CSE399: Advanced Programming
Handout 3
Polymorphism
The Length Function is Polymorphic

```haskell
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.

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

\[(++) :: [a] \to [a] \to [a]\]
\[id :: a \to a\]
\[head :: [a] \to a\]
\[tail :: [a] \to [a]\]
\[[] :: [a]\]

-- 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:

\[
(:) :: a \rightarrow [a] \rightarrow [a] \\
(,) :: a \rightarrow b \rightarrow (a,b)
\]

Note the way that the tupling operator is identified, which generalizes to \((,),(,),\), \((,,,,)\), etc. When we write \((1,2,3,4)\), we really mean \((1,(2,(3,4))))\).

We can also define new polymorphic data structures...
The type variable `a` on the left-hand-side of the `=` tells Haskell that `Maybe` is a polymorphic data type:

```haskell
data Maybe a = Nothing | Just a
```

Note the types of the constructors:

- `Nothing :: Maybe a`
- `Just :: a -> Maybe a`

Thus:

- `Just 3 :: Maybe Int`
- `Just "x" :: Maybe String`
- `Just (3,True) :: Maybe (Int,Bool)`
- `Just (Just 1) :: Maybe (Maybe 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 `div` in Haskell will cause a run-time error if its second argument is zero. Thus we may wish to define a safe division function, as follows:

```haskell
safeDivide :: Int -> Int -> Maybe Int
safeDivide x 0 = Nothing
safeDivide x y = Just (x 'div' y)
```
Higher-Order Functions
Recall from Section 4.1:

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

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

Also, from Chapter 3:

\[
\begin{align*}
\text{putCharList} & :: [\text{Char}] \rightarrow [\text{IO ()}] \\
\text{putCharList} \ [\] & = [] \\
\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} \text{ vs. the function putChar}.

We can use the abstraction principle to take advantage of this regularity.
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:

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

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 :: \text{Vertex} \rightarrow \text{Point}`, we have

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

And since `putChar :: \text{Char} \rightarrow \text{IO ()}`,

\[
\text{map putChar} :: [\text{Char}] \rightarrow [\text{IO ()}]
\]
Haskell provides a convenient special syntax for lists of numbers obeying simple rules:

\[
[1 \ldots 6] \Rightarrow [1, 2, 3, 4, 5, 6]
\]
\[
[1, 3 \ldots 9] \Rightarrow [1, 3, 5, 7, 9]
\]
\[
[5, 4 \ldots 1] \Rightarrow [5, 4, 3, 2, 1]
\]
\[
[2.4, 2.1 \ldots 0.3] \Rightarrow [2.4, 2.1, 3.8, 3.5, \text{ etc.}]
\]
circles :: [Shape]
circles = map circle [2.4, 2.1 .. 0.3]

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

```haskell
zip :: [a] -> [b] -> [(a,b)]
zip (a:as) (b:bs) = (a,b) : zip as bs
zip _ _ = []
```

For example:

```haskell
zip [1,2,3] [True, False, False]
⇒ [(1,True), (2,False), (3,False)]
```

Quick check: What does `zip [1..3] [1..5]` yield?
colCircles :: [(Color,Shape)]
colCircles = zip [Red,Blue,Green,
                 Cyan,Red,Magenta,
                 Yellow,White]
circles
drawShapes :: Window -> [(Color,Shape)] -> IO ()

drawShapes w css =
    sequence_ (map aux css)
    where aux (c,s) =
        drawInWindow w
        (withColor c
        (shapeToGraphic s))

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.
The Main Action

g = do w <- openWindow "Bulls eye" (600,600)
    drawShapes w colCircles
    k <- getKey w
    closeWindow w

main = runGraphics g
When to Define Higher-Order Functions

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

```plaintext
listSum [] = 0
listSum (x:xs) = x + listSum xs

listProd [] = 1
listProd (x:xs) = x * listProd xs
```

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 (\textit{op} and \textit{init}) leaves this common part:

\begin{verbatim}
fold op init [] = init
fold op init (x:xs) = x \text{ ‘} op \text{‘} fold op init xs
\end{verbatim}

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

\begin{verbatim}
listSum xs = fold (+) 0 xs
listProd xs = fold (*) 1 xs
\end{verbatim}

Note that \texttt{fold} is polymorphic:

\begin{verbatim}
fold :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow \texttt{[a]} \rightarrow b
\end{verbatim}
The `fold` function is predefined in Haskell, but it is actually called `foldr`, because it “folds from the right.” That is:

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

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

\[
\text{foldl } \text{op} \text{ init} (x_1 : x_2 : \ldots : x_n : []) \Rightarrow (\ldots((\text{init} \ 'op' \ x_1) \ 'op' \ x_2)\ldots) \ 'op' \ x_n
\]
Why two folds? Because sometimes using one can be more efficient than the other. For example:

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

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
```
Obvious but inefficient (why?):

```haskell
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

Much better (why?):

```haskell
reverse xs = rev [] xs
  where rev acc [] = acc
        rev acc (x:xs) = rev (x:acc) xs
```

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

```haskell
reverse xs = foldl revOp [] xs
  where revOp a b = b : a
```