Introduction to Functional Languages

1. Referential transparency, no side effects
   “substitution of equals for equals”

2. Function definitions can be used
   Suppose \( f \) is defined to be the function \((\text{fn } x \Rightarrow \text{exp})\), then \( f \) (\( \text{arg} \)) can be replaced by \( \text{exp}[x := \text{arg}] \)

3. Lists not arrays

4. Recursion not iteration

5. Universal parametric polymorphism, type reconstruction

6. Higher-order functions
   New idioms, total procedural abstraction
Barendredt, 2013, page xvii. The power of fun programming derives from:

- constant meaning (referential transparency)
- flexibility of high-order functions
- goal direction (no storage management)
• In a functional language an expression is the program plus its input.
• Expressions have parts which can be reduced $\triangle$

$$\triangle \rightarrow \blacktriangle$$

• Reduction continues until no more reducible parts exist
• The result corresponds to the output.
Schematic Representation of Reduction

△ △ △

▲ △ △

▲ ▲ △

▲ ▲ ▲
fun square x = x * x;
fun sos (x,y) = (square x) + (square y);

sos (3,4)
==> (square 3) + (square 4)  [Def’n of sos]
==> 3*3 + (square 4)  [Def’n of square]
==> 9 + (square 4)  [Def’n of *]
==> 9 + 4*4  [Def’n of square]
==> 9 + 16  [Def’n of *]
==> 25  [Def’n of +]
Language of expressions only, no statements.

fun test (x) = if x>20 then "big" else "small"

test (sos (3,4))
===> test(25)
===> if 25>20 then "big" else "small"
===> "big"
 Canonical Value

Canonical value. A *canonical value* is one which cannot be rewritten further. For example, $2+3$ is not canonical, it evaluates to 5; 5 is a canonical value. See *canonical* in the “The on-line hacker Jargon File,” version 4.4.7, 29 Dec 2003.
History of Functional Languages

1959  LISP: List processing, John McCarthy
1975  Scheme: MIT
1977  FP: John Backus
1980  Hope: Burstall, McQueen, Sannella
1984  COMMON LISP: Guy Steele
1985  ML: meta-language (of LCF), Robin Milner
1986  Miranda: Turner
1990  Haskell: Hudak & Wadler editors
# History of Lisp Dialects

<table>
<thead>
<tr>
<th>Year</th>
<th>LISP 1, 1.5, LISP 2 (abandoned)</th>
<th>Maclisp</th>
<th>Interlisp</th>
<th>Lisp Machine Lisp</th>
<th>Scheme</th>
<th>R5RS</th>
<th>R6RS</th>
<th>R7RS small</th>
<th>NIL</th>
<th>Common Lisp</th>
<th>Le Lisp</th>
<th>T</th>
<th>Emacs Lisp</th>
<th>AutoLISP</th>
<th>OpenLISP</th>
<th>PicoLisp</th>
<th>EuLisp</th>
<th>ISLISP</th>
<th>newLISP</th>
<th>Racket</th>
<th>GNU Guile</th>
<th>Visual LISP</th>
<th>Qi, Qiil</th>
<th>Shen</th>
<th>Clojure</th>
<th>Arc</th>
<th>LFE</th>
<th>Hy</th>
</tr>
</thead>
</table>
LAST NIGHT I DRIFTED OFF WHILE READING A LISP BOOK.

SUDDENLY, I WAS BATHED IN A SUFFUSION OF BLUE.

AT ONCE, JUST LIKE THEY SAID, I FELT A GREAT ENLIGHTENMENT. I SAW THE NAKED STRUCTURE OF LISP CODE UNFOLD BEFORE ME.

TRULY, THIS WAS THE LANGUAGE FROM WHICH THE GODS WROUGHT THE UNIVERSE.

IN MY GOD-OF-EVENTS, THE PATTERNS AND META-PATTERNS DANCED. SYNTAX DANCED, AND I SWAM IN THE PURITY OF QUANTIFIED CONCEPTION. OF IDEAS MANIFEST.

I MEAN, OSTEINSKY, YES. HONESTLY, WE HACKED MOST OF IT TOGETHER WITH PERL.

xkcd—a webcomic of romance, sarcasm, math, and language by Randall Munroe
Lazy Versus Eager

Lazy: don’t evaluate the function (constructor) arguments until needed (call-by-name), e.g., Haskell. Permits infinite data structures.
Eager: call-by-value, e.g., ML
Lazy versus Eager

A tutorial video
by the University of Glasgow

Call-by-need is a descriptive, but distinct variation of the call-by-name parameter passing mechanism.
Made key breakthroughs in disparate areas: proofs, languages, concurrency
Robin Milner (1934–2010)

John Robin Gorell Milner was educated at Eton College and Cambridge. He worked for a few years in Ferranti Ltd before joining the University of Edinburgh in 1973, becoming Professor of Computation Theory in 1984. In 1986, with colleagues, he founded the Laboratory for Foundations of Computer Science at Edinburgh. He was elected Fellow of the Royal Society in 1988, and in 1991 won the Turing Award.

Robin Milner was appointed Professor of Theoretical Computer Science at Cambridge University in 1995, and was Head of the Computer Laboratory there from January 1996 to October 1999. Before that he spent two years in the Artificial Intelligence Laboratory at Stanford University.
1991 Turing Award citation lists three achievements:

1. LCF, the mechanisation of Scott’s Logic of Computable Functions, probably the first theoretically based yet practical tool for machine-assisted proof construction;
2. ML, the first language to contain polymorphic type inference together with a type-safe exception handling mechanism;
3. CCS (Calculus of Communicating Systems), a general theory of concurrency.
Salient Features of SML

1. Strongly-typed, eager, functional language
2. Polymorphic types, type inference
3. Algebraic type definitions
4. Pattern matching function definitions
5. Exception handling
6. Module (signatures/structures) system
7. Interactive
Information about ML


ML and Haskell

- Similar to ML: functional, strongly-typed, algebraic data types, type inferencing
- Differences: no references, exception handling, or side effects of any kind; lazy evaluation, list comprehensions
Introduction to Haskell

1. Haskell (1.0) 1990
2. By 1997 four iterations of language design (1.4)
Salient Features of Haskell

1. Strongly-typed, lazy, functional language
2. Polymorphic types, type inference
3. Algebraic type definitions
4. Pattern matching function definitions
5. System of classes
6. Interactive
GHC Interactive, version 6.4.1, for Haskell 98.
http://www.haskell.org/ghc/
Type :? for help.

Loading package base-1.0 ... linking ... done.

Prelude> :load u:main
Compiling Main
Ok, modules loaded: Main.
*Main> f "abcdefg"
"gfedcba"
*Main> :reload
Ok, modules loaded: Main.
*Main> :quit

module Main where

import System.IO as IO

-- main program to reverse each line of input
main =
do IO.getContents >>= IO.putStrLn . unlines . map f . lines

f line = reverse line

{-
  built-in functions:
  -- unlines :: [String] -> String: breaks input into separate lines
  -- lines :: String -> [String]: combines separate lines into one string
  -- reverse :: [a] -> [a]: reverse list back to front
-}
Information about Haskell


Hutton, Graham, *Programming in Haskell.*
O’Donnell et al., *Discrete Mathematics Using a Computer.*
Haskell

- Similar to ML: functional, strongly-typed, algebraic data types, type inferencing
- Differences: no references, exception handling, or side effects of any kind; lazy evaluation, list comprehensions

```haskell
fac n = if n==0 then 1 else n * fac (n-1)

data Tree = Leaf | Node (Tree, String, Tree)

size (Leaf) = 1
size (Node (l,_,r)) = size (l) + size (r)

squares = [ n*n | n <- [0..] ]
pascal = iterate (\row->zipWith (+) ([0]++row) (row++[0])) [1]
```
Section Outline

1. Lists
2. Patterns
3. Higher-Order Functions
4. Function Spaces, Currying
5. Common Patterns
6. Haskell Data Structures
Cons 1 (Cons 2 Nil)

1 : 2 : Nil

[1, 2]
\[ [1, 2, 3, 4] \]
Section Outline

1. Lists
2. Patterns
3. Higher-Order Functions
4. Function Spaces, Currying
5. Common Patterns
6. Haskell Data Structures
Patterns are a very natural way of expressing complex choices. Consider the code to re-balance red-black trees. This is usually quite complex to express in a programming language. But with patterns it can be more concise. Notice that constructors of user-defined types (line `RBTree`) as well as pre-defined types (like list) can be used in patterns.
data Color = R | B deriving (Show, Read)
data RBTree a = Empty | T Color (RBTree a) a (RBTree a)
deriving (Show, Read)

balance :: RBTree a -> a -> RBTree a -> RBTree a
balance (T R a x b) y (T R c z d) = T R (T B a x b) y (T B c z d)
balance (T R (T R a x b) y c) z d = T R (T B a x b) y (T B c z d)
balance (T R a x (T R b y c)) z d = T R (T B a x b) y (T B c z d)
balance a x (T R b y (T R c z d)) = T R (T B a x b) y (T B c z d)
balance a x (T R (T R b y c) z d) = T R (T B a x b) y (T B c z d)
balance a x b = T B a x b
Read the On-Line Tutorial
Learn You a Haskell For Great Good
Functions

\[ x \rightarrow 2 \times x \]
A function defined on a type can only be called by constructing the type (with all its elements) and then apply the functions. A curried function is more flexible. Apply the function to each argument individually is possible and many times the result are useful.

```
add3 (x, y, z) = x + y + z + 1
add3' x y z = x + y + z + 1
```

```
add3 (1, 2, 3)
ad3 1 2 3
add3 1 2
add3 1
```
Section Outline

1. Lists
2. Patterns
3. Higher-Order Functions
4. Function Spaces, Currying
5. Common Patterns
6. Haskell Data Structures
Higher-Order Functions

A function is called *higher-order* if it takes a function as an argument or returns a function as a result.

```
twice :: (a -> a) -> a -> a
twice f x = f (f x)
twice = \ f x = f (f x)
twice = \ f -> \ x -> f (f x)
```

```
let twice f x = f (f x);
let twice = fun f x -> f (f x);
let twice = fun f -> fun x -> f (f x);
```

twice is higher-order because it takes a function as an argument.
Higher-order functions are useful

- “by allowing common programming patterns to be encapsulated as functions” [Hutton, page 74]
- “to define domain-specific languages within Haskell” [Hutton, page 74]
- properties of function can be used in proof of correctness
Higher-Order Functions

\[ f_1 = \lambda x \rightarrow \lambda y \rightarrow x + y + 4 \]
\[ f_2 = \lambda x \ y \rightarrow x + y + 4 \]
\[ f_3 \ x = \lambda y \rightarrow x + y + 4 \]
\[ f_4 \ x \ y = x + y + 4 \]

-- pleasing and familiar symmetry of definition and use

\[ f_4 \ 3 \ 4 \]
Higher-Order Functions

\( \text{con1} = \lambda x \rightarrow \lambda y \rightarrow x \)
\( \text{con2} = \lambda x \, y \rightarrow x \)
\( \text{con3} \, x = \lambda y . \, x \)
\( \text{con4} \, x \, y = x \)

\( \text{app1} = \lambda f \rightarrow \lambda x \rightarrow f \, x \)
\( \text{app2} = \lambda f \, x \rightarrow f \, x \)
\( \text{app3} \, f = \lambda x \rightarrow f \, x \)
\( \text{app4} \, f \, x = f \, x \)
Partial Application

Any curried function may be called with fewer arguments than it was defined for. The result is a function of the remaining arguments. If $f$ is a function $\text{Int} \to \text{Bool} \to \text{Int} \to \text{Bool}$, then

\begin{align*}
f & : \text{Int} \to \text{Bool} \to \text{Int} \to \text{Bool} \\
f\ 2 & : \text{Bool} \to \text{Int} \to \text{Bool} \\
f\ 2\ \text{True} & : \text{Int} \to \text{Bool} \\
f\ 2\ \text{True}\ 3 & : \text{Bool}
\end{align*}
Section Outline

1. Lists
2. Patterns
3. Higher-Order Functions
4. Function Spaces, Currying
5. Common Patterns
6. Haskell Data Structures
Haskell Brooks Curry (1900–1982)

Haskell Curry was born on 12 September 1900 in Mills, Massachusetts. His father, Samuel Silas Curry, was the founder of Curry College in Boston. Curry was supervised at the University of Göttingen, Germany, by David Hilbert and worked closely with Paul Bernays, receiving a Ph.D. in 1930 with a dissertation on combinatory logic. He joined Penn State faculty as an assistant professor in 1929. He wrote several books including *Combinatory Logic and Foundations of Mathematical Logic*. Curry was an avid bird watcher, having documented nearly 150,000 separate birds over a fifty year span. He died 1 September 1982 in Pennsylvania.
Examine currying first; before higher-order, recursive list patterns (map, filter, foldl and foldr).
Every function take one argument. Every function $f : D \rightarrow B$ has one domain and one range.
Currying

\[ A \times B \rightarrow C \simeq A \rightarrow B \rightarrow C \]

The function space \( A \times B \rightarrow C \) is isomorphic to the function space \( A \rightarrow B \rightarrow C \).

\[
\begin{align*}
\text{curry} & \quad f \ x \ y \quad = \quad f \ (x,y) \\
\text{uncurry} & \quad g \ (x,y) \quad = \quad g \ x \ y
\end{align*}
\]
Currying

These functions can be written more explicitly without the special Haskell function declaration notation.

\[
\begin{align*}
\text{curry } f \ x \ y &= f(x,y) \\
\text{uncurry } g (x,y) &= g \ x \ y
\end{align*}
\]

\[
\begin{align*}
\text{curry } f &= \ \backslash \ x \ \rightarrow \ \backslash \ y \ \rightarrow \ f(x,y) \\
\text{uncurry } g &= \ \backslash \ (x,y) \ \rightarrow \ g \ x \ y
\end{align*}
\]

\[
\begin{align*}
\text{curry } &= \ \backslash f \ \rightarrow \ \backslash \ x \ \rightarrow \ \backslash \ y \ \rightarrow \ f(x,y) \\
\text{uncurry } &= \ \backslash g \ \rightarrow \ \backslash (x,y) \ \rightarrow \ g \ x \ y
\end{align*}
\]
A careful derivation of the types of functions follows.

\[ \text{curry} :: ((a,b)\rightarrow c) \rightarrow a \rightarrow b \rightarrow c \]
\[ \text{curry} = \lambda f \rightarrow \lambda x \rightarrow \lambda y \rightarrow f(x,y) \]

\[ \text{uncurry} :: (a \rightarrow b \rightarrow c) \rightarrow (a,b)\rightarrow c \]
\[ \text{uncurry} = \lambda g \rightarrow \lambda (x,y) \rightarrow g x y \]
curry :: ((a,b)->c) -> a -> b -> c
curry = \f -> \ x -> \ y -> f (x,y)

curry :: [ \f -> \ x -> \ y -> f (x,y) ] :: A
  -- A = B->C
curry :: [\f::B] -> [ \ x -> \ y -> f (x,y) ] :: C
  -- C = D->E
curry :: [\f::B] -> ( [\x::D] -> [ \ y -> f (x,y) ] :: E )
  -- E = F->G
curry :: [\f::B] -> ( [\x::D] -> ( [\y::F] -> [f (x,y)] :: E )
  -- B = H->G
curry :: [\f::H->G] -> ( [\x::D] ->( [\y::F] ->[f(x,y)] :: G
  -- H = D*F
curry :: [\f::D*F->G] -> ( [\x::D] ->( [\y::F] ->[f(x,y)] :: G
  -- A = B->C = B->D->E = B->D->F->G
  -- = (H->G)->D->F->G = (D*F->G)->D->F->G
uncurry :: (a -> b -> c) -> (a,b)-> c
uncurry = \g -> \ (x,y) -> g x y

uncurry :: [ \g -> \ (x,y) -> g x y ] :: A
  -- A = B->C
uncurry :: [\g::B] -> [ \ (x,y) -> g x y ] :: C
  -- C = D*E->F
uncurry :: [\g::B] -> ( [\ (x,y)::D*E] -> [ g x y ] :: F )
  -- B = D->G
uncurry :: [\g::D->G] -> ( [\(x,y)::D*E] -> [ g x y ] :: F )
  -- G = E->F
uncurry :: [\g::D->E->F] -> ( [\(x,y)::D*E] -> [ g x y ] :: F )
  -- A = B->C = B->D*E->F = (D->G)->D*E->F = (D->E->F)->D*E->F
Parameters A B C : Set.

Definition curry (f : A \times B \to C) := fun a => fun b => f (a, b).
Definition uncurry (g : A \to B \to C) := fun p => g (fst p) (snd p).

(* Prove that uncurry (curry f) is extensionally equivalent to f. *)
Theorem Left_Inverse : \forall f a b, uncurry (curry f) (a, b) = f (a, b).
Proof.
intros.
unfold curry, uncurry.
simpl. (* fst(a,b)=a; snd(a,b)=b *)
reflexivity. (* f(a,b)=f(a,b) *)
Qed.

(* Prove that curry (uncurry g) is extensionally equivalent to g. *)
Theorem Right_Inverse : \forall g a b, curry (uncurry g) a b = g a b.
Proof.
intros.
unfold curry, uncurry.
simpl.
reflexivity.
Qed.
\textbf{Definition} Isomorphic \((X \ Y : \text{Set}) := \)
\begin{align*}
\exists f : X &\to Y, \\
\exists g : Y &\to X, \\
(\forall x, g(f(x)) = x) &\land (\forall y, f(g(y)) = y).
\end{align*}

\textbf{Axiom} Extensionality : \(\forall (X \ Y : \text{Set}) (f \ g : X \to Y), \)
\begin{align*}
(\forall x, f(x) = g(x)) &\to f = g.
\end{align*}

\textbf{Theorem} IsoFunSpace : Isomorphic \((A \times B \to C)(A \to B \to C).\)
\textbf{Proof.}
\begin{align*}
exists \ curry, \ uncurry. \\
\text{split.} \\
\text{apply Extensionality. intros [a b].} \\
\text{apply Left_Inverse. intro g.} \\
\text{apply Extensionality. intro a.} \\
\text{apply Extensionality. intro b.} \\
\text{apply Right_Inverse.}
\end{align*}
\textbf{Qed.}
Common Patterns

Two of the most common list patterns are higher-order.

1. map
2. fold (aka reduce)
Also filter.
map pattern
filter pattern
fold left pattern
fold right pattern
Haskell Fold

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

\[
\text{foldl} :: (a \rightarrow b \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow a
\]
\[
\text{foldl}\ f\ z\ [] = z
\]
\[
\text{foldl}\ f\ z\ (x:xs) = \text{foldl}\ f\ (f\ z\ x)\ xs
\]

\[
\text{foldl}' :: (a \rightarrow b \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow a
\]
\[
\text{foldl}'\ f\ z0\ xs = \text{foldr}\ f'\ \text{id}\ xs\ z0
\]
\[
\text{where}\ f'\ x\ k\ z = k\ !\ f\ z\ x
\]

foldl is tail recursive!

[Real World Haskell says never use foldl instead use foldl'.]
Haskell Fold

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

foldk k f z [] = k z
        foldk k f z (x:xs) = foldk (k.(\ v -> f x v)) f z xs

foldr = foldk id
Haskell Fold

Evaluates its first argument to head normal form, and then returns its second argument as the result.

\[
\text{seq} :: \text{a} \to \text{b} \to \text{b}
\]

Strict (call-by-value) application, defined in terms of 'seq'.

\[
(\$!) :: (\text{a} \to \text{b}) \to \text{a} \to \text{b}
f \$! x = x \text{ 'seq' } f x
\]
Haskell Fold

“A tutorial on the universality and expressiveness of fold” by Graham Hutton (J. Fun Prog, 1999)

Wikipedia: Fold (higher order function)
> foldr (\x y -> concat ["("",x","","+","",y","")"] ) "0" (map show [1..13])
"(1+(2+(3+(4+(5+(6+(7+(8+(9+(10+(11+(12+(13+0))))))))))))"'

> foldl (\x y -> concat ["("",x","","+","",y","")"] ) "0" (map show [1..13])
"((((((0+1)+2)+3)+4)+5)+6)+7)+8)+9)+10)+11)+12)+13)"
Fold

\[\text{foldr } \otimes \ z_r \ [x_1, x_2, \ldots, x_n] = x_1 \otimes (x_2 \otimes (\ldots(x_n \otimes z_r)\ldots))\]

\[\text{foldr } f \ z \ [x_1, x_2, \ldots, x_n] = x_1 \ 'f' \ (x_2 \ 'f' \ (\ldots(x_n \ 'f' \ z)\ldots))\]

\[\text{foldr } f \ z \ [x_1, x_2, \ldots, x_n] = f \ x_1 \ (f \ x_2 \ (\ldots(f \ x_n \ z)\ldots))\]
Fold

\[ \text{fold} \otimes z \ [x_1, x_2, \ldots, x_n] = (\ldots((z \otimes x_1) \otimes x_2) \ldots) \otimes x_n \]

\[ \text{fold} f z \ [x_1, x_2, \ldots, x_n] = (\ldots((z \ (f \ x_1) \ (f \ x_2)) \ldots) \ (f \ x_n) \]

\[ \text{fold} f z \ [x_1, x_2, \ldots, x_n] = f (\ldots(f (f z x_1) x_2) \ldots) x_n \]
One important thing to note in the presence of lazy, or normal-order evaluation, is that foldr will immediately return the application of f to the recursive case of folding over the rest of the list. Thus, if f is able to produce some part of its result without reference to the recursive case, and the rest of the result is never demanded, then the recursion will stop. This allows right folds to operate on infinite lists. By contrast, foldl will immediately call itself with new parameters until it reaches the end of the list. This tail recursion can be efficiently compiled as a loop, but can’t deal with infinite lists at all – it will recurse forever in an infinite loop.
Another technical point to be aware of in the case of left folds in a normal-order evaluation language is that the new initial parameter is not being evaluated before the recursive call is made. This can lead to stack overflows when one reaches the end of the list and tries to evaluate the resulting gigantic expression. For this reason, such languages often provide a stricter variant of left folding which forces the evaluation of the initial parameter before making the recursive call, in Haskell, this is the foldl’ (note the apostrophe) function in the Data.List library. Combined with the speed of tail recursion, such folds are very efficient when lazy evaluation of the final result is impossible or undesirable.
Haskell Fold (Associative Operations)

\[
\begin{align*}
\text{sumR} &= \text{foldr} \ (\ + \ ) \ 0 \\
\text{sumL} &= \text{foldl} \ (\ + \ ) \ 0 \\
\text{productR} &= \text{foldl} \ (\ \ast \ ) \ 1 \\
\text{productL} &= \text{foldl} \ (\ \ast \ ) \ 1 \\
\text{andR} &= \text{foldl} \ (\ &&) \ \text{True} \\
\text{andL} &= \text{foldl} \ (\ &&) \ \text{True} \\
\text{orR} &= \text{foldr} \ (\ || \ ) \ \text{False} \\
\text{orL} &= \text{foldl} \ (\ || \ ) \ \text{False} \\
\text{concatR} &= \text{foldl} \ (\ ++ \ ) \ [] \\
\text{concatL} &= \text{foldl} \ (\ ++ \ ) \ [] \\
\text{unionsR} &= \text{foldl} \ \text{Set.union} \ \text{Set.empty} \\
\text{unionsL} &= \text{foldl} \ \text{Set.union} \ \text{Set.empty} \\
\text{composeR} &= \text{foldr} \ (\ . \ ) \ \text{id} \\
\text{composeL} &= \text{foldl} \ (\ . \ ) \ \text{id}
\end{align*}
\]
Haskell Fold

\[
\text{lengthR} = \text{foldr} \ (\text{const} \ (+1)) \ 0 \\
\text{lengthL} = \text{foldl} \ (\text{const} \ . \ (+1)) \ 0
\]

\[
\text{idR} = \text{foldr} \ (:) \ [] \\
\text{idL} = \text{foldl} \ (\lambda x s \to \text{x} + [\text{x}]) \ [] = \text{foldl} \ \text{snoc} \ [] \\
\]

\[
\text{appendR} = \text{foldr} \ (:) = \text{foldr} \ (\lambda y l \to \text{y} : \text{l}) \\
\text{appendL} \ \text{x} = \text{foldl} \ {- \ imposible -} \ {- \ imposible -} \\
\]

\[
\text{reverseR} = \text{foldr} \ (((\text{flip} \ (+)))([])) = \text{foldr} \ (\lambda x s \to \text{x} + s) \\
\text{reverseL} = \text{foldl} \ (\text{flip} \ (:)) \ [] = \text{foldl} \ (\lambda x s \to \text{x} : \text{s}) \\
\]

\[
\text{elemR} \ \text{a} = \text{foldr} \ (\lambda x r \to (\text{compare} \ \text{a} \ x = \text{EQ}) | | \ r) \ \text{False} \\
\text{elemL} \ \text{a} = \text{foldl} \ (\lambda r x \to (\text{compare} \ \text{a} \ x = \text{EQ}) | | \ r) \ \text{False}
\]
lengthL ['ab', '', 'def']
foldl (const . (+1)) 0 ['ab', '', 'def']
  (const .(+1)) ((const .(+1)) ((const .(+1)) 0 "ab") '') "def"
  (const .(+1)) ((const .(+1)) (const 1 "ab") '') "def"
  (const .(+1)) (const .(+1)) 1 "" "def"
  (const .(+1)) (const 2 "") "def"
  (const .(+1)) 2 "def"
const 3 "def"
3
Haskell Length Using Fold Right

```
lengthR ["ab", "", "def"]
foldr (const (+1)) 0 ["ab", "", "def"]
(const (+1)) "ab" ((const (+1)) "" (const (+1) "def" 0))
(+1) ((const (+1)) "" (const (+1) "def" 0))
(+1) ((+1) (const (+1) "def" 0))
(+1) ((+1) (+1) 0))
(+1) ((+1) 1)
(+1) 2
3
```
Haskell Reverse Using Fold Left

\[ reverseL \ [1,3,5,7] \]
\[ foldl (flip (:)) [] [1,3,5,7] \]

\[ flip(\cdot) ((flip(\cdot)) ((flip(\cdot)) [1] 3) 5) 7 \]
\[ flip(\cdot) ((flip(\cdot)) [3,1] 5) 7 \]
\[ flip(\cdot) [5,3,1] 7 \]
\[ [7,5,3,1] \]
Haskell Reverse Using Fold Right

reverseR [1,3,5,7]
foldR (f) [] [1,3,5,7]
  (f) 1 ((f) 3 ((f) 5 ((f) 7 [])))
  (++[1]) (++[3]) (++[5]) (++[7]) []
  (++[1]) (++[3]) (++[5]) [7]
  (++[1]) [7,5]
  [7,5,3,1]
Haskell Fold

reverse = foldl (\ xs x -> xs ++ [x]) []
map f = foldl (\ xs x -> f x : xs) []
filter p = foldl (\ xs x -> if p x then x:xs else xs) []
Haskell Fold

If this is your pattern

\[
\begin{align*}
g \; [ ] &= v \\
g \; (x:xs) &= f \; x \; (g \; xs)
\end{align*}
\]

then

\[
g = \text{foldr} \; f \; v
\]
Haskell Fold

Left fold is tail recursive. Can one make right fold tail recursive? Solve the equation

\[ \text{foldr} = \text{foldTR id} \]

We need

\[
\begin{align*}
\text{foldTR } k \ f \ z \ [] &= \{- \ldots -\} \\
\text{foldTR } k \ f \ z \ (x:xs) &= \text{foldTR } k' \ f \ z \ xs \\
\text{where} \\
k' \ v &= \{- \ldots -\}
\end{align*}
\]
Haskell Fold

Left fold is tail recursive. Can one make right fold tail recursive? Solve the equation

\[ \text{foldr} = \text{foldTR \ id} \]

We need

\[
\begin{align*}
\text{foldTR } k \ f \ z \ [] &= \{- \ldots -\} \\
\text{foldTR } k \ f \ z \ (x:xs) &= \text{foldTR } k' \ f \ z \ xs \\
\text{where} \\
&\quad k' \ v = \{- \ldots -\}
\end{align*}
\]

\[
\begin{align*}
\text{foldTR } k \ f \ z \ [] &= k \ z \\
\text{foldTR } k \ f \ z \ (x:xs) &= \text{foldTR } k' \ f \ z \ xs \\
\text{where} \\
&\quad k' \ v = k \ (f \ x \ v)
\end{align*}
\]
Section Outline

1. Lists
2. Patterns
3. Higher-Order Functions
4. Function Spaces, Currying
5. Common Patterns
6. Haskell Data Structures
Haskell Data Structures

See haskell_data.tex
next :: (Enum a, Bounded a, Eq a) => a -> a
next x | x == maxBound = minBound
        | otherwise = succ x
Haskell Classes

data Triangle = Triangle
data Square = Square
data Octagon = Octagon

class Shape s where
  sides :: s -> Integer

instance Shape Triangle where
  sides _ = 3

instance Shape Square where
  sides _ = 4

instance Shape Octagon where
  sides _ = 8