Chapter 5

Polymorphic and

Higher-Order Functions

Polymorphic Length

"**a**" is a type variable. It is lowercase to distinguish it from type names, which are capitalized.

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

Polymorphic functions don't "look at" their polymorphic arguments, and thus don't care what the type is:

```
length [1,2,3] ➔ 3
length ['a','b','c'] ➔ 3
length [[2],[],[1,2,3]] ➔ 3
```

Polymorphism

- Many predefined functions are polymorphic. For example:

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

- But you can define your own as well. For example, suppose we define:

```
tag1 x = (1,x)
```

  Then:

```
Hugs> :type tag1
tag1 :: a -> (Int,a)
```

Polymorphic Data Structures

- Polymorphism is common in data structures that "don't care" what kind of data they contain.

- The examples on the previous page involve *lists* and *tuples*. In particular, note that:
  ```
  (:) :: a -> [a] -> [a]
  (,) :: a -> b -> (a,b)
  ```
  (note the way that the tupling operator is identified – which generalizes to `(,,)` , `(,,,)` , etc.)

- But we can also easily define new data structures that are polymorphic.

Example

- The type variable **a** causes **Maybe** to be polymorphic:

```
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)
```

Maybe may be useful

- 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 **(/)** 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:

```
safeDivide :: Int -> Int -> Maybe Int
safeDivide x 0 = Nothing
safeDivide x y = Just (x/y)
```

Abstraction Over Recursive Definitions

- Recall from Section 4.1:

```
transList []        = []
transList (p:ps)    = trans p : transList ps

putCharList []      = []
putCharList (c:cs)  = putChar c : putCharList cs
```

- There is something strongly similar about these definitions. Indeed, the only thing different about them (besides the variable names) is the function **trans** vs. the function **putChar**.

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

Abstraction Yields `map`

- `trans` and `putChar` are what's different; so they should be arguments to the abstracted function.
- In other words, we would like to define a function called `map` (say) such that `map trans` behaves like `transList`, and `map putChar` behaves like `putCharList`.
- No problem:

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

- Given this, it is not hard to see that we can redefine `transList` and `putCharList` as:

```
transList   xs = map trans   xs
putCharList cs = map putChar cs
```

`map` is Polymorphic

- The greatest thing about map is that it is *polymorphic*. Its most general (i.e. principal) type is:

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

  Note that whatever type is instantiated for "**`a`**" must be the same at both instances of "**`a`**"; the same is true for "**`b`**".

- For example, since **`trans :: Vertex -> Point`**, then

  ```
  map trans :: [Vertex] -> [Point]
  ```

  and since **`putChar :: Char -> IO ()`**, then

  ```
  map putChar :: [Char] -> [IO ()]
  ```

Arithmetic Sequences

- Special syntax for computing lists with regular properties.

```
[1 .. 6]     = [1,2,3,4,5,6]
[1,3 .. 9]   = [1,3,5,7,9]
[5,4 .. 1]   = [5,4,3,2,1]
```

- Infinite lists too!

```
take 9 [1,3..] = [1,3,5,7,9,11,13,15,17]
take 5 [5..]   = [5,6,7,8,9]
```

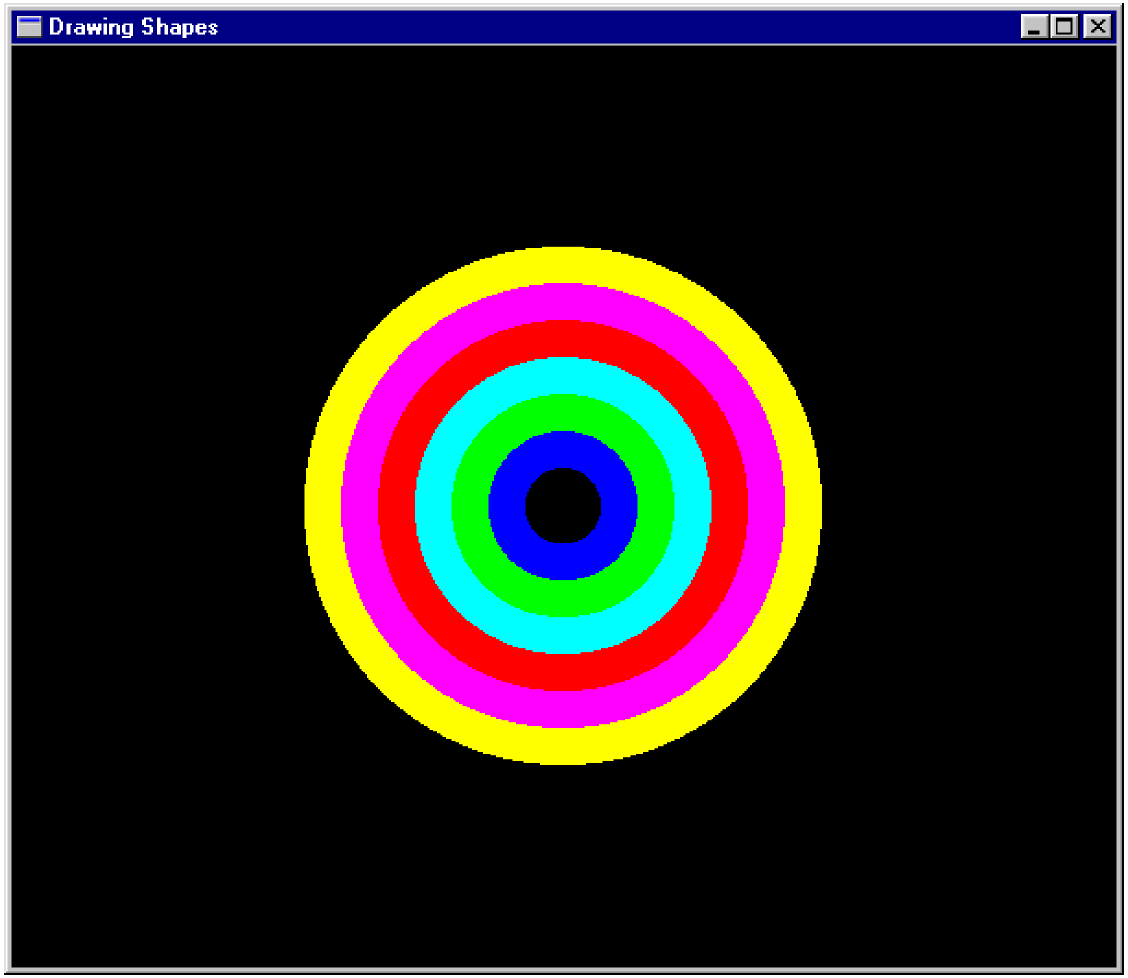
Another Example

```
conCircles = map circle [2.4, 2.1 .. 0.3]

coloredCircles =
  zip [Black,Blue,Green,Cyan,Red,Magenta,Yellow,White]
      conCircles

main =
   runGraphics $
     do w <- openWindow "Drawing Shapes" (xWin,yWin)
        drawShapes w (reverse coloredCircles)
        spaceClose w
```

The Result



When to Define Higher-Order Functions

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

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

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

- Note the similarities. Also note the differences (underlined), which need to become parameters to the abstracted function.

Abstracting

- This leads to:

```
fold op init []     = init
fold op init (x:xs) = x `op` fold op init xs
```

- Note that `fold` is also *polymorphic*:

```
fold :: (a -> b -> b) -> b -> [a] -> b
```

- `listSum` and `listProd` can now be redefined:

```
listSum  xs = fold (+) 0 xs
listProd xs = fold (*) 1 xs
```

Two Folds are Better than One

- **`fold`** is predefined in Haskell, though with the name **`foldr`**, because it "folds from the right". That is:

```
foldr op init (x1 : x2 : ... : xn : [])
➔ x1 `op` (x2 `op` (...(xn `op` init)...))
```

- But there is another function **`foldl`** which "folds from the left":

```
foldl op init (x1 : x2 : ... : xn : [])
➔ (...((init `op` x1) `op` x2)...) `op` xn
```

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

```
foldr (++) [] [x,y,z] ➔ x ++ (y ++ z)
foldl (++) [] [x,y,z] ➔ (x ++ y) ++ z
```

  The former is more efficient than the latter; but not always – sometimes **`foldl`** is more efficient than **`foldr`**. Choose wisely!

Reversing a List

- Obvious but inefficient (why?):

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

- Much better (why?):

```
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:

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