The Haskell HOP: Higher-order Programming

COS 441 Slides 6

Slide content credits: Ranjit Jhala, UCSD
Agenda

• Haskell so far:
  – First-order functions

• This time:
  – Higher-order functions:
    • Functions as data, arguments & results
    • Reuseable abstractions
    • Capturing recursion patterns
  – Functional programming really starts to differentiate itself!
FUNCTIONS AS FIRST CLASS VALUES
A Perspective on Java

• In Java, you can do lots of things with integers:
  – create them wherever you want, in any bit of code
  – operate on them (add, subtract, etc)
  – pass them to functions, return them as results from functions
  – store them in data structures

• In Java, you can do barely anything at all with a method:
  – all you can do is declare a method inside a pre-existing class
    • you can't pass them to functions
    • you can't return them as results
    • you can't store them in data structures
    • you can't define them locally where you need them
  – of course, you can declare an entire new class (at the top level) and put the one method you are interested in inside it
    • this is incredibly heavy weight and still isn't very flexible!!
    • you still can't define methods locally where you want them
Functions as First-Class Data

- Haskell treats functions as first-class data. So does:
  - SML, OCaml, Scala (an OO language)

- "First-class" == all the "privileges" of any other data type:
  - you can declare them wherever you want
    - declarations can depend upon local variables in the context
  - you can pass them as arguments to functions
  - you can return them as results
  - you can store them in data structures

- This feature makes it easy to create powerful abstractions
- Because it is easy, it encourages a programming style in which there is great code reuse, many abstractions and clear code
Functions as First-Class Data

• An example:

\[
\begin{align*}
\text{plus1} & \quad x = x + 1 \\
\text{minus1} & \quad x = x - 1
\end{align*}
\]

• Storing functions in data structures:

\[
\begin{align*}
\text{funp} :: (\text{Int} \to \text{Int}, \text{Int} \to \text{Int}) \\
\text{funp} = (\text{plus1}, \text{minus1})
\end{align*}
\]

• .. any data structure:

\[
\begin{align*}
\text{funs} :: [\text{Int} \to \text{Int}] \\
\text{funs} = [\text{plus1}, \text{minus1}, \text{plus1}]
\end{align*}
\]
Functions as Inputs

- An example:

\[
\text{doTwice } f \ x = f \ (f \ x)
\]

- Using it:

\[
\text{plus2 :: Int } \to \text{ Int}
\]

\[
\text{plus2 } = \text{doTwice } \text{plus1}
\]
Functions as Inputs

• An example:

\[ \text{doTwice } f \ x = f \ (f \ x) \]

• Using it:

\[ \text{plus2 :: Int } \to \text{ Int } \]
\[ \text{plus2} = \text{doTwice plus1} \]

• Reasoning about it:

\[ \text{plus2 } 3 \]
Functions as Inputs

• An example:

\[
\text{doTwice } f \ x = f (f \ x)
\]

• Using it:

\[
\text{plus2} :: \text{Int} \rightarrow \text{Int}
\]
\[
\text{plus2} = \text{doTwice } \text{plus1}
\]

• Reasoning about it:

\[
\text{plus2} \ 3 \\
= (\text{doTwice } \text{plus1}) \ 3 \quad \text{(unfold plus2)}
\]
Functions as Inputs

• An example:

\[ \text{doTwice } f \ x = f \ (f \ x) \]

• Using it:

\[ \begin{align*}
\text{plus2} & : \text{Int} \to \text{Int} \\
\text{plus2} & = \text{doTwice } \text{plus1}
\end{align*} \]

• Reasoning about it:

\[ \begin{align*}
\text{plus2} \ 3 &= (\text{doTwice } \text{plus1}) \ 3 \quad \text{(unfold plus2)} \\
&= \text{doTwice } \text{plus1} \ 3 \quad \text{(parenthesis convention)}
\end{align*} \]
Functions as Inputs

• An example:

\[ \text{doTwice } f \ x = f \ (f \ x) \]

• Using it:

\[ \text{plus2 :: Int } \rightarrow \ \text{Int} \]
\[ \text{plus2 } = \text{doTwice } \text{plus1} \]

• Reasoning about it:

\[
\begin{align*}
\text{plus2 } 3 & = (\text{doTwice } \text{plus1}) \ 3 \\
& = \text{doTwice } \text{plus1 } 3 \\
& = \text{plus1 } (\text{plus1 } 3) \\
& = \text{plus1 } (3 + 1) \\
& = \text{plus1 } 4 \\
& = 4 + 1 = 5
\end{align*}
\]
Interlude

• What have we learned?
What have we learned? Almost nothing!

- function application is left-associative:
  - \(((f \ x) \ y) \ z \equiv f\ x\ y\ z\)
- like + or - is left-associative:
  - \((3 - 4) - 6 \equiv 3 - 4 - 6\)
  - this is useful, but intellectually uninteresting

We have, however, \textbf{unlearned} something important:

- some things one might have thought were fundamental differences between functions and other data types, turn out not to be differences at all!

PL researchers (like me!) often work with the theory of functional languages because they are uniform and elegant

- they don't make unnecessary distinctions
- they get right down to the essentials, the heart of computation
- at the same time, they do not lack expressiveness
Functions as Results

• Rather than writing multiple functions "plus1", "plus2", "plus3" we can write one:

\[
\text{plusn} :: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}) \\
\text{plusn } n = f \\
\text{ where } f \ x = x + n
\]

• **plusn** returns a function -- one that adds \( n \) to its argument

• any time we need an instance of plus, it is easy to build one:

\[
\text{plus10} :: \text{Int} \rightarrow \text{Int} \\
\text{plus10} = \text{plusn} \ 10
\]

• we can also use **plusn** directly:

\[
\text{result1} = (\text{plusn} \ 25) \ 100
\]
More trivial reasoning:

\[
\begin{align*}
\text{result1} &= (\text{plusn } 25) \ 100 \\
&= (f) \ 100 \ \text{where } f \ x = x + 25 \quad \text{(unfold plusn)} \\
&= 100 + 25 \quad \text{(unfold f)} \\
&= 125 \quad \text{(def of +)}
\end{align*}
\]

\[
\text{plusn} :: \text{Int} \to (\text{Int} \to \text{Int}) \\
\text{plusn } n = f \\
\text{where } f \ x = x + n
\]
• Function app is left-assoc.; Function types are right-assoc.

\[(\text{plusn } 25) \ 100 \equiv \text{plusn } 25 \ 100\]

\[\text{Int -> (Int -> Int)} \equiv \text{Int -> Int -> Int}\]

• We've seen two uses of \text{plusn}:

\[\text{plus20} = \text{plusn } 20\]

\[\text{oneTwentyFive} = \text{plusn } 25 \ 100\]

• Whenever we have a function \(f\) with type \(T_1 \rightarrow T_2 \rightarrow T_3\), we can choose:
  
  – apply \(f\) to both arguments right now, giving a \(T_3\)

  – partially applying \(f\), ie: applying \(f\) to one argument, yielding new function with type \(T_2 \rightarrow T_3\) and a chance to apply the new function to a second argument later
Defining higher-order functions

• The following was a stupid way to define plusn --- but it made it clear plusn was indeed returning a function:

\[
\text{plusn} :: \text{Int} \to \text{Int} \to \text{Int} \\
\text{plusn} \ n \ = \ f \\
\quad \text{where } f \ x \ = \ x \ + \ n
\]

• This is more beautiful code:

\[
\text{plusn}' :: \text{Int} \to \text{Int} \to \text{Int} \\
\text{plusn}' \ n \ x \ = \ x \ + \ n
\]

• We can prove them equivalent for all arguments a and b

\[
\text{plusn} \ a \ b \ = \ f \ b \ \text{where } f \ x \ = \ x \ + \ a \quad \text{(unfold plusn)} \\
\quad = \ b \ + \ a \quad \text{(unfold f)} \\
\quad = \ \text{plusn}' \ a \ b \quad \text{(fold plusn')}
\]

• So of course we can partially apply plusn' just like plusn
Anonymous Numbers

• You are all used to writing down numbers inside expressions
  – This:
    
    \[ 2 + 3 \]
  
  – Is way more compact than this:
    
    \[
    \begin{align*}
    \text{two} &= 2 \\
    \text{three} &= 3 \\
    \text{sum} &= \text{two} + \text{three}
    \end{align*}
    \]

  – Why can't functions play by the same rules?
Anonymous Numbers

• Compare:

\[
\begin{align*}
\text{plus1 } x &= x + 1 \\
\text{minus1 } x &= x - 1 \\
\text{doTwice } f \; x &= f (f \; x) \\
\text{baz} &= \text{doTwice } \text{plus1 } 3 \\
\text{bar} &= \text{doTwice } \text{minus1 } 7 \\
\end{align*}
\]

• When are anonymous functions a good idea?
  – When functions are small and not reused.

• Why is this a good language feature?
  – It encourages the definition of abstractions like doTwice
  – Why? Without anonymous functions, doTwice would be a little harder to use -- heavier weight; programmers would do it less
  – Moreover, why make different rules for numbers vs. functions?
More useful abstractions

• Do you like shell scripting? Why not build your own pipeline operator in Haskell?

\[(|>) \ x \ f = f \ x\]

define an infix operator by putting a name made of symbols inside parens

arguments, body are the same as usual

• Use it:

\[
dothrice \ f \ x = x |> f |> f |> f
\]

transmute \ x = x |> plusn 4
|> minus1
|> even
|> not
More useful abstractions

- Do you like shell scripting? Why not build your own pipeline operator in Haskell?

\[(\mid>) \; x \; f = f \; x\]

define an infix operator by putting a name made of symbols inside parens

arguments, body are the same as usual

- Use it:

\[
\text{dothrice } f \; x = x \; \mid> \; f \; \mid> \; f \; \mid> \; f
\]

\[
\text{transmute } x = x \; \mid> \; (\text{plusn} \; 4) \; \\
\quad \mid> \; \text{minus1} \; \\
\quad \mid> \; \text{even} \; \\
\quad \mid> \; \text{not}
\]

by default: function application has the highest precedence

by default: operators left associative
More useful abstractions

• Do you like shell scripting? Why not build your own pipeline operator in Haskell?

\[(\mid >) \ x \ f = f \ x\]

- define an infix operator by putting a name made of symbols inside parens
- arguments, body are the same as usual

• Use it:

\[\text{dothrice } f \ x = x \mid > f \mid > f \mid > f\]

\[\text{transmute } x = (((x \mid > \text{plusn} \ 4) \mid > \text{minus1}) \mid > \text{even}) \mid > \text{not})\]

by default: operators left associative
More useful abstractions

• Understanding functions in Haskell often boils down to understanding their type

• What type does the pipeline operator have?

\[(|>) \ x \ f = f \ x\]

\[(|>) :: a \to (a \to b) \to b\]

• Read it like this: "for all types \(a\) and all types \(b\), \(|>\) takes a value of type \(a\) and a function from \(a\) to \(b\) and returns a \(b\)"

• Hence:

\[(3 \ |> \ \text{plus1}) :: \text{Int}\] (a was Int, b was Int)

\[(3 \ |> \ \text{even}) :: \text{Bool}\] (a was Int, b was Bool)

\[("hello" \ |> \ \text{putStrLn}) :: \text{IO ()}\] (a was String, b was IO ())
• Another heavily-used operator, function composition:

\((.) f g x = f (g x)\)
More useful abstractions

• Another heavily-used operator, function composition:

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

• What type does it have?

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

type of f

type of g

type of f . g
More useful abstractions

• Another heavily-used operator, function composition:

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

• What type does it have?

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

  type of f  type of g  type of f . g

• Examples:

  plus2 = plus1 . plus1

  odd = even . plus1

  bof = doTwice plus1 . doTwice minus1
  baz = doTwice (plus1 . minus1)

Exercise: prove equivalence
ABSTRACTING RECURSION PATTERNS
Abstracting Computation Patterns

- Higher-order functions and polymorphism are the "secret sauce" that really makes functional programming fun.

- They make it not only possible but easy and delightful* for programmers to factor out repeated patterns in their code into highly reusable routines.

- It's especially effective in recursive routines -- one can sometimes eliminate the explicit recursion to be left with simple, non-recursive and abundantly clear code.

* Some people find delight from different sources than I do.
Recall: Polymorphic Lists

• Lists are heavily used in Haskell and other functional programming languages because they are light-weight, built-in "collection" data structure

• However, every major idea we present using lists applies similarly to any collection data structure we might define

• Recall some of the basic operations:
  
  \[
  \begin{align*}
  [] & : \text{[a]} \\
  (:) & : \text{a} \rightarrow \text{[a]} \rightarrow \text{[a]} \\
  (++) & : \text{[a]} \rightarrow \text{[a]} \rightarrow \text{[a]} \\
  \text{head} & : \text{[a]} \rightarrow \text{a} \\
  \text{tail} & : \text{[a]} \rightarrow \text{a} \\
  \text{length} & : \text{[a]} \rightarrow \text{Int}
  \end{align*}
  \]
Computation Pattern: "Apply to all"

• Recall that strings are lists:

\[
\text{type String} = [\text{Char}]
\]

• Suppose we want to convert all characters to upper case:

\[
toUpperString :: \text{String} \to \text{String}
\]
\[
toUpperString \emptyset = \emptyset
\]
\[
toUpperString (x:xs) = \text{toUpper } x : \text{toUpperString } xs
\]

• Here I've applied toUpper to all elements of the list

Comment: try finding functions like "toUpper" by searching by type on http://haskell.org/hoogle
Computation Pattern: "Apply to all"

- Similar idioms come up often, even in completely different applications:

```haskell
type Point = (Int, Int)
type Vector = (Int, Int)
type Polygon = [XY]
```

- It is easy to move a single point:

```haskell
shiftPoint :: Vector -> Point -> Point
shiftPoint (dx, dy) (x, y) = (x + dx, y + dy)
```

- And with more work, entire polygon:

```haskell
shift :: Vector -> Polygon -> Polygon
shift d ([ ] ) = [ ]
shift d (x:xs) = shiftPoint d x : shift d xs
```
Computation Pattern: "Apply to all"

- How to extract the pattern?

  \[
  \text{shift} :: \text{Vector} \rightarrow \text{Polygon} \rightarrow \text{Polygon} \\
  \text{shift} \ d \ \text{[]} \ = \ \text{[]} \\
  \text{shift} \ d \ (x:xs) = \text{shiftPoint} \ d \ x : \text{shift} \ d \ xs
  \]

- VS

  \[
  \text{toUpperCaseString} :: \text{String} \rightarrow \text{String} \\
  \text{toUpperCaseString} \ \text{[]} \ = \ \text{[]} \\
  \text{toUpperCaseString} \ (x:xs) = \text{toUpperCase} \ x : \text{toUpperCaseString} \ xs
  \]
Computation Pattern: "Apply to all"

- How to extract the pattern?

  ```haskell
  shift :: Vector -> Polygon -> Polygon
  shift d [ ]   = [ ]
  shift d (x:xs) = shiftPoint d x : shift d xs
  ```

- VS

  ```haskell
  toUpperString :: String -> String
  toUpperString [ ]   = [ ]
  toUpperString (x:xs) = toUpper x : toUpperString xs
  ```

- Here's the common pattern:

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

- map applies f to all elements of the list in place
Computation Pattern: "Apply to all"

- Rewriting:

  \[
  \text{toUpperString } s = \text{map toUpper } s
  \]

- and

  \[
  \text{shift } d \text{ polygon } = \text{map (shiftPoint } d) \text{ polygon}
  \]

- Now that's delightful!

- Compare:

  \[
  \begin{align*}
  \text{toUpperString } [ ] & = [ ] \\
  \text{toUpperString } (x:xs) & = \text{toUpper } x : \text{toUpperString } xs
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{shift } d [ ] & = [ ] \\
  \text{shift } d (x:xs) & = \text{shiftPoint } d \ x : \text{shift } d \ xs
  \end{align*}
  \]
• Rewrite this:

```
toUpperString s = map toUpper s
shift d polygon = map (shiftPoint d) polygon
```

• To this:

```
toUpperString = map toUpper
shift d = map (shiftPoint d)
```
A step further

- Rewrite this:

  \[
  \text{toUpperCaseString}\ s = \text{map}\ \text{toUpperCase}\ s \\
  \text{shift}\ d\ \text{polygon} = \text{map}\ (\text{shiftPoint}\ d)\ \text{polygon}
  \]

- To this:

  \[
  \text{toUpperCaseString} = \text{map}\ \text{toUpperCase} \\
  \text{shift}\ d = \text{map}\ (\text{shiftPoint}\ d)
  \]

- In general, rewrite:

  \[
  f\ x = e\ x
  \]

- To

  \[
  f = e \quad \text{(when}\ x\ \text{does not appear in}\ e)
  \]

  this is quite common but I actually find it harder to read
  the syntactic redundancy with argument "x" gives me a hint about the type
Computation Pattern: Iteration

• Two more functions:

\[
\begin{align*}
\text{listAdd} \; [\; ] &= 0 \\
\text{listAdd} \; (x:xs) &= x + (\text{listAdd} \; xs)
\end{align*}
\]

\[
\begin{align*}
\text{listMul} \; [\; ] &= 1 \\
\text{listMul} \; (x:xs) &= x \times (\text{listMul} \; xs)
\end{align*}
\]

• You can see the syntactic pattern. How do I capture it?
Computation Pattern: Iteration

- Two more functions:

  \[
  \text{listAdd} \ [\ ] = 0 \\
  \text{listAdd} \ (x:xs) = x + (\text{listAdd} \ xs)
  \]

  \[
  \text{listMul} \ [\ ] = 1 \\
  \text{listMul} \ (x:xs) = x \times (\text{listMul} \ xs)
  \]

- You can see the syntactic pattern. How do I capture it?

  \[
  \text{foldr} \ \text{op} \ \text{base} \ [\ ] = \text{base} \\
  \text{foldr} \ \text{op} \ \text{base} \ (x:xs) = x \ `\text{op}` \ (\text{foldr} \ \text{op} \ \text{base} \ xs)
  \]
Computation Pattern: Iteration

- Two more functions:

  \[
  \text{listAdd} \ [ \ ] = 0 \\
  \text{listAdd} \ (x:xs) = x + (\text{listAdd} \ xs)
  \]

  \[
  \text{listMul} \ [ \ ] = 1 \\
  \text{listMul} \ (x:xs) = x \ast (\text{listMul} \ xs)
  \]

- You can see the syntactic pattern. How do I capture it?

  \[
  \text{foldr} \ \text{op} \ \text{base} \ [ \ ] = \text{base} \\
  \text{foldr} \ \text{op} \ \text{base} \ (x:xs) = x \ `\text{op}` \ (\text{foldr} \ \text{op} \ \text{base} \ xs)
  \]

  \[
  \text{listAdd} = \text{foldr} \ 0 \ (+) \\
  \text{listMul} = \text{foldr} \ 1 \ (*)
  \]
Computation Pattern: Iteration

• Some more folds:

\[\text{length } \emptyset = 0\]
\[\text{length } (x:xs) = 1 + (\text{length } xs)\]

\[\text{factorial } 0 = 1\]
\[\text{factorial } n = n \times (\text{factorial } (n-1))\]

\[\text{sequence\_ :: } [\text{IO } ()] \rightarrow \text{IO } ()\]
\[\text{sequence\_ } \emptyset = \text{null}\]
\[\text{sequence\_ } (a:as) = a \gg \text{sequence\_ } as\]

\[\text{foldr op base } \emptyset = \text{base}\]
\[\text{foldr op base } (x:xs) = x \ `\text{op}` \ (\text{foldr op base } xs)\]
Computation Pattern: Iteration

- Some more folds:

\[
\text{length } [ ] = 0 \\
\text{length } (x:xs) = 1 + (\text{length } xs)
\]

\[
\text{length } xs = \text{foldr } 0 (1+) xs
\]

factorial 0 = 1
factorial n = n * (factorial (n-1))

sequence_ :: [IO ()] -> IO ()
sequence_ [ ] = null
sequence_ (a:as) = a >> sequence_ as

\[
\text{foldr } \text{op} \text{ base } [ ] = \text{base} \\
\text{foldr } \text{op} \text{ base } (x:xs) = x \ `\text{op}` (\text{foldr } \text{op} \text{ base } xs)
\]
Computation Pattern: Iteration

• Some more folds:

\[
\begin{align*}
\text{length } [ ] & = 0 \\
\text{length } (x:\text{x}s) & = 1 + (\text{length } \text{x}s) \\
\text{length } \text{x}s & = \text{foldr } 0 (1+) \text{x}s \\
\text{factorial } 0 & = 1 \\
\text{factorial } n & = n \times (\text{factorial } (n-1)) \\
\text{factorial } \text{n} & = \text{foldr } 1 (*) [1..n] \\
\text{sequence } \text{a}s & = \text{sequence}_\text{:: } [\text{IO ()}] \rightarrow \text{IO ()} \\
\text{sequence } [ ] & = \text{null} \\
\text{sequence } (a:\text{a}s) & = a \gg \text{sequence } \text{a}s \\
\text{foldr op base } [ ] & = \text{base} \\
\text{foldr op base } (x:\text{x}s) & = x \ `\text{op}` \ (\text{foldr op base } \text{x}s)
\end{align*}
\]
Computation Pattern: Iteration

- Some more folds:

  \[
  \begin{align*}
  \text{length } [ ] &= 0 \\
  \text{length } (x:xs) &= 1 + (\text{length } xs)
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{length } xs &= \text{foldr } 0 (1+) \text{ } xs \\
  \text{factorial } 0 &= 1 \\
  \text{factorial } n &= n \times (\text{factorial } (n-1))
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{sequence_ } :: [\text{IO ()}] \rightarrow \text{IO ()} \\
  \text{sequence_ } [ ] &= \text{null} \\
  \text{sequence_ } (a:as) &= a \gg \text{sequence_ } as
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{foldr } \text{op} \text{ } \text{base } [ ] &= \text{base} \\
  \text{foldr } \text{op} \text{ } \text{base } (x:xs) &= x \ `\text{op}` \ (\text{foldr } \text{op} \text{ } \text{base } xs)
  \end{align*}
  \]
Can we define map in terms of foldr?

map :: (a -> b) -> [a] -> [b]

foldr :: b -> (a -> b -> b) -> [a] -> b

• Can we define map in terms of foldr?
Map and Fold

map :: (a -> b) -> [a] -> [b]
foldr :: b -> (a -> b -> b) -> [a] -> b

• Can we define map in terms of foldr?

map f xs = foldr [] (\ys -> f x : ys) xs
Map and Fold

map :: (a -> b) -> [a] -> [b]
foldr :: b -> (a -> b -> b) -> [a] -> b

• Can we define foldr in terms of map?
Can we define foldr in terms of map?

- No. How do we prove it?
- A formal theorem might say:
  - for all $b, f, xs$, there exists $g, ys$ such that $\text{foldr } b \ f \ xs = \text{map } g \ ys$
Can we define `foldr` in terms of `map`?

- No. How do we prove it?
- A formal theorem might say:
  - for all `b`, `f`, `xs`, there exists `g`, `ys` such that `foldr b f xs == map g ys`
- To disprove that theorem, find a counter-example. Consider:
  - `length xs = foldr 0 (1+) xs`
- Does there exist a `g` and `ys` such that
  - `fold 0 (1+) xs == map g ys`?
Can we define foldr in terms of map?
- No. How do we prove it?
- A formal theorem might say:
  - for all $b$, $f$, $xs$, there exists $g$, $ys$ such that $\text{foldr } b f xs == \text{map } g ys$
- To disprove that theorem, find a counter-example. Consider:
  - $\text{length } xs = \text{foldr } 0 (1+) xs$
- Does there exist a $g$ and $ys$ such that
  - $\text{fold } 0 (1+) xs == \text{map } g ys$?
- Consider the types:
  - $\text{fold } 0 (1+) xs :: \text{Int}$
  - $\text{map } g ys :: [b]$ incomparable types no matter what $b$ is!
Exercises

• Lists are one kind of container data structure; they support
  – map: the "apply all in place" pattern
  – fold: "the accumulative iteration" pattern

• What about trees?

```haskell
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

• Define treeMap and treeFold
• Give them appropriate types
• Can you define treeMap in terms of treeFold? Vice versa?
A NOTE ON I/O
A Note on I/O

• What is the null action?
  
  \[
  \text{null} :: \text{IO}() \\
  \text{null} = \text{return}() \\
  \]

• \text{return} is very (very!) different from return in Java or C

• "\text{return v}" creates an action that has no effect but results in v
  
  \[
  \text{return } "hi" \quad -- \text{action that returns the string } "hi" \text{ and does nothing else} \\
  \text{return } () \quad -- \text{action that returns the unit value } () \text{ and does nothing else}
  \]
A Note on I/O

• We can use `return` in conjunction with do notation

• Example:

```haskell
do
    s <- return "hi"
    putStrLn s
```

• In general:

```haskell
do
    x <- return e
    ...
    x ...
```

• This is another powerful law for reasoning about programs using substitution of equals for equals

• The fascinating thing is that it interacts safely with effects

• More on this later!
Summary

• Higher-order programs
  – receive functions as arguments
  – return functions as results
  – store functions in data structures
  – use anonymous functions wisely

• Great programmers identify repeated patterns in their code and devise higher-order functions to capture them
  – map and fold are two of the most useful