The essence of functional programming

Beginning to learn functional programming

Resources, advice and bottlenecks
THE ESSENCE OF FUNCTIONAL PROGRAMMING
higher-order functions

pattern matching
data types

lambda

recursion
fun
Model the world as data

+ Functions over the data

+ Functions as data
And when I say “function” . . .

I mean in the mathematical sense, taking inputs to outputs, and doing nothing else!
The approach underlies lots of examples.
The approach underlies lots of examples, libraries, laziness, monads, lenses ...
Rock–Paper–Scissors
We choose what to play, depending on the history of all your moves.
We choose what to play, depending on the history of all your moves.

```haskell
data Move = Rock | Paper | Scissors

beat :: [Move] -> Move

beat [] = Rock
beat (x:xs) = case x of
  Rock  -> Scissors
  Paper -> Rock
  Scissors -> Paper
```
Parsers
Side-effects
store before

input

store after

result
Delay/force . . . streams
cons(X,Xs) ->
    fun() -> {X,Xs} end.

head(L) ->
    case (L()) of
    {H,_} -> H
    end.

tail(L) ->
    case (L()) of
    {_,T} -> T
    end.
define(cons(X,Xs),
    fun() -> {X,Xs} end).

head(L) ->
    case (L()) of
        {H,_} -> H
    end.

tail(L) ->
    case (L()) of
        {_,T} -> T
    end.
github.com/simonjohntompson/streams

github.com/simonjohntompson/interaction
type
Functions give us expressivity + Types help to constrain that + Give a language for modelling
BEGINNING
FUNCTIONAL
PROGRAMMING
higher-order functions

data

types

lambdas
higher-order functions

pattern matching
data types

lambdas

recursion
Pick any language

Start with the concrete before going to complex abstractions.
PATTERN MATCHING
type Point = (Float, Float)

data Shape = Circle Point Float
           | Rectangle Point Float Float

area :: Shape -> Float

area (Circle _ r) = pi*r*r
area (Rectangle _ h w) = h*w
type Point = (Float, Float)

data Shape = Circle Point Float
              | Rectangle Point Float Float

area :: Shape -> Float

area (Circle _ r) = pi * r * r
area (Rectangle _ h w) = h * w
type Point = (Float, Float)
data Shape = Circle Point Float
   | Rectangle Point Float Float

area :: Shape -> Float

area (Circle _ r) = pi*r*r
area (Rectangle _ h w) = h*w
PATTERN MATCHING!
area :: Shape -> Float

area (Circle _ r) = pi*r*r
area (Rectangle _ h w) = h*w
area :: Shape -> Float

area (Circle _ r) = pi * r * r
area (Rectangle _ h w) = h * w

Link to something more familiar
area :: Shape -> Float

area (Circle _ r) = \pi*r*r
area (Rectangle _ h w) = h*w

area shape =
  if is_circle shape
    then \pi*(radius shape)*(radius shape)
    else height shape * width shape
area :: Shape -> Float

area (Circle _ r) = pi*r*r
area (Rectangle _ h w) = h*w

area shape =
  if is_circle shape
    then pi*(radius shape)*(radius shape)
    else height shape * width shape

is_circle :: Shape -> Bool

is_circle (Circle _) = True
is_circle _ = False

radius, height, width :: Shape -> Float

radius (Circle _ r) = r
height (Rectangle _ h _) = h
width (Rectangle _ _ w) = w
RECURSION
The nub of a list is the list with all duplicates removed.

\[
\text{nub} :: \text{Eq } a \Rightarrow [a] \rightarrow [a]
\]
\[
\text{nub } [] = []
\]
\[
\text{nub}(x:xs) =
\]
\[
\text{ if } \text{elem } x \text{ xs }
\]
\[
\text{ then } \text{nub } xs
\]
\[
\text{ else } x : \text{nub } xs
\]
The nub of a list is the list with all duplicates removed.

\[
\text{nub}([\]) \rightarrow []; \\
\text{nub}([X|Xs]) \rightarrow [X|\text{nub}(\text{remove}(X,Xs))].
\]
The nub of a list is the list with all duplicates removed.

\[
\begin{align*}
nub([]) & \rightarrow []; \\
nub([X|Xs]) & \rightarrow [X|nub(remove(X,Xs))]. \\
remove(_,[]) & \rightarrow []; \\
remove(X,[X|Xs]) & \rightarrow remove(X,Xs); \\
remove(X,[Y|Xs]) & \rightarrow [Y|remove(X,Xs)].
\end{align*}
\]
RECURSION!
The nub of a list is the list with all duplicates removed.

Generate all the answers?

\[
\text{nub} :: \mathbf{Eq} \ a \Rightarrow [a] \rightarrow [a]
\]

\[
\text{nub} \ [\] = []
\]

\[
\text{nub}(x:xs) =
\begin{align*}
\text{if} \ & \ \text{elem} \ x \ xs \\
\text{then} \ & \ \text{nub} \ xs \\
\text{else} \ & \ x : \ \text{nub} \ xs
\end{align*}
\]
The nub of a list is the list with all duplicates removed.

Generate all the answers?

\[
\begin{align*}
nub \; [] & = [] \\
nub \; [1] & = \text{nub} \; (1:[]) = 1 : \text{nub} \; [] = [1]
\end{align*}
\]
The nub of a list is the list with all duplicates removed.

nub :: Eq a => [a] -> [a]

nub [] = []
nub(x:xs) = 
  if elem x xs
    then nub xs
    else x : nub xs

Generate all the answers?

nub [] = []
nub [1] = nub (1:[]) = 1 : nub [] = [1]
nub [2,1] = nub (2:[1]) = 2 : nub [1] = [2,1]
The nub of a list is the list with all duplicates removed.

Generate all the answers?

\[
\text{nub} \ :\ \text{Eq}\ a\Rightarrow\ [a] \rightarrow\ [a] \\
\text{nub}\ [\ ] = [\ ] \\
\text{nub}(x:xs) = \\
\quad\text{if elem}\ x\ xs \\
\quad\quad\text{then}\ \text{nub}\ xs \\
\quad\quad\text{else}\ x : \text{nub}\ xs \\
\text{nub} [\ ] = [\ ] \\
\text{nub} [1] = \text{nub}(1:[\ ]) = 1 : \text{nub} [\ ] = [1] \\
\text{nub} [2,1] = \text{nub}(2:[1]) = 2 : \text{nub} [1] = [2,1] \\
\text{nub} [1,2,1] = \text{nub}(1:[2,1]) = \text{nub} [2,1] = [2,1]
\]
The nub of a list is the list with all duplicates removed.

Rewrite … work “top down”

\[ nub \text{ :: } Eq \ a \Rightarrow [a] \rightarrow [a] \]

\[ nub \ ] = [ ] \]

\[ nub(x:xs) = \]
\[ \quad \text{if} \ elem \ x \ xs \]
\[ \quad \text{then} \ nub \ xs \]
\[ \quad \text{else} \ x : \ nub \ xs \]

\[ nub \ [1,2,1] \]
The nub of a list is the list with all duplicates removed.

Rewrite ... work “top down”

nub :: Eq a => [a] -> [a]
nub [] = []
nub(x:xs) = 
    if elem x xs 
    then nub xs 
    else x : nub xs
The nub of a list is the list with all duplicates removed.

nub :: Eq a => [a] -> [a]
nub [] = []
nub(x:xs) =
    if elem x xs
    then nub xs
    else x : nub xs

Rewrite ... work “top down”

nub [1,2,1]
= nub (1:[2,1])
= nub [2,1]
The nub of a list is the list with all duplicates removed.

Rewrite ... work “top down”

\[
\text{nub} \ [1,2,1] \\
= \text{nub} \ (1: [2,1]) \\
= \text{nub} \ [2,1] \\
= \text{nub} \ (2:[1])
\]
The nub of a list is the list with all duplicates removed.

Rewrite ... work “top down”

```
nub :: Eq a => [a] --> [a]
nub [] = []
nub(x:xs) =
  if elem x xs
    then nub xs
    else x : nub xs
```

```
nub [1,2,1]
= nub (1:[2,1])
= nub [2,1]
= nub (2:[1])
= 2 : nub [1]
```
The nub of a list is the list with all duplicates removed.

\[ \text{nub} :: \text{Eq } a \Rightarrow [a] \to [a] \]

\[ \text{nub} \[\] = [\] \]

\[ \text{nub}(x:xs) = \]
\[ \quad \text{if} \ \text{elem } x \ \text{xs} \]
\[ \quad \quad \text{then } \text{nub} \ \text{xs} \]
\[ \quad \quad \text{else } x : \text{nub} \ \text{xs} \]

Rewrite ... work “top down”

\[ \text{nub} [1,2,1] \]
\[ = \text{nub} (1:[2,1]) \]
\[ = \text{nub} [2,1] \]
\[ = \text{nub} (2:[1]) \]
\[ = 2 : \text{nub} [1] \]
\[ = 2 : \text{nub} (1:[]) \]
The nub of a list is the list with all duplicates removed.

nub :: Eq a => [a] -> [a]
nub [] = []
nub(x:xs) =
  if elem x xs
  then nub xs
  else x : nub xs

nub [1,2,1]
= nub (1:[2,1])
= nub [2,1]
= nub (2:[])
= 2 : nub [1]
= 2 : nub (1:[])
= 2 : 1 : nub []

Rewrite ... work “top down”
The nub of a list is the list with all duplicates removed.

Rewrite … work “top down”

\[
\begin{align*}
nub \ [1,2,1] &= nub \ (1:\ [2,1]) \\
&= nub \ [2,1] \\
&= nub \ (2:\ [1]) \\
&= 2 : nub \ [1] \\
&= 2 : nub \ (1:\ []) \\
&= 2 : 1 : nub \ [] \\
&= 2 : 1 : []
\end{align*}
\]
The nub of a list is the list with all duplicates removed.

Rewrite ... work “top down”

```
nub :: Eq a => [a] -> [a]
nub [] = []
nub(x:xs) =
    if elem x xs
    then nub xs
    else x : nub xs
```

```
nub [1,2,1] = nub (1:[2,1]) = nub [2,1] = nub (2:[1]) = 2 : nub [1] = 2 : nub (1:[]) = 2 : 1 : nub [] = 2 : 1 : [] = 2 :[1]
```
The nub of a list is the list with all duplicates removed.

\[
\text{nub} \quad :: \quad \text{Eq} \ a \Rightarrow [a] \rightarrow [a]
\]

\[
\text{nub} \ [\] = []
\]

\[
\text{nub}(x:xs) =
\begin{align*}
\text{if} \ & \ \text{elem} \ x \ xs \\
\text{then} \ & \ \text{nub} \ xs \\
\text{else} \ & \ x : \ \text{nub} \ xs
\end{align*}
\]

Rewrite ... work “top down”

\[
\text{nub} \ [1,2,1] \\
= \text{nub} \ (1:[2,1]) \\
= \text{nub} \ [2,1] \\
= \text{nub} \ (2:[1]) \\
= 2 : \ \text{nub} \ [1] \\
= 2 : \ \text{nub} \ (1:[]) \\
= 2 : 1 : \ \text{nub} \ [\] \\
= 2 : 1 : \ [] \\
= 2 : [1] \\
= [2,1]
\]
The nub of a list is the list with all duplicates removed.

Accept the template: the lists get shorter . . .

foo [] = ...
foo (x:xs) = ... x ... foo xs ...
The nub of a list is the list with all duplicates removed.

Accept the template: the lists get shorter . . .

foo [] = ...
foo (x:xs) = ... x ... foo xs ...  

... and look at some examples

nub [1,2] = [1,2]
nub [2,1,2] = [1,2]
How to get started (with recursion)?

Examples, examples, examples. From simple “five finger” exercises, to a favourite small library.
RESOURCES
HASKELL
the craft of functional programming
Third edition

Graham Hutton
Programming in Haskell
Second Edition
Wiki-books, MOOCs, video channels, try-XXX, tutorials
Working together

XXX-bridge, code clubs, meet-ups, reading groups...
AND THEN ...
Pick any language

Start with the concrete before going to complex abstractions.
Choosing a language
Find a project

- Reimplement something
- Try something new
- Join an Open Source project
Systems
This board belongs to Newcastle University PhD student Tom Fisher, who is doing research in homological algebra. Thanks to Christian Perfect for the photo. whatsonmyblackboard.wordpress.com
Enjoy!

Type-driven development

Functional Concurrency

Systems programming in ML
SIMON THOMPSON

LEARNING FUNCTIONAL PROGRAMMING