Recursion and Higher-Order Functions

Stephen A. Edwards

Columbia University

Fall 2019
Recursion in Haskell

Pattern matching works nicely:

recfun <base case> = <base value>
recfun <part> <rest> = <some work> <part> <combined with> recfun <rest>

maximum' :: Ord a => [a] -> a
maximum' [] = error "empty list"
maximum' [x] = x -- base case
maximum' (x:xs)
  | x > maxTail = x -- found a new maximum
  | otherwise = maxTail
where maxTail = maximum' xs -- recurse

The list elements need to be ordered so we can perform > on them

maximum is part of the standard prelude; you do not need to write this
But far better to build the solution out of helpful pieces. Don’t worry about efficiency; GHC very aggressively inlines code to avoid function call overhead

```haskell
max' :: Ord a => a -> a -> a
max' a b
    | a > b = a
    | otherwise = b

maximum' :: Ord a => [a] -> a
maximum' [] = error "empty list"
maximum' [x] = x
maximum' (x:xs) = x `max'` maximum' xs
```

This is still twice as complicated as it needs to be; we’ll revisit this later
**Replicate and Take**

\[
\text{replicate'} :: (\text{Num } n, \text{Ord } n) \Rightarrow n \rightarrow a \rightarrow [a]
\]
\[
\text{replicate'} n x
| n \leq 0 = []
| \text{otherwise} = x : \text{replicate'} (n-1) x
\]

Oddly, the Num typeclass (-) does not require Ord (for \(\leq\))

Used a guard since we’re testing a condition \(n \leq 0\) rather than just a constant.

\[
\text{take'} :: (\text{Num } n, \text{Ord } n) \Rightarrow n \rightarrow [a] \rightarrow [a]
\]
\[
\text{take'} n _ | n \leq 0 = [] \quad -- \text{base case}
\]
\[
\text{take'} _ [] = [] \quad -- \text{base case}
\]
\[
\text{take'} n (x:xs) = x : \text{take'} (n-1) xs \quad -- \text{recurse}
\]
Replicate and Take Revisited

The Standard Prelude implementation uses infinite lists

\[
\begin{align*}
take' &: (\text{Num } n, \text{Ord } n) \Rightarrow n \rightarrow [a] \rightarrow [a] \\
take' n \_ | \ n \leq 0 &= [] \\
take' \_ [] &= [] \\
take' n (x:xs) &= x : take' (n-1) \ xs
\end{align*}
\]

\[
\begin{align*}
\text{repeat'} &: a \rightarrow [a] \\
\text{repeat'} x &= xs \quad \text{where} \quad xs = x : xs \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad
\end{align*}
\]

\[
\begin{align*}
\text{replicate'} &: (\text{Num } n, \text{Ord } n) \Rightarrow n \rightarrow a \rightarrow [a] \\
\text{replicate'} n \ x &= take' n (\text{repeat'} x)
\end{align*}
\]
Zip: Combine Two Lists Into a List of Pairs

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

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

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

\[
\text{\textit{zip'}} (x:xs) (y:ys) = (x,y) : \text{\textit{zip'}} xs ys
\]

Works nicely with lists of mismatched lengths, including infinite:

*Main> \text{\textit{zip'}} [0..3] [1..5] :: [(Int, Int)]
[(0,1),(1,2),(2,3),(3,4)]

*Main> \text{\textit{zip'}} "abc" ([1..] :: [Int])
[('a',1),('b',2),('c',3)]
QuickSort in Haskell

- Pick and remove a pivot
- Partition into two lists: smaller or equal to and larger than pivot
- Recurse on both lists
- Concatenate smaller, pivot, then larger

```
quicksort :: Ord a => [a] -> [a]
quicksort [] = []
quicksort (p:xs) = quicksort [x | x <- xs, x <= p] ++ [p] ++ quicksort [x | x <- xs, x > p]
```

Efficient enough: ++ associates to the right so `a ++ b ++ c` is `(a ++ (b ++ c))`
Haskell does not have classical *for* or *do* loops.

Recursion can implement either of these plus much more. Tail-recursion is just as efficient as such loops.

Most of the time, however, your loop or recursive function fits a well-known pattern that is already in a Standard Prelude function that you should use instead.

A key advantage of functional languages, including Haskell, is that you can build **new control constructs**.
The (+) syntax also permits a single argument to be applied on either side and returns a function that take the “missing” argument:

```
Prelude> (+ "", hello"") "Stephen"
"Stephen, hello"
Prelude> ("Hello, " ++) "Stephen"
"Hello, Stephen"

Prelude> (<= (5::Int)) 10
False
Prelude> (<= (5::Int)) 5
True
Prelude> (<= (5::Int)) 4
True
```
Passing functions as arguments is routine yet powerful

```
Prelude> :{
Prelude| applyTwice :: (a -> a) -> a -> a
Prelude| applyTwice f x = f (f x)
Prelude| :}

Prelude> applyTwice (+5) 1
11
Prelude> applyTwice (++) " is stupid") "Stephen"
"Stephen is stupid is stupid"
```

“applyTwice takes a function and return a function that takes a value and applies the function to the value twice”
Map

A fundamental function in the Standard Prelude. Two equivalent encodings:

\[
\begin{align*}
\text{map'} &:: (a \to b) \to [a] \to [b] \\
\text{map'} \_ \ [\] & = [] \\
\text{map'} f (x:xs) & = f x : \text{map'} f xs
\end{align*}
\]

\[
\begin{align*}
\text{map''} &:: (a \to b) \to [a] \to [b] \\
\text{map''} f xs & = [ f x | x <- xs ]
\end{align*}
\]

*Main> map (+5) ([1..5] :: [Int])
[6,7,8,9,10]
*Main> map (++ "!" ) ["BIFF","BAM","POW"]
["BIFF!","BAM!","POW!"]

You’ve written many loops that fit \textit{map} in imperative languages
Another Standard Prelude function \texttt{zipWith} takes a function and two lists and applies the function to the list elements, like a combination of \texttt{zip} and \texttt{map}:

\[
\text{\texttt{zipWith'}} :: (a \to b \to c) \to [a] \to [b] \to [c]
\]

\[
\text{\texttt{zipWith'}} _ [] _ [] = []
\]

\[
\text{\texttt{zipWith'}} _ _ [] = []
\]

\[
\text{\texttt{zipWith'}} f (x:xs) (y:ys) = f x y : \text{\texttt{zipWith'}} f xs ys
\]

Prelude> \texttt{zipWith} (+) [1..5] [10,20..] :: [Int]
[11,22,33,44,55]

The Standard Prelude implements \texttt{zip} with \texttt{zipWith}:

\[
\text{\texttt{zip'}} :: [a] \to [b] \to [(a,b)]
\]
\[
\text{\texttt{zip'}} = \text{\texttt{zipWith'}} (,)
\]
Filter: Select each element of a list that satisfies a predicate

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

```
fILTER :: (a -> Bool) -> [a] -> [a]
FILTER p xs = [ x | x <- xs, p x ]
```

Prelude> filter (>= 3) [1..10] :: [Int]
[3,4,5,6,7,8,9,10]

What’s the largest number under 100,000 that’s divisible by 3,829?

```
Prelude> x `divides` y = y `mod` x == 0
Prelude> head (filter (3829 `divides`) [100000,99999..])
99554
```
Using *filter* instead of list comprehensions:

```haskell
quicksort :: Ord a => [a] -> [a]
quicksort [] = []
quicksort (p:xs) = quicksort (filter (<= p) xs) ++ [p] ++ quicksort (filter (> p) xs)
```

Similar performance; choose the one that’s easier to understand
takeWhile: Select the first elements that satisfy a predicate
Same type signature as filter, but stop taking elements from the list once the predicate is false. Also part of the Standard Prelude

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

Prelude> takeWhile (/= ' ') "Word splitter function"
"Word"

What’s the sum of all odd squares under 10,000?

```
Prelude> sum (takeWhile (<10000) (filter odd (map (^2) [1..])))
166650
Prelude> sum (takeWhile (<10000) [ n^2 | n <- [1..], odd (n^2) ])
166650
```
Twin Primes

Twin Primes differ by two, e.g., 3 and 5, 11 and 13, etc.

Prelude> primes = \_f = \_f [2..] where
Prelude| \_f (p:xs) = p : \_f [ x | x <- xs, x `mod` p /= 0 ]

Prelude> twinPrimes = filter twin (zip primes (tail primes)) where
Prelude| twin (a,b) = a+2 == b

Prelude> take 7 twinPrimes
[(3,5),(5,7),(11,13),(17,19),(29,31),(41,43),(59,61)]

Prelude> length twinPrimes

(Left as an exercise for the reader)
Collatz sequences
For starting numbers between 1 and 100, how many Collatz sequences are longer than 15?

collatz :: \textbf{Int} \rightarrow \textbf{[Int]}
collatz 1 = [1]
collatz n
  | even n = n : collatz (n `div` 2)
  | otherwise = n : collatz (n * 3 + 1)

numLongChains :: \textbf{Int}
umLongChains = \textbf{length} (filter isLong (map collatz [1..100]))
  where isLong xs = \textbf{length} xs > 15

*Main> collatz 30
[30,15,46,23,70,35,106,53,160,80,40,20,10,5,16,8,4,2,1]
*Main> numLongChains
66
A *lambda expression* is an unnamed function. \ is a \( \lambda \) missing a leg:

\[
\lambda \langle \text{args} \rangle \rightarrow \langle \text{expr} \rangle
\]

Things like \((+ \ 5)\) and \(\text{max} \ 5\) are also unnamed functions, but the lambda syntax is more powerful: