x
```
Outline

• Good Haskell style

• **What is refactoring?**

• Strategies for refactoring

• Emphasizing function composition
Why refactor?

- make code easier to read and maintain
- generalize to new/related problems
- identify reusable components
- gain deeper insights
What is refactoring?

...a disciplined technique for restructuring existing code, altering its internal structure without changing its external behavior

— Martin Fowler

\[ \text{sum } xs = \begin{cases} 0 & \text{if } \text{null } xs \\ \text{head } xs + \text{sum } (\text{tail } xs) & \text{else} \end{cases} \]

\[ \text{sum } [] = 0 \]
\[ \text{sum } (x:xs) = x + \text{sum } xs \]
\[ \text{sum } = \text{foldr } (+) 0 \]
Refactoring relations

Laws that are the formal basis for refactoring

**Eta reduction**

\[ (\lambda x \to f\ x) \iff f \]

**Map fusion**

\[ \text{map } f \cdot \text{map } g \iff \text{map } (f \cdot g) \]

“Algebra of computer programs”

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

• Good Haskell style

• What is refactoring?

• **Strategies for refactoring**

• Emphasizing function composition
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

... then broaden the types

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

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

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

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

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

showResult = maybe "ERROR" show
getCommand = maybe Stay Move
addToMaybe x = maybe x (x+)
Notes on abstraction

**abstraction**: to separate a concept from its specific instances and make it reusable

Haskell has powerful tools for abstraction:

- **referential transparency**
  shared code can always safely be factored out

- **higher-order functions**
  can capture high-level patterns as functions

- **lazy evaluation**
  supports separation of concerns and definition of new control structures

- **type classes**
  describe common interface across many data types
Refactoring data types

```haskell
data Expr = Lit Int
          | Ref Var
          | Add Expr Expr
          | Sub Expr Expr
          | Mul Expr Expr

data Op = Add | Sub | Mul

simplifies writing many functions

...especially when we don't need to distinguish these cases

Factor out shared structure:

```

Outline

• Good Haskell style
• What is refactoring?
• Strategies for refactoring
• Emphasizing function composition
Function composition

\[( . ) :: (b \to c) \to (a \to b) \to a \to c\]

\[f \ (g \ x) \iff (f \ . \ g) \ x\]

\[f_1 \ (f_2 \ x \ (f_3 \ (f_4 \ y)))) \rightarrow (f_1 \ . \ f_2 \ x \ . \ f_3 \ . \ f_4) \ y\]

Advantages:

- emphasizes functions over results
- reveals opportunities for eta reduction (next slide)
Eta reduction

\[ (\lambda x \to f \ x) \iff f \]

```haskell
data Base = A | C | G | T deriving Show
type DNA = [Base]

showDNA :: DNA -> String
showDNA bs = concat (map (\b -> show b) bs)
```

**Eta reduction**

**Rewrite as composition**

\[ f \ (g \ x) \iff (f \ . \ g) \ x \]

**To lambda-notation**

\[ f \ x = y \iff f = \lambda x \to y \]

**Eta reduction**
Point-free style

Functions are defined:

- without referring to their arguments by name
- only by applying and composing other functions

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

showDNA :: DNA -> String
showDNA = concat . map show

topGrades :: [(Name, Grade)] -> [Name]
topGrades = map fst . filter ((>= 0.9) . snd)
```
Point-free tradeoffs

Advantages:

- emphasize functions over data
- what does this function do? vs. how does it do it?
- result of refactoring – often leads to insights
- shows off how clever you are :-)

But ... it's easy to get carried away – leads to obfuscation

\[
\text{flip \ flip \ snd \ . \ (ap \ .) \ . \ flip \ flip \ fst \ .} \\
\text{(((.) \ .) \ . \ flip \ . \ (((.) \ . (,)) \ .)}
\]

“pointless style”

vs. \[
(\text{\mathcal{f} \ g \ (x,y) \ -> \ (f \ x, \ g \ y)})
\]
Ordering arguments

Note that library functions are always:

- parameters first
- primary data structure last

\[
\begin{align*}
\text{map} & : (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\
\text{foldr} & : (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b \\
\text{maybe} & : b \rightarrow (a \rightarrow b) \rightarrow \text{Maybe } a \rightarrow a
\end{align*}
\]

Supports partial application and composition – you should do it too!

\[
\text{showMaybeInts} :: [\text{Maybe } \text{Int}] \rightarrow \text{String} \\
\text{showMaybeInts} = \text{concat . intersperse } "," \ . \text{map (maybe "" show)}
\]