CSE399: Advanced Programming
Handout 2
Higher-Order Programming
In Haskell (and other functional languages), functions can be treated as “ordinary data”—they can be passed as arguments to other functions, returned as results, stored in lists, etc., etc.

Taking a function as an argument:

\[
\begin{align*}
\text{thrice} & : (\text{Int} \to \text{Int}) \to \text{Int} \to \text{Int} \\
\text{thrice} \; f \; x & = f (f (f \; x)) \\
\text{plus2} & : \text{Int} \to \text{Int} \\
\text{plus2} \; x & = x + 2 \\
\text{foo} & : \text{Int} \\
\text{foo} & = \text{thrice} \; \text{plus2} \; 10 \quad -- \quad \text{foo} = 16
\end{align*}
\]
Returning a function as a result:

plusn :: Int -> (Int->Int)
plusn n = f
    where  f x = n + x

plus5 :: Int -> Int
plus5 = plusn 5

bar1 = plus5 10  -- bar1 = 15

-- Or just...
bar2 = (plusn 5) 10  -- bar2 = 15
The type constructor \( \rightarrow \) is right-associative — i.e.,

\[
\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}
\]

means the same as

\[
\text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})
\]

That is, a function of type \( \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \) can be thought of as a function that takes an integer and returns a function from integers to integers!

So we can write \texttt{plusn} in a simpler way:

\[
\text{plusn'} :: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}) \quad -- \text{i.e., } \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}
\]

\[
\text{plusn'} \ n \ x = n + x \quad -- \text{i.e., } \text{plusn'} = (+)
\]

Each time we use \texttt{plusn}, we can choose whether to apply it to two integers to get an integer or to “partially apply” it to just one integer, yielding a function.
Putting these together...

\[
\text{thriceplus2} :: \text{Int} \to \text{Int} \\
\text{thriceplus2} = \text{thrice plus2} \quad \quad \text{-- partial application!}
\]

\[
\text{baz} :: \text{Int} \to \text{Int} \\
\text{baz} = \text{thrice thriceplus2} \quad \quad \text{-- again!!}
\]

\[
\text{-- Check: What is (baz 0)??}
\]
Polymorphism
The Length Function is Polymorphic

```
length :: [a] -> Int
length []     = 0
length (x:xs) = 1 + length xs
```

The “a” in the type of `length` is a placeholder that can be replaced with any type when `length` is applied.

- `length [1,2,3] ⇒ 3`
- `length ["a","b","c"] ⇒ 3`
- `length [[1],[[],[2,3]]] ⇒ 3`
Many of Haskell’s predefined functions are polymorphic

(++) :: [a] -> [a] -> [a]
id :: a -> a
head :: [a] -> a
tail :: [a] -> [a]
[] :: [a] -- interesting!

Quick check: what is the type of \texttt{tag1}?

\texttt{tag1 x = (1,x)}
Polymorphic functions — functions that can operate on any type of data — are often associated with polymorphic data structures — structures that can contain any type of data.

The previous examples involved lists and tuples. In particular, here are the types of the list and tuple constructors:

\[
(\cdot) :: a \to [a] \to [a] \\
(,) :: a \to b \to (a, b)
\]

We can also define new polymorphic data structures...
A User-Defined Polymorphic Data Structure

The type variable \( a \) on the left-hand-side of the \( = \) tells Haskell that \textit{Maybe} is a polymorphic data type:

\[
\text{data Maybe } a = \text{Nothing} \mid \text{Just } a
\]

Note the types of the constructors:

- \( \text{Nothing} :: \text{Maybe } a \)
- \( \text{Just} :: a \rightarrow \text{Maybe } a \)

Thus:

- \( \text{Just 3} :: \text{Maybe } \text{Int} \)
- \( \text{Just "x"} :: \text{Maybe } \text{String} \)
- \( \text{Just (3,True)} :: \text{Maybe } (\text{Int},\text{Bool}) \)
- \( \text{Just (Just 1)} :: \text{Maybe } (\text{Maybe } \text{Int}) \)
The most common use of Maybe is with a function that “may” return a useful value, but may also fail.

For example, the division operator \texttt{div} in Haskell will cause a run-time error if its second argument is zero. Thus we may wish to define a \texttt{safe division} function, as follows:

\begin{verbatim}
safeDivide :: Int -> Int -> Maybe Int
safeDivide x 0 = Nothing
safeDivide x y = Just (x \texttt{‘div‘} y)
\end{verbatim}
Polymorphic Higher-Order Functions
Recall from Section 4.1:

\[
\begin{align*}
\text{transList} :& \quad [\text{Vertex}] \to [\text{Point}] \\
\text{transList} [] & = [] \\
\text{transList} (p:ps) & = \text{trans} p : \text{transList} ps
\end{align*}
\]

(where \(\text{trans}\) converts ordinary cartesian coordinates into screen coordinates).

Also, from Chapter 3:

\[
\begin{align*}
\text{putCharList} :& \quad [\text{Char}] \to [\text{IO ()}] \\
\text{putCharList} [] & = [] \\
\text{putCharList} (c:cs) & = \text{putChar} c : \text{putCharList} cs
\end{align*}
\]

These definitions are very similar. Indeed, the only thing different about them (besides the variable names) is the function \(\text{trans}\) vs. the function \(\text{putChar}\).
Since `trans` and `putChar` are the differing elements, they should be arguments to the abstraction. In other words, we would like to define a function — let’s call it `map` — such that `map trans` behaves like `transList` and `map putChar` behaves like `putCharList`.

No problem:

```haskell
map f [] = []
map f (x:xs) = f x : map f xs
```

Now it is easy to redefine `transList` and `putCharList` in terms of `map`:

```haskell
transList xs = map trans xs
putCharList cs = map putChar cs
```
The great thing about `map` is that it is polymorphic. Its **most general** (or **principal**) **type** is:

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

Whatever type is instantiated for the type variable `a` must be the same at both instances of `a`, and similarly for `b`.

For example, since `trans :: Vertex \to Point`, we have

\[
\text{map trans} :: [Vertex] \to [Point]
\]

and since `putChar :: Char \to IO ()`,

\[
\text{map putChar} :: [Char] \to [IO ()]
\]
Digression: Arithmetic Sequences

Haskell provides a convenient special syntax for lists of numbers obeying simple rules:

\[
\begin{align*}
[1 \ldots 6] & \Rightarrow [1,2,3,4,5,6] \\
[1,3 \ldots 9] & \Rightarrow [1,3,5,7,9] \\
[5,4 \ldots 1] & \Rightarrow [5,4,3,2,1] \\
[2.4, 2.1 \ldots 0.3] & \Rightarrow [2.4, 2.1, 1.8, 1.5, \text{ etc.}]\end{align*}
\]
Another Example of Map

circles :: [Shape]
circles = map circle [2.4, 2.1 .. 0.3]

Now let’s draw them...
Another useful higher-order function:

\[
\text{zip} :: [a] \to [b] \to [(a,b)]
\]

\[
\text{zip } (a:as) (b:bs) = (a,b) : \text{zip } as \text{ bs}
\]

\[
\text{zip } _ _ = []
\]

For example:

\[
\text{zip } [1,2,3] \ [True, False, False]
\]

⇒ \[(1,True), (2,False), (3,False)\]

Quick check: What does \text{zip } [1..3] \ [1..5] \text{ yield?}
colCircles :: [(Color,Shape)]
colCircles = zip [Red,Blue,Green,
                 Cyan,Red,Magenta,
                 Yellow,White]
circles
Recall from Chapter 3 that `sequence_` takes a list of `IO()` actions and returns an `IO()` action that performs all the actions in the list in sequence.
g = do w <- openWindow "Bulls eye" (600,600)
    drawShapes w colCircles
    k <- getKey w
    closeWindow w

main = runGraphics g
The Result
When to Define Higher-Order Functions

Recognizing repeating patterns is the key, as we did for map. As another example, consider:

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

Note the similarities. Also note the differences (0 vs. 1 and + vs. *): it is these that will become parameters to the abstracted function.
Abstracting out the differences (\texttt{op} and \texttt{init}) leaves this common part:

\[
\begin{align*}
\text{fold } \text{op} \text{ init } \text{[]} & = \text{init} \\
\text{fold } \text{op} \text{ init } (x:xs) & = x \ '\text{op}' \ \text{fold } \text{op} \text{ init } \text{xs}
\end{align*}
\]

We recover \texttt{listSum} and \texttt{listProd} by instantiating \texttt{fold} with the appropriate parameters:

\[
\begin{align*}
\text{listSum } \text{xs} & = \text{fold } (+) \ 0 \ \text{xs} \\
\text{listProd } \text{xs} & = \text{fold } (*) \ 1 \ \text{xs}
\end{align*}
\]

Note that \texttt{fold} is polymorphic:

\[
\text{fold} \ :\ (a \to b \to b) \to b \to [a] \to b
\]
The **fold** function is predefined in Haskell, but it is actually called **foldr**, because it “folds from the right.” That is:

\[
\text{foldr \ op \ init \ (x_1 : x_2 : \ldots : x_n : [])} \\
\Rightarrow x_1 \ 'op' \ (x_2 \ 'op' \ (\ldots(x_n \ 'op' \ \text{init})\ldots))
\]

There is another predefined function **foldl** that “folds from the left”:

\[
\text{foldl \ op \ init \ (x_1 : x_2 : \ldots : x_n : [])} \\
\Rightarrow (\ldots((\text{init} \ 'op' \ x_1) \ 'op' \ x_2)\ldots) \ 'op' \ x_n
\]
Two Folds Are Better Than One

Why two folds? Because sometimes using one can be more efficient than the other. For example:

\[
\begin{align*}
\text{foldr} \ (\cdot\cdot) \ [] \ [x,y,z] & \Rightarrow x \cdot\cdot (y \cdot\cdot z) \\
\text{foldl} \ (\cdot\cdot) \ [] \ [x,y,z] & \Rightarrow (x \cdot\cdot y) \cdot\cdot z
\end{align*}
\]

The former is considerably more efficient than the latter (as discussed in the book); but this is not always the case — sometimes \text{foldl} is more efficient than \text{foldr}. Choose wisely!
We have seen the function `sequence_`, which takes a list of actions of type `IO()` and produces a single action of type `IO()`.

We can define `sequence_` in terms of `>>` and `foldl` as follows:

```haskell
sequence_ :: [IO ()] -> IO ()
sequence_ acts = foldl (>>) (return ()) acts
```
Reversing a List

Obvious but inefficient (why?):

\[
\begin{align*}
\text{reverse } [] &= [] \\
\text{reverse } (x:\cdot:xs) &= \text{reverse } xs \mathbin{+} [x]
\end{align*}
\]

Much better (why?):

\[
\begin{align*}
\text{reverse } xs &= \text{rev } [] \cdot xs \\
&\quad \text{where rev } acc \cdot [] = acc \\
&\quad \text{rev } acc \cdot (x:xs) = \text{rev } (x:acc) \cdot xs
\end{align*}
\]

This looks a lot like foldl. Indeed, we can redefine reverse as:

\[
\begin{align*}
\text{reverse } xs &= \text{foldl } \text{revOp } [] \cdot xs \\
&\quad \text{where revOp } a \cdot b = b : a
\end{align*}
\]