1 Higher-Order Functions

- A function is Higher-Order if it takes a function as an argument or returns one as its result.
- Higher-order function aren’t weird; the differentiation operation from high-school calculus is higher-order:

\[
\text{deriv} :: (\text{Float} \rightarrow \text{Float}) \rightarrow \text{Float} \rightarrow \text{Float} \\
\text{deriv} \ f \ x = (f(x+dx) - f \ x) / 0.0001
\]

- Many recursive functions share a similar structure. We can capture such “recursive patterns” in a higher-order function.
- We can often avoid the use of explicit recursion by using higher-order functions. This leads to functions that are shorter, and easier to read and maintain.

2 Currying Revisited

- We have already seen a number of higher-order functions. In fact, any curried function is higher-order. Why? Well, when a curried function is applied to one of its arguments it returns a new function as the result.

  \text{Uh, what was this currying thing?}

- A curried function does not have to be applied to all its arguments at once. We can supply some of the arguments, thereby creating a new specialized function. This function can, for example, be passed as argument to a higher-order function.
3 Currying Revisited...

How is a curried function defined?

- A curried function of \( n \) arguments (of types \( t_1, t_2, \ldots, t_n \)) that returns a value of type \( t \) is defined like this:

\[
\text{fun} : : t_1 \rightarrow t_2 \rightarrow \cdots \rightarrow t_n \rightarrow t
\]

- This is sort of like defining \( n \) different functions (one for each \( \rightarrow \)). In fact, we could define these functions explicitly, but that would be tedious:

\[
\begin{align*}
\text{fun}_1 & : : t_2 \rightarrow \cdots \rightarrow t_n \rightarrow t \\
\text{fun}_1 a_2 \cdots a_n & = \cdots \\
\text{fun}_2 & : : t_3 \rightarrow \cdots \rightarrow t_n \rightarrow t \\
\text{fun}_2 a_3 \cdots a_n & = \cdots
\end{align*}
\]

4 Currying Revisited...

Duh, how about an example?

- Certainly. Let's define a recursive function \( \text{get\_nth } n \) \( \text{x}\) which returns the \( n \):th element from the list \( \text{x}\):

\[
\begin{align*}
\text{get\_nth } 1 & (x:_\_ ) = x \\
\text{get\_nth } n & (\_ :xs) = \text{get\_nth } (n-1) \text{ xs}
\end{align*}
\]

\( \text{get\_nth } 10 \) "Bartholomew" \( \Rightarrow \) 'e'

- Now, let's use \( \text{get\_nth} \) to define functions \( \text{get\_second}, \text{get\_third}, \text{get\_fourth}, \) and \( \text{get\_fifth}, \) without using explicit recursion:

\[
\begin{align*}
\text{get\_second} & = \text{get\_nth } 2 & \text{get\_fourth} & = \text{get\_nth } 4 \\
\text{get\_third} & = \text{get\_nth } 3 & \text{get\_fifth} & = \text{get\_nth } 5
\end{align*}
\]

5 Currying Revisited...

\( \text{get\_fifth} \) "Bartholomew" \( \Rightarrow \) 'h'

map (\( \text{get\_nth } 3 \))

["mob","sea","tar","bat"] \( \Rightarrow \)

"bart"

So, what's the type of \( \text{get\_second} \)?

- Remember the Rule of Cancellation?

- The type of \( \text{get\_nth} \) is \( \text{Int} \rightarrow [a] \rightarrow a \).

- \( \text{get\_second} \) applies \( \text{get\_nth} \) to one argument. So, to get the type of \( \text{get\_second} \) we need to cancel \( \text{get\_nth} \)'s first type: \( \text{Int} \rightarrow [a] \rightarrow a \equiv [a] \rightarrow a \).
6 Patterns of Computation

- **Mappings**
  - Apply a function \( f \) to the elements of a list \( L \) to make a new list \( L' \). Example: Double the elements of an integer list.

- **Selections**
  - Extract those elements from a list \( L \) that satisfy a predicate \( p \) into a new list \( L' \). Example: Extract the even elements from an integer list.

- **Folds**
  - Combine the elements of a list \( L \) into a single element using a binary function \( f \). Example: Sum up the elements in an integer list.

7 The map Function

- \( \text{map} \) takes two arguments, a function and a list. \( \text{map} \) creates a new list by applying the function to each element of the input list.

- \( \text{map} \)'s first argument is a function of type \( a \rightarrow b \). The second argument is a list of type \( [a] \). The result is a list of type \( [b] \).

\[
\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]
\]

\[
\text{map} f \ [] = [ ]
\]

\[
\text{map} f (x:xs) = f x : \text{map} f xs
\]

- We can check the type of an object using the \texttt{:type} command. Example: \texttt{:type map}.

8 The map Function...

\[
\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]
\]

\[
\text{map} f \ [] = [ ]
\]

\[
\text{map} f (x:xs) = f x : \text{map} f xs
\]

\[
\text{map inc [1,2,3,4]} \Rightarrow [2,3,4,5]
\]

9 The map Function...

\[
\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]
\]

\[
\text{map} f \ [] = [ ]
\]

\[
\text{map} f (x:xs) = f x : \text{map} f xs
\]

\[
\text{map f []} = [ ] \text{ means: “The result of applying the function f to the elements of an empty list is the empty list.”}
\]
map \( f \) \((x:xs)\) = \( f \) \( x \) : map \( f \) \( xs \) means: “applying \( f \) to the list \((x:xs)\) is the same as applying \( f \) to \( x \) (the first element of the list), then applying \( f \) to the list \( xs \), and then combining the results.”

10  The map Function...

Simulation:

map square [5,6] ⇒
  square 5 : map square [6] ⇒
    25 : map square [6] ⇒
      25 : (square 6 : map square []) ⇒
        25 : (36 : map square []) ⇒
          25 : (36 : []) ⇒
            25 : [36] ⇒
              [25,36]

11  The filter Function

- Filter takes a predicate \( p \) and a list \( L \) as arguments. It returns a list \( L' \) consisting of those elements from \( L \) that satisfy \( p \).
- The predicate \( p \) should have the type \( \text{a \rightarrow Bool} \), where \( \text{a} \) is the type of the list elements.

Examples:

filter even [1..10] ⇒ [2,4,6,8,10]
filter even (map square [2..5]) ⇒
  filter even [4,9,16,25] ⇒ [4,16]
filter gt10 [2,5,9,11,23,114]
  where gt10 x = x > 10 ⇒ [11,23,114]

12  The filter Function...

- We can define filter using either recursion or list comprehension.

Using recursion:

filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs)
  | p x = x : filter p xs
  | otherwise = filter p xs

Using list comprehension:

filter :: (a -> Bool) -> [a] -> [a]
filter p xs = [x | x <- xs, p x]
13 The filter Function...

\[ \text{filter :: (a->Bool)->[a]->[a]} \]
\[
\text{filter} \_ \ [] = []
\]
\[
\text{filter} \ p \ (x:xs)
\]
\[
| \ p \ x = x \ : \ \text{filter} \ p \ xs
\]
\[
| \ \text{otherwise} = \text{filter} \ p \ xs
\]

\text{filter even [1,2,3,4] \Rightarrow [2,4]} \]

14 The filter Function...

- \text{doublePos} doubles the positive integers in a list.

\text{getEven :: [Int] -> [Int]}
\text{getEven xs = filter even xs}

\text{doublePos :: [Int] -> [Int]}
\text{doublePos xs = map dbl (filter pos xs)}
\text{where dbl x = 2 * x}
\text{pos x = x > 0}

Simulations:

\text{getEven [1,2,3] \Rightarrow [2]}

\text{doublePos [1,2,3,4] \Rightarrow}
\text{map dbl (filter pos [1,2,3,4]) \Rightarrow}
\text{map dbl [2,4] \Rightarrow [4,8]}

15 fold Functions

- A common operation is to combine the elements of a list into one element. Such operations are called reductions or accumulations.

Examples:

\text{sum [1,2,3,4,5] \equiv}
\text{(1 + (2 + (3 + (4 + (5 + 0))))) \Rightarrow 15}

\text{concat ["H","i","!"] \equiv}
\text{("H" ++ ("i" ++ ("!" ++ ")))) \Rightarrow "Hi!"}

- Notice how similar these operations are. They both combine the elements in a list using some binary operator (+, ++), starting out with a “seed” value (0, "").
16 fold Functions...

- Haskell provides a function `foldr` ("fold right") which captures this pattern of computation.
- `foldr` takes three arguments: a function, a seed value, and a list.

Examples:

```plaintext
foldr (+) 0 [1,2,3,4,5] ⇒ 15
foldr (++) "" ["H","i","!"] ⇒ "Hi!
```

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

17 fold Functions...

- Note how the fold process is started by combining the last element \( x_n \) with \( z \). Hence the name seed.

\[
foldr(\oplus)z[x_1 \cdots x_n] = (x_1 \oplus (x_2 \oplus \cdots (x_n \oplus z)))
\]

- Several functions in the standard prelude are defined using `foldr`:

```plaintext
and, or :: [Bool] → Bool
and xs = foldr (&&) True xs
or xs = foldr (||) False xs

?? or [True,False,False] ⇒
foldr (||) False [True,False,False] ⇒
True || (False || (False || False)) ⇒ True
```

18 fold Functions...

- Remember that `foldr` binds from the right:

```
foldr (+) 0 [1,2,3] ⇒ (1+(2+(3+0)))
```

- There is another function `foldl` that binds from the left:

```
foldl (+) 0 [1,2,3] ⇒ (((0+1)+2)+3)
```

- In general:

\[
foldl(\oplus)z[x_1 \cdots x_n] = (((z \oplus x_1) \oplus x_2) \oplus \cdots \oplus x_n)
\]

19 fold Functions...

- In the case of (+) and many other functions

```
foldl(\oplus)z[x_1 \cdots x_n] = foldr(\oplus)z[x_1 \cdots x_n]
```

- However, one version may be more efficient than the other.
21 Operator Sections

- We’ve already seen that it is possible to use operators to construct new functions:

  - $(\ast 2)$ – function that doubles its argument
  - $(>2)$ – function that returns `True` for numbers > 2.

- Such partially applied operators are know as **operator sections**. There are two kinds:

  $(\text{op } a) \ b = b \ \text{op} \ a$

  $(\ast 2) \ 4 = 4 \ * \ 2 = 8$
  $(>2) \ 4 = 4 \ > \ 2 = \text{True}$
  $(++ "\n") \ "Bart" = "Bart" \ ++ \ "\n"

22 Operator Sections...
23 takeWhile & dropWhile

- We've looked at the list-breaking functions drop & take:

\[
\text{take 2 ['a','b','c']} \Rightarrow ['a','b']
\]
\[
\text{drop 2 ['a','b','c']} \Rightarrow ['c']
\]

- takeWhile and dropWhile are higher-order list-breaking functions. They take/drop elements from a list while a predicate is true.

\[
\text{takeWhile even [2,4,6,5,7,4,1]} \Rightarrow [2,4,6]
\]
\[
\text{dropWhile even [2,4,6,5,7,4,1]} \Rightarrow [5,7,4,1]
\]

24 takeWhile & dropWhile...

\[
\text{takeWhile p [ ] = [ ]}
\]
\[
\text{takeWhile p (x:xs)}
\]
\[
\quad | \ p x = x : \ \text{takeWhile p xs}
\]
\[
\quad | \ otherwise = [ ]
\]

\[
\text{dropWhile p [ ] = [ ]}
\]
\[
\text{dropWhile p (x:xs)}
\]
\[
\quad | \ p x = \ \text{dropWhile p xs}
\]
\[
\quad | \ otherwise = x:xs
\]

25 takeWhile & dropWhile...

- Remove initial/final blanks from a string:

\[
\text{dropWhile ((==) ' ') "   Hi!" } \Rightarrow "Hi!"
\]
\[
\text{takeWhile ((/=) ' ') "Hi! " } \Rightarrow "Hi!"
\]

26 Summary

- Higher-order functions take functions as arguments, or return a function as the result.

- We can form a new function by applying a curried function to some (but not all) of its arguments. This is called partial application.

- Operator sections are partially applied infix operators.
27 Summary...

- The standard prelude contains many useful higher-order functions:
  
  \textbf{map} \; f \; xs \; \text{creates a new list by applying the function} \; f \; \text{to every element of a list} \; xs.

  \textbf{filter} \; p \; xs \; \text{creates a new list by selecting only those elements from} \; xs \; \text{that satisfy the predicate} \; p \; \text{(i.e.} \; (p \; x) \; \text{should return True).}

  \textbf{foldr} \; f \; z \; xs \; \text{reduces a list} \; xs \; \text{down to one element, by applying the binary function} \; f \; \text{to successive elements, starting from the right.}

  \textbf{scanl/scanr} \; f \; z \; xs \; \text{perform the same functions as foldr/foldl, but instead of returning only the ultimate value they return a list of all intermediate results.}

28 Homework

\underline{Homework (a):}

- Define the map function using a list comprehension.

  \underline{Template:}

  \textbf{map} \; f \; x = [ \; \cdots \; | \; \cdots \; ]

\underline{Homework (b):}

- Use map to define a function \textbf{lengthall} \; xs which takes a list of strings \; xs \; as argument and returns a list of their lengths as result.

\underline{Examples:}

? \textbf{lengthall} \; ["Ay", \; "Caramba!"]
[2,8]

29 Homework

1. Give a accumulative recursive definition of foldl.

2. Define the minimum \; xs \; function using foldr.

3. Define a function \textbf{sumsq} \; n \; that returns the sum of the squares of the numbers \; [1 \cdots n]. Use map and foldr.

4. What does the function \textbf{mystery} \; below do?

\texttt{mystery} \; xs =
\texttt{foldr} \; (\++) \; [] \; (\texttt{map} \; \texttt{sing} \; \texttt{xs})
\texttt{sing} \; x = [x]

\underline{Examples:}

\texttt{minimum} \; [3,4,1,5,6,3] \; \Rightarrow \; 1
30 Homework...

- Define a function \( \text{zipp } f \text{ xs ys} \) that takes a function \( f \) and two lists \( \text{xs}=[x_1, \ldots, x_n] \) and \( \text{ys}=[y_1, \ldots, y_n] \) as argument, and returns the list \( [f \text{ } x_1 \text{ } y_1, \ldots, f \text{ } x_n \text{ } y_n] \) as result.

  - If the lists are of unequal length, an error should be returned.

**Examples:**

\[
\text{zipp (+) [1,2,3] [4,5,6]} \Rightarrow [5,7,9]
\]

\[
\text{zipp (==) [1,2,3] [4,2,2]} \Rightarrow \text{[False,True,True]}
\]

\[
\text{zipp (==) [1,2,3] [4,2]} \Rightarrow \text{ERROR}
\]

31 Homework

- Define a function \( \text{filterFirst p xs} \) that removes the first element of \( \text{xs} \) that does not have the property \( p \).

  **Example:**

\[
\text{filterFirst even [2,4,6,5,6,8,7]} \Rightarrow [2,4,6,6,8,7]
\]

- Use \( \text{filterFirst} \) to define a function \( \text{filterLast p xs} \) that removes the last occurrence of an element of \( \text{xs} \) without the property \( p \).

  **Example:**

\[
\text{filterLast even [2,4,6,5,6,8,7]} \Rightarrow [2,4,6,5,6,8]
\]