CSci 450: Org. of Programming Languages
Higher-Order Functions

H. Conrad Cunningham

9 October 2017

Contents

Acknowledgements: In Summer 2016, I adapted and revised this chapter from chapter 6 of my *Notes on Functional Programming with Haskell* and from my notes on *Functional Data Structures* from the Spring 2016 Scala variant of CSci 555. The latter was based, in part, on chapter 3 of the book *Functional Programming in Scala* by Paul Chiusano and Runar Bjarnason (Manning, 2015).

I also expanded the discussion of of combinators and functional composition and added discussion of `seq`, `$`, and `$!`. These drew on ideas from chapters 4 and 7 of Richard Bird's *Thinking Functionally with Haskell* (Cambridge University Press, 2015) and chapter 11 of Simon Thompson's *Haskell: The Craft of Programming, Third Edition* (Pearson 2011).

In 2017, I continue to develop this chapter.

I maintain these notes as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the notes to HTML, PDF, and other forms as needed.

Advisory: The HTML version of this document may require use of a browser that supports the display of MathML. A good choice as of October 2017 is a recent version of Firefox from Mozilla.

TODO:

- Consider other examples for use of map, filter, fold
- Consider testing
- Consider larger case study
- Edit and add more exercises

5 Higher-Order Functions

5.1 Chapter Introduction

This chapter introduces higher-order functions and constructs a library of useful higher-order functions to process lists.

The goals of the chapter are for the students to be able to:

- develop correct Haskell functional programs that include appropriate higher-order polymorphic functions, functional composition, combinators, and inline function definitions
- develop Haskell programs that terminate and are efficient and elegant.

Upon successful completion of this chapter, students should be able to:

1. describe the syntax and semantics of Haskell function concepts and operators such as strictness, currying, partial application, operator sections, combinators, lambda expressons, and strict function application
2. describe the syntax and semantics of first-class and higher-order Haskell functions
3. develop Haskell programs using using Haskell function concepts and operators such as strictness, currying, partial application, operator sections, combinators, lambda expressons, and strict function application
4. use a library of higher-order Haskell functions to develop Haskell programs
5. generalize and develop higher-order Haskell functions for reuse
6. appreciate the mathematical characteristics of Haskell functions (e.g., to state properties of functions and operators)

The Haskell module for this chapter is in `Mod05HigherOrder.hs`.

5.2 Higher-Order List Programming

5.2.1 Generalizing procedural abstractions

A function in a programming language is a *procedural abstraction*. It separates the logical properties of a computation from the details of how the computation is implemented. It abstracts a pattern of behavior and encapsulates it within a program unit.

Suppose we wish to perform the *same* computation on a set of *similar* data structures. As we have seen, we can encapsulate the computation in a function having the data structure as an argument. For example, the function `length'` computes the number of elements in a list of any type.

Suppose instead we wish to perform a *similar* (but not identical) computation on a set of *similar* data structures. For example, we want to compute the sum or the product of a list of numbers. In this case, we may can pass the operation itself into the function.

This kind of function is called a *higher-order function*. A higher-order function is a function that takes functions as arguments or returns functions in a result. Most traditional imperative languages do not fully support higher-order functions.

In most functional programming languages, functions are treated as *first class* values. That is, functions can be stored in data structures, passed as arguments to functions, and returned as the results of functions. Historically, imperative languages have not treated functions as first-class values. (Recently, many imperative languages, such as Java 8, have added support for functions as first-class values.)

The higher-order functions in Haskell and other functional programming languages enable us to construct regular and powerful abstractions and operations. By taking advantage of a library of higher-order functions that capture common patterns of computation, we can quickly construct concise, yet powerful, programs.

This can increase programmer productivity and program reliability because such programs are shorter, easier to understand, and constructed from well-tested components.

Higher-order functions can also increase the *modularity* of programs by enabling simple program fragments to be "glued together" readily into more complex programs.

In this section, we examine several common patterns and build a library of useful higher-order functions.

5.2.2 Defining `map`

Consider the following two functions, noting their type signatures and patterns of recursion.

The first, `squareAll`, takes a list of integers and returns the corresponding list of squares of the integers.

```
squareAll :: [Int] -> [Int]  squareAll :: [Int] -> [Int]
squareAll []     = []
squareAll (x:xs) = (x * x) : squareAll xs
```

The second, `lengthAll`, takes a list of lists and returns the corresponding list of the lengths of the element lists; it uses the Prelude function `length`.

```
lengthAll :: [[a]] -> [Int]
lengthAll []       = []
lengthAll (xs:xss) = (length xs) : lengthAll xss
```

Although these functions take different kinds of data (a list of integers versus a list of polymorphically typed lists) and apply different operations (squaring versus list length), they exhibit the same pattern of computation. That is, both take a list of some type and apply a given function to each element to generate a resulting list of the same length as the original.

The combination of polymorphic typing and higher-order functions allow us to abstract this pattern of computation into a standard function.

We can abstract the pattern of computation common to `squareAll` and `lengthAll` as the (broadly useful) function `map`, which we define as follows. (In this chapter, we often add a suffix to the base function names to avoid conflicts with the similarly named functions in the Prelude. Here we use `map'` instead of `map`.)

```
map' :: (a -> b) -> [a] -> [b]  -- map in Prelude
map' f []     = []
map' f (x:xs) = f x : map' f xs
```

Function `map` *generalizes* `squareAll`, `lengthAll`, and similar functions by adding a higher-order parameter for the operation applied and making the input and the output lists polymorphic. Specifically, the function takes a function `f` of type `a -> b` and a list of type `[a]`, applies function `f` to each element of the list, and produces a list of type `[b]`.

Thus we can *specialize* `map` to give new definitions of `squareAll` and `lengthAll` as follows:

```
squareAll2 :: [Int] -> [Int]
squareAll2 xs = map' sq xs
                where sq x = x * x
InIn
lengthAll2 :: [[a]] -> [Int]
lengthAll2 xss = map' length xss
```

Consider the following questions.

- Under what circumstances does `map' f xs` terminate? Do we have to assume anything about `f`? about `xs`?
- What is the time complexity of `map f xs`? space complexity?
- What is the time complexity of `squareAll2 xs`? of `lengthAll2 xs`?

5.2.3 Thinking about data transformations

Above we define `map` as a recursive function that transforms the elements of a list one by one. However, it is often more useful to think of `map` in one of two ways:

1. as a powerful list operator that transforms every element of the list. We can combine `map` with other powerful operators to quickly construct powerful list processing programs.

   We can consider `map` as operating on every element of the list "simultaneously". In fact, an implementation could use separate processors to transform each element: this is essentially the `map` operation in Google's `mapReduce` distributed "big data" processing framework.

   Referential transparency and immutable data structures make parallelism easier in Haskell than in most imperative languages.

2. as a operator node in a dataflow network. A stream of data objects flows into the `map` node. The `map` node transforms each object by applying the

argument function. Then the data object flows out to the next node of the network.

The lazy evaluation of the Haskell functions enables such an implementation.

Although in the early parts of these notes we give attention to the details of recursion, learning how to *think like a functional programmer* requires us to think about large-scale transformations of collections of data.

5.2.4 Generalizing function definitions

Whenever we recognize a computational pattern in a set of related functions, we can *generalize the function* definition as follows:

1. Do a *scope-commonality-variability analysis* on the set of related functions.

   That is, identify what is to be included and what not (i.e., the scope), the parts of functions that are the same (the *commonalities* or *frozen spots*), and the parts that differ (the *variabilities* or *hot spots*)

2. Leave the commonalities in the generalized function's body.

3. Move the variabilities into the generalized function's header – its type signature and parameter list.

   - If the part moved to the generalized function's parameter list is an expression, then make that part a function with a parameter for each local variable accessed.
   - If a data type potentially differs from a specific type used in the set of related functions, then add a type parameter to the generalized function.
   - If the same data value or type appears in multiple roles, then consider adding distinct type or value parameters for each role.

4. Consider other approaches if the generalized function's type type signature and parameter list become too complex.

   For example, we can introduce new data or procedural abstractions for parts of the generalized function. These may be in the same module of the generalized function or in an appropriately defined separate module.

5.2.5 Defining `filter`

Consider the following two functions.

The first, `getEven`, takes a list of integers and returns the list of those integers that are even (i.e., are multiples of 2). The function preserves the relative order of the elements in the list.

```
getEven :: [Int] -> [Int]
getEven []      = []
getEven (x:xs)
    | even x    = x : getEven xs
    | otherwise = getEven xs
```

The second, `doublePos`, takes a list of integers and returns the list of doubles of the positive integers from the input list; it preserves the relative order of the elements.

```
doublePos :: [Int] -> [Int]
doublePos []          = []
doublePos (x:xs)
    | 0 < x     = (2 * x) : doublePos xs
    | otherwise = doublePos xs
```

Function `even` is from the Prelude; it returns `True` if its argument is evenly divisible by 2 and returns `False` otherwise.

What do these two functions have in common? What differs?

- Both take a list of integers and return a (possibly shorter) list of integers.

  However, the fact they use integers is not important; the key fact is that they take and return lists of the same element type.

- Both return an empty list when its input list is empty.

- In both, the relative orders of elements in the output list is the same as in the input list.

- Both select some elements to copy to the output and others not to copy.

  Function `getEven` selects elements that are even numbers and function `doublePos` selects elements that are positive numbers.

- Function `doublePos` doubles the value copied and `getEven` leaves the value unchanged.

Using the generalization method outlined above, we abstract the pattern of computation common to `getEven` and `doublePos` as the (broadly useful) function `filter` found in the Prelude. (We call the function `filter'` below to avoid a name conflict.)

```
filter' :: (a -> Bool) -> [a] -> [a]  -- filter in Prelude
filter' _ []    = []
filter' p (x:xs)
    | p x       = x : xs'
```

```
    | otherwise = xs'
                where xs' = filter' p xs
```

Function `filter` takes a predicate `p` of type `a -> Bool` and a list of type `[a]` and returns a list containing those elements that satisfy `p`, in the same order as the input list. Note that the keyword `where` begins in the same column as the `=` in the defining equations; thus the scope of the definition of `xs'` extends over *both* legs of the definition.

Function `filter` does not incorporate the doubling operation from `doublePos`. We could have included it as another higher-order parameter, but we leave it out to keep the generalized function simple. We can use the already defined `map` function to achieve this separately.

Therefore, we can specialize `filter` to give new definitions of `getEven` and `doublePos` as follows:

```
getEven2 :: [Int] -> [Int]
getEven2 xs = filter' even xs

doublePos2 :: [Int] -> [Int]
doublePos2 xs = map' dbl (filter' pos xs)
                where dbl x  = 2 * x
                      pos x = (0 < x)
```

Note that function `doublePos2` exhibits both the `filter` and the `map` patterns of computation.

The standard higher-order functions `map` and `filter` allow us to restate the three-leg definitions of `getEven` and `doublePos` in just one leg each, except that `doublePos` requires two lines of local definitions. In subsequent subsections, we see how to eliminate these simple local definitions as well.

Consider the following questions.

- Under what circumstances does `filter' p xs` terminate? Do we have to assume anything about `p`? about `xs`?
- What is the time complexity of `filter' p xs`? space complexity?
- What is the time complexity of `getEven2 xs`? space complexity?
- What is the time complexity of `doublePos2 xs`? space complexity?

5.2.6 Defining fold right (`foldr`)

Consider the `sum` and `product` functions we defined in a previous chapter, ignoring the short-cut handling of the zero element in `product`.

```
sum' :: [Int] -> Int              -- sum in Prelude
sum' []     = 0
```

```
sum' (x:xs) = x + sum' xs

product' :: [Integer] -> Integer -- product in Prelude
product' []     = 1
product' (x:xs) = x * product' xs
```

Both `sum'` and `product'` apply arithmetic operations to integers. What about other operations with similar pattern of computation?

Also consider a function `concat` that concatenates a list of lists of some type into a list of that type with the order of the input lists and their elements preserved.

```
concat' :: [[a]] -> [a]  -- concat in Prelude
concat' []       =  []
concat' (xs:xss) =  xs ++ concat' xss
```

For example,

```
sum' [1,2,3]           = (1 + (2 + (3 + 0)))
product' [1,2,3]       = (1 * (2 * (3 * 1)))
concat' ["1","2","3"] = ("1" ++ ("2" ++ ("3" ++ "")))
```

What do `sum'`, `product'`, and `concat'` have in common? What differs?

All exhibit the same pattern of computation.

- All take a list.

  But the element type differs. Function `sum'` takes a list of `Int`s, `product'` takes a list of `Integer`s, and `concat'` takes a polymorphic list.

- All insert a binary operator between all the consecutive elements of the list in order to reduce the list to a single value.

  But the binary operation differs. Function `sum'`applies integer addition, `product'`applies integer multiplication, and `concat'` applies `++`.

- All group the operations from the right to the left.

- Each function returns some value for an empty list. The function extends nonempty input lists to implicitly include this value as the "rightmost" value of the input list.

  But the actual value differs.

  Function `sum'` returns integer 0, the (right) identity element for addition.

  Function `product'` returns 1, the (right) identity element for multiplication.

  Function `concat'` returns `[]`, the (right) identity element for `++`.

  In general, this value could be something other than the identity element,

- All return a value of the same element type as the input list.

  But the input type differs, as we noted above.

This group of functions inserts operations of type `a -> a -> a` between elements a list of type `[a]`.

But these are special cases of more general operations of type `a -> b -> b`. In this case, the value returned must be of type `b` in the case of both empty and nonempty lists.

We can abstract the pattern of computation common to `sum'`, `product'`, and `concat'` as the function `foldr` (pronounced "fold right") found in the Prelude. (Here we use `foldrX` to avoid the name conflict.)

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

Function `foldr`:

- uses two type parameters `a` and `b` – one for the type of elements in the list and one for the type of the result
- passes in the general binary operation `f` (with type `a -> b -> b`) that combines (i.e., folds) the list elements
- passes in the "seed" element `z` (of type `b`) to be returned for empty lists

The `foldr` function "folds" the list elements (of type `a`) into a value (of type `b`) by "inserting" operation `f` between the elements, with value `z` "appended" as the rightmost element.

Often the seed value `z` is the right identity element for the operation, but `foldr` may be useful in some circumstances where it is not (or perhaps even if there is no right identity).

For example, `foldr f z [1,2,3]` expands to `f 1 (f 2 (f 3 z))`, or, using an infix style:

```
1 `f` (2 `f` (3 `f` z))
```

Function `foldr` does not depend upon `f` being associative or having either a right or left identity.

Function `foldr` is backward recursive. If the function application is fully evaluated, it needs a new stack frame for each element of the input list. If its list argument is long or the folding function itself is expensive, then the function can terminate with a *stack overflow* error.

In Haskell, `foldr` is called a *fold* operation. Other languages sometimes call this a *reduce* or *insert* operation.

We can specialize `foldr` to restate the definitions for `sum'`, `product'`, and `concat'`.

```
sum2 :: [Int] -> Int              -- sum
sum2 xs = foldrX (+) 0 xs
```

```
product2 :: [Int] -> Int          -- product
product2 xs = foldrX (*) 1 xs

concat2:: [[a]] -> [a]            -- concat
concat2 xss = foldrX (++) [] xss
```

As further examples, consider the folding of the Boolean operators `&&` ("and") and `||` ("or") over lists of Boolean values as Prelude functions `and` and `or` (shown as `and'` and `or'` below to avoid name conflicts):

```
and', or' :: [Bool] -> Bool  -- and, or in Prelude
and' xs = foldrX (&&) True xs
or'  xs = foldrX (||) False xs
```

Although their definitions look different, `and'` and `or'` are actually identical to functions `and` and `or` in the Prelude.

Consider the following questions.

- Under what circumstances does `foldrX f z xs` terminate? Do we have to assume anything about `f`? about `xs`?
- What is the time complexity of `product2`? of `concat2`?

5.2.7 Using `foldr`

The fold functions are very powerful. By choosing an appropriate folding function argument, many different list functions can be implemented in terms of `foldr`.

For example, we can implement `map` using `foldr` as follows:

```
map2 :: (a -> b) -> [a] -> [b] -- map
map2 f xs = foldr mf [] xs
    where mf y ys = (f y) : ys
```

The folding function `mf y ys = (f y):ys` applies the mapping function `f` to the next element of the list (moving right to left) and attaches the result on the front of the processed tail. This is a case where the folding function `mf` does not have a right identity, but where `foldr` is quite useful.

We can also implement `filter` in terms of `foldr` as follows:

```
filter2 :: (a -> Bool) -> [a] -> [a]  -- filter
filter2 p xs = foldr ff [] xs
    where ff y ys = if p y then (y:ys) else ys
```

The folding function `ff y ys = if p x then (y:ys) else ys` applies the filter predicate `p` to the next element of the list (moving right to left). If the predicate evaluates to `True`, the folding function attaches that element on the front of the processed tail; otherwise, it omits the element from the result.

We can also use `foldr` to compute the length of a polymorphic list.

```
length2 :: [a] -> Int  -- length
length2 xs  = foldr len  0  xs
    where len _ acc = acc + 1
```

This uses the `z` parameter of `foldr` to initialize the count to 0. Higher order argument `f` of `foldr` is a function that takes an element of the list as its left argument and the previous accumulator as its right argument and returns the accumulator incremented by 1. In this application, `z` is not the identity element for `f` but is a convenient beginning value for the counter.

We can construct an "append" function that uses `foldr` as follows:

```
append2 :: [a] -> [a] -> [a]  -- ++
append2 xs ys = foldr (:) ys xs
```

Here the the list that `foldr` operates on the first argument of the append. The `z` parameter is the entire second argument and the folding function is just `(:)`. So the effect is to replace the `[]` at the end of the first list by the entire second list.

Function `foldr` 1s a backward recursive function that processes the elements of a list one by one. However, as we have seen, it is often more useful to think of `foldr` as a powerful list operator that reduces the element of the list into a single value. We can combine `foldr` with other operators to conveniently construct list processing programs.

5.2.8 Defining fold left (`foldl`)

We designed function `foldr` as a backward linear recursive function with the signature:

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

As noted:

```
foldr f z [1,2,3] ==  f 1 (f 2 (f 3 z))
                  ==  1 `f` (2 `f` (3 `f` z))
```

Consider a function `foldl` (pronounced "fold left") such that:

```
foldl f z [1,2,3]  ==  f (f (f z 1) 2) 3
                   ==  ((z `f` 1) `f` 2) `f` 3`
```

This function folds from the left. It offers us the opportunity to use parameter `z` as an accumulating parameter in a tail recursive implementation. This is shown below as `foldlX`, which is similar to `foldl` in the Prelude.

```
foldlX :: (a -> b -> a) -> a -> [b] -> a  -- foldl in Prelude
foldlX f z []     = z
foldlX f z (x:xs) = foldlX f (f z x) xs
```

Note how the second leg of `foldlX` implements the left binding of the operation. In the recursive call of `foldlX` the "seed value" argument is used as an accumulating parameter.

Also note how the types of `foldr` and `foldl` differ.

Often the beginning value of `z` is the left identity of the operation `f`, but `foldl` (like `foldr`) can be a quite useful function in circumstances when it is not (or when `f` has no left identity).

5.2.9 Using `foldl`

If $\oplus$ is an associative binary operation of type `t -> t -> t` with identity element `z` (i.e., $\oplus$ and `t` form the algebraic structure know as a monoid), then, for any `xs`,

```
foldr (⊕) z xs = foldl (⊕) z xs
```

The classic Bird and Wadler textbook [Bird-Wadler, 1998] calls this property the *first duality theorem*.

Because `+`, `*`, and `++` are all associative operations with identity elements, `sum`, `product`, and `concat` can all be implemented with either `foldr` or `foldl`.

Which is better?

Depending upon the nature of the operation, an implementation using `foldr` may be more efficient than `foldl` or vice versa.

We defer a more complete discussion of the efficiency until we study evaluation strategies further in a later chapter.

As a rule of thumb, however, if the operation $\oplus$ is *nonstrict* in either argument, then it is usually better to use `foldr`. That form takes better advantage of lazy evaluation.

If the operation $\oplus$ is *strict* in both arguments, then it is often better (i.e., more efficient) to use the optimized version of `foldl` called `foldl'` from the standard Haskell module `Data.List`.

The append operation `++` is nonstrict in its second argument, so it is better to use `foldr` to implement `concat`.

Addition and multiplication are strict in both arguments, so we can implement `sum` and `product` functions efficiently with `foldl'`, as follows:

```
import Data.List   -- to make foldl' available
sum3, product3 :: Num a =>  [a] -> a -- sum, product
sum3 xs     = foldl' (+) 0 xs
product3 xs = foldl' (*) 1 xs
```

Note that we generalize these functions to operate on polymorphic lists with a base type in class `Num`. Class `Num` includes all numeric types.

Function `length3` uses `foldl`. It is like `length2` except that the arguments of function `len` are reversed.

```
length3 :: [a] -> Int  -- length
length3 xs  = foldl len 0  xs
    where len acc _ = acc + 1
```

However, it is usually better to use the `foldr` version `length2` because the folding function `len` is nonstrict in the argument corresponding to the list.

We can also implement list reversal using `foldl` as follows:

```
reverse2 :: [a] -> [a]  -- reverse
reverse2 xs = foldl rev [] xs
    where rev acc x = (x:acc)
```

This gives a solution similar to the tail recursive `reverse` function from a previous chapter. Function `foldl`'s `z` parameter is initially an empty list; `foldl`'s folding function parameter `f` uses `(:)` to "attach" each element of the list as the new head of the accumulator, incrementally building the list in reverse order.

Although cons is nonstrict in its right operand, `reverse2` builds up that argument from `[]`, so `reverse2` cannot take advantage of lazy evaluation by using `foldr` instead.

To avoid a stack overflow situation with `foldr`, we can first apply `reverse` to the list argument and then apply `foldl` as follows:

```
foldr2 :: (a -> b -> b) -> b -> [a] -> b  -- foldr
foldr2 f z xs = foldl flipf z (reverse xs)
    where flipf y x = f x y
```

The combining function in the call to `foldl` is the same as the one passed to `foldr` except that its arguments are reversed.

5.2.10 Defining `concatMap` (flatmap)

The higher-order function `map` applies its function argument `f` to every element of a list and returns the list of results. If the function argument `f` returns a list, then the result is a list of lists. Often we wish to flatten this into a single list, that is, apply a function like `concat` defined in a previous section.

This computation is sufficiently common that we give it the name `concatMap`. We can define it in terms of `map` and `concat` as

```
concatMap' :: (a -> [b]) -> [a] -> [b]
concatMap' f xs = concat (map f xs)
```

or by combining `map` and `concat` into one `foldr` as:

```
concatMap2 :: (a -> [b]) -> [a] -> [b]
concatMap2 f xs = foldr fmf [] xs
    where fmf x ys = f x ++ ys
```

Above, the function argument to `foldr` applies the `concatMap` function argument `f` to each element of the list argument and then appends the resulting list in front of the result from processing the elements to the right.

We can also define `filter` in terms of `concatMap` as follows:

```
filter3 :: (a -> Bool) -> [a] -> [a]
filter3 p xs = concatMap' fmf xs
    where fmf x = if p x then [x] else []
```

The function argument to `concatMap` generates a one-element list if the filter predicate `p` is true and an empty list if it is false.

Some other languages (e.g., Scala) call the `concatMap` function by the name `flatmap`.

5.3 Haskell Function Concepts and Notation

5.3.1 Strictness

Some expressions cannot be reduced to a simple value, for example, `div 1 0`. The attempted evaluation of such expressions either return an error immediately or cause the interpreter to go into an "infinite loop".

In our discussions of functions, it is often convenient to assign the symbol $\perp$ (pronounced "bottom") as the value of expressions like `div 1 0`. We use $\perp$ is a polymorphic symbo–as a value of every type.

The symbol $\perp$ is not in the Haskell syntax and the interpreter cannot actually generate the value $\perp$. It is merely a name for the value of an expression in situations where the expression cannot really be evaluated. It's use is somewhat analogous to use of symbols such as ∞ in mathematics.

Although we cannot actually produce the value $\perp$, we can, conceptually at least, apply any function to $\perp$.

If `f` $\perp = \perp$, then we say that the function is *strict*; otherwise, it is *nonstrict* (sometimes called *lenient*).

That is, a strict argument of a function must be evaluated before the final result can be computed. A nonstrict argument of a function may not need to be evaluated to compute the final result.

Assume that lazy evaluation is being used and consider the function `two` that takes an argument of any type and returns the integer value two.

```
two :: a -> Int
two x = 2
```

The function `two` is nonstrict. The argument expression is not evaluated to compute the final result. Hence, `two` $\perp = 2$.

Consider the following examples.

- The arithmetic operations (e.g., `+`) are strict in both arguments.
- Function `rev` (discussed in a previous chapter) is strict in its one argument.
- Operation `++` is strict in its first argument, but nonstrict in its second argument.
- Boolean functions `&&` and `||` from the Prelude are also strict in their first arguments and nonstrict in their second arguments.

  ~{.haskell} (&&), (||) :: Bool -> Bool -> Bool False && x = False – second argument not evaluated True && x = x

  ```
  False || x = x
  True  || x = True   -- second argument not evaluated
  ```

  ~

5.3.2 Currying and partial application

Consider the following two functions:

```
add :: (Int,Int) -> Int
add (x,y) = x + y

add' :: Int -> (Int -> Int)
add' x y  = x + y
```

These functions are closely related, but they are not identical.

For all integers `x` and `y`, `add (x,y) = add' x y`. But functions `add` and `add'` have different types.

Function `add` takes a 2-tuple `(Int,Int)` and returns an `Int`. Function `add'` takes an `Int` and returns a function of type `Int -> Int`.

What is the result of the application `add 3`? An error.

What is the result of the application `add' 3`? The result is a function that "adds 3 to its argument".

What is the result of the application `(add' 3) 4`? The result is the integer value .

By convention, function application (denoted by the juxtaposition of a function and its argument) binds to the left. That is, `add' x y = ((add' x) y)`.

Hence, the higher-order functions in Haskell allow us to replace any function that takes a tuple argument by an equivalent function that takes a sequence of simple arguments corresponding to the components of the tuple. This process is called *currying*. It is named after American logician Haskell B. Curry, who first exploited the technique.

Function `add'` above is similar to the function `(+)` from the Prelude (i.e., the addition operator).

We sometimes speak of the function `(+)` as being *partially applied* in the expression `((+) 3)`. In this expression, the first argument of the function is "frozen in" and the resulting function can be passed as an argument, returned as a result, or applied to another argument.

Partially applied functions are very useful in conjunction with other higher-order functions.

For example, consider the partial applications of the relational comparison operator `(<)` and multiplication operator `(*)` in the function `doublePos3`. This function, which is equivalent to the function `doublePos` discussed in an earlier subsection, doubles the positive integers in a list:

```
doublePos3 :: [Int] -> [Int]
doublePos3 xs = map' ((*) 2) (filter' ((<) 0) xs)
```

Related to the concept of currying is the *property of extensionality*. Two functions `f` and `g` are extensionally equal if `f x = g x` for all `x`.

Thus instead of writing the definition of `g` as

```
f, g :: a -> a
f x = some_expression

g x = f x
```

we can write the definition of `g` as simply:

```
g = f
```

5.3.3 Operator sections

Expressions such as `((*) 2)` and `((<) 0)`, used in the definition of `doublePos3` in the previous subsection, can be a bit confusing because we normally use these operators in infix form. (In particular, it is difficult to remember that `((<) 0)` returns `True` for positive integers.)

Also, it would be helpful to be able to use the division operator to express a function that halves (i.e., divides by two) its operand. The function `((/) 2)` does not do it; it divides 2 by its operand.

We can use the function `flip` from the Prelude to state the halving operation. Function `flip` takes a function and two additional arguments and applies the argument function to the two arguments with their order reversed.

```
flip' :: (a -> b -> c) -> b -> a -> c  -- flip in Prelude
flip' f x y = f y x
```

Thus we can express the halving operator with the expression `(flip (/) 2)`.

Because expressions such as `((<) 0)` and `(flip (/) 2)` are quite common in programs, Haskell provides a special, more compact and less confusing, syntax.

For some infix operator $\oplus$ and arbitrary expression `e`, expressions of the form `(e⊕)` and `(⊕e)` represent `((⊕) e)` and `(flip (⊕) e)`, respectively. Expressions of this form are called *operator sections*.

Examples of operator sections include:

`(1+)` is the successor function, which returns the value of its argument plus 1.

`(0<)` is a test for a positive integer.

`(/2)` is the halving function.

`(1.0/)` is the reciprocal function.

`(:[])` is the function that returns the singleton list containing the argument.

Suppose we want to sum the cubes of list of integers. We can express the function in the following way:

```
sumCubes :: [Int] -> Int
sumCubes xs = sum' (map' (^3) xs)
```

Above `^` is the exponentiation operator and `sum` is the list summation function defined in the Prelude as:

```
sum' = foldl' (+) 0  -- sum
```

5.3.4 Combinators

The function `flip` in the previous subsection is an example of a useful type of function called a combinator.

A *combinator* is a function without any free variables. That is, on right side of a defining equation there are no variables or operator symbols that are not bound on the left side of the equation.

For historical reasons, `flip` is sometimes called the `C` combinator.

There are several other useful combinators in the Prelude.

The combinator `const` (shown below as `const'`) is the constant function constructor; it is a two-argument function that returns its first argument. For historical reasons, this combinator is sometimes called the `K` combinator.

```
const' :: a -> b -> a  -- const in Prelude
const' k x = k
```

Example: `(const 1)` takes any argument and returns the value 1.

Question: What does `sum (map (const 1) xs)` do?

Function `id` (shown below as `id'`) is the identity function; it is a one-argument function that returns its argument unmodified. For historical reasons, this function is sometimes called the `I` combinator.

```
id' :: a -> a  -- id in Prelude
id' x = x
```

Combinators `fst` and `snd` (shown below as `fst'` and `snd'`) extract the first and second components, respectively, of 2-tuples.

```
fst' :: (a,b) -> a  -- fst in Prelude
fst' (x,_) = x

snd' :: (a,b) -> b  -- snd in Prelude
snd' (_,y) = y
```

Similarly, `fst3`, `snd3`, and `thd3` extract the first, second, and third components, respectively, of 3-tuples.

An interesting example that uses a combinator is the function `reverse` as defined in the Prelude (shown below as `reverse'`):

```
reverse' :: [a] -> [a]          -- reverse in Prelude
reverse' = foldlX (flip' (:)) []
```

Function `flip (:)` takes a list on the left and an element on the right. As this operation is folded through the list from the left it attaches each element as the new head of the list.

We can also define combinators that convert an uncurried function into a curried function and vice versa. The functions `curry'` and `uncurry'` defined below are similar to the Prelude functions.

```
curry' :: ((a, b) -> c) -> a -> b -> c -- curry in Prelude
curry' f x y =  f (x, y)

uncurry' :: (a -> b -> c) -> ((a, b) -> c) -- uncurry in Prelude
uncurry' f p =  f (fst p) (snd p)
```

Two other useful combinators are `fork` and `cross`. Combinator `fork` applies each component of a pair of functions to a value to create a pair of results. Combinator `cross` applies each component of a pair of functions to the corresponding components of a pair of values to create a pair of results. We can define these as follows:

```
fork :: (a -> b, a -> c) -> a -> (b,c)
fork (f,g) x = (f x, g x)

cross :: (a -> b, c -> d) -> (a,c) -> (b,d)
cross (f,g) (x,y) = (f x, g y)
```

5.3.5 Functional composition

The functional composition operator allows several "smaller" functions to be combined to form "larger" functions. In Haskell, this combinator is denoted by the period (`.`) symbol and is defined in the Prelude as follows:

```
infixr 9 .
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x = f (g x)
```

Composition's default binding is from the right and its precedence is higher than all the operators we have discussed so far except function application itself.

Functional composition is an associative binary operation with the identity function `id` as its identity element:

```
f . (g . h) = (f . g) . h
id . f  = f . id
```

An advantage of the use of functional composition is that some expressions can be written more concisely. For example, the function

```
doit x = f1 (f2 (f3 (f4 x)))
```

can be written more concisely as:

```
doit = f1 . f2 . f3 . f4
```

This latter definition is in what is called the *point-free style*. It defines function `doit` as being equal to the composition of the other functions. It leaves the parameters (i.e., the points) of `doit` as implicit; `doit` has the same parameters as the composition.

The former definition (that gives all the explicit parameters) is in what is called *pointful style*.

If `doit` is defined in point-free style as above, then we can use the function in expressions such as `map (doit) xs`. Of course, if this is the only use of the `doit`

function, we can eliminate it completely and use the composition directly, e.g., `map (f1 . f2 . f3 . f4) xs`.

As an example, consider the function `count` that takes two arguments, an integer `n` and a list of lists, and returns the number of the lists from the second argument that are of length `n`. Note that all functions composed below are single-argument functions: `length`, `(filter (== n))`, `(map length)`.

```
count :: Int -> [[a]] -> Int
count n
    | n >= 0    = length' . filter' (== n) . map' length'
    | otherwise = const' 0   -- discard 2nd arg, return 0
```

We can think of the point-free expression `length' . filter' (== n) . map' length'` as defining a *function pipeline* through which data flows from right to left.

1. The pipeline takes a polymorphic list of lists as input.
2. The `map' length'` component of the pipeline replaces each inner list by its length.
3. The `filter' (== n)` component takes the list created by the previous step and removes all elements not equal to `n`.
4. The `length'` component takes the list created by the previous step and determines how many elements are remaining.
5. The pipeline outputs the value computed by the previous component. The number of lists within the input list of lists that are of length `n`.

Thus composition is a powerful form of "glue" that can be used to "stick" simpler functions together to build more powerful functions. The simpler functions in this case include partial applications of higher order functions from the library we have developed.

As we see above in the definition of `count`, partial applications (e.g., `filter (== n)`), operator sections (e.g., `(== n)`), and combinators (e.g., `const`) are useful as *plumbing* the function pipeline.

Remember the function `doublePos` that we discussed in earlier subsections.

```
doublePos3 xs = map' ((*) 2) (filter' ((<) 0) xs)
```

Using composition, partial application, and operator sections we can restate its definition in point-free style as follows:

```
doublePos4 :: [Int] -> [Int]
doublePos4 = map' (2*) . filter' (0<)
```

Consider a function `last` to return the last element in a non-nil list and a function `init` to return the initial segment of a non-nil list (i.e., everything except the last element). These could quickly and concisely be written as follows:

```
last' = head . reverse'             -- last in Prelude
init' = reverse' . tail . reverse' -- init in Prelude
```

However, since these definitions are not very efficient, the Prelude implements functions `last` and `init` in a more direct and efficient way similar to the following:

```
last2 :: [a] -> a    -- last in Prelude
last2 [x]    = x
last2 (_:xs) = last2 xs

init2 :: [a] -> [a]  -- init in Prelude
init2 [x]    = []
init2 (x:xs) = x : init2 xs
```

The definitions for Prelude functions `any` and `all` are similar to the definitions show below; they take a predicate and a list and apply the predicate to each element of the list, returning `True` when any and all, respectively, of the individual tests evaluate to `True`.

```
any', all' :: (a -> Bool) -> [a] -> Bool
any' p = or' . map' p   -- any in Prelude
all' p = and' . map' p  -- all in Prelude
```

The functions `elem` and `notElem` test for an object being an element of a list and not an element, respectively. They are defined in the Prelude similarly to the following:

```
elem', notElem' :: Eq a => a -> [a] -> Bool
elem'    = any' . (==)  -- elem in Prelude
notElem' = all' . (/=)  -- notElem in Prelude
```

These are a bit more difficult to understand since `any`, `all`, `==`, and `/=` are two-argument functions. Note that expression `elem x xs` would be evaluated as follows:

```
     elem' x xs
==>  { expand elem' }
     (any' . (==)) x xs
==>  { expand composition }
     any' ((==) x) xs
```

The composition operator binds the first argument with `(==)` to construct the first argument to `any'`. The second argument of `any'` is the second argument of `elem'`.

5.3.6 Lambda expressions

Remember the function `squareAll2` we examined in an earlier subsection on maps:

```
squareAll2 :: [Int] -> [Int]
squareAll2 xs = map' sq xs
                where sq x = x * x
```

We introduced the local function definition `sq` to denote the function to be passed to `map`. It seems to be a waste of effort to introduce a new symbol for a simple function that is only used in one place in an expression. Would it not be better, somehow, to just give the defining expression itself in the argument position?

Haskell provides a mechanism to do just that, an anonymous function definition. For historical reasons, these nameless functions are called *lambda expressions*. They begin with a backslash `\` and have the syntax:

`\` *atomicPatterns* `->` *expression*

For example, the squaring function (`sq`) could be replaced by a lambda expression as `(\x -> x * x)`. The pattern `x` represents the single argument for this anonymous function and the expression `x * x` is its result.

Thus we can rewrite `squareAll2` in point-free style using a lambda expression as follows:

```
squareAll3 :: [Int] -> [Int]
squareAll3 = map' (\x -> x * x)
```

A lambda expression to average two numbers can be written `(\x y -> (x+y)/2)`.

An interesting example that uses a lambda expression is the function `length` as defined in the Prelude—similar to `length4` below. (Note that this uses the optimized function `foldl'` from the standard Haskell `Data.List` module.)

```
length4 :: [a] -> Int   -- length in Prelude
length4  = foldl' (\n _ -> n+1) 0
```

The anonymous function `(\n _ -> n+1)` takes an integer "counter" and a polymorphic value and returns the "counter" incremented by one. As this function is folded through the list from the left, this function counts each element of the second argument.

5.3.7 Application operator $

In Haskell, function application associates to the left and has higher binding power than any infix operator. For example, for some function two-argument function `f` and values `w`, `x`, `y`, and `z`

```
w + f x y * z
```

is the same as

```
w + (((f x) y) * z)
```

given the relative binding powers of function application and the numeric operators.

However, sometimes we want to be able to use function application where it associates to the right and binds less tightly than any other operator. Haskell defines the `$` operator to enable this style, as follows:

```
infixr 0 $
($) :: (a -> b) -> a -> b
f $ x = f x
```

Thus, for single argument functions `f`, `g`, and `h`,

```
f $ g $ h $ z + 7
```

is the same as

```
(f (g (h (z+7))))
```

and as:

```
(f . g . h) (z+7)
```

Similarly, for two-argument functions `f'`, `g'`, and `h'`,

```
f' w $ g' x $ h' y $ z + 7
```

is the same as

```
((f' w) ((g' x) ((h' y) (z+7))))
```

and as:

```
(f' w . g' x . h' y) (z+7)
```

For example, this operator allows us to write

```
foldr (+) 0 $ map (2*) $ filter odd $ enumFromTo 1 20
```

where Prelude function `enumFromTo m n` generates the sequence of integers from `m` to `n`, inclusive.

5.3.8 Eager evaluation using `seq` and `$!`

Haskell is a lazily evaluated language. That is, if an argument is nonstrict it may never be evaluated.

Sometimes, using the technique called *strictness analysis*, the Haskell compiler can detect that an argument's value will always be needed. The compiler can

then safely force eager evaluation as an optimization without changing the meaning of the program.

In particular, by selecting the `-O` option to the Glasgow Haskell Compiler (GHC), we can enable GHC's code optimization processing. GHC will generally create smaller, faster object code at the expense of increased compilation time by taking advantage of strictness analysis and other optimizations.

However, sometimes we may want to force eager evaluation explicitly without invoking a full optimization on all the code (e.g., to make a particular function's evaluation more space efficient). Haskell provides the primitive function `seq` that enables this. That is,

```
seq :: a -> b -> b
x `seq` y = y
```

where it just returns the second argument except that, as a side effect, `x` is evaluated before `y` is returned. (Technically, `x` is evaluated to what is called *head normal form*. It is evaluated until the outer layer of structure such as `h:t` is revealed, but `h` and `t` themselves are not fully evaluated. We examine evaluation in more detail in a later chapter.)

Function `foldl`, the "optimized" version of `foldl` can be defined using `seq` as follows

```
foldlP :: (a -> b -> a) -> a -> [b] -> a  -- foldl' in Data.List
foldlP f z []     = z
foldlP f z (x:xs) = y `seq` foldl' f y xs
                    where y = f z x
```

That is, this evaluates the `z` argument of the tail recursive application eagerly.

Using `seq`, Haskell also defines `$!`, a strict version of the `$` operator, as follows:

```
infixr 0 $!
($!) :: (a -> b) -> a -> b
f $! x = x `seq` f x
```

The effect of `f $! x` is the same as `f $ x` except that `$!` eagerly evaluates the argument `x` before applying function `f` to it.

We can rewrite `foldl'` using `$!` as follows:

```
foldlQ :: (a -> b -> a) -> a -> [b] -> a  -- foldl' in Data.List
foldlQ f z []     = z
foldlQ f z (x:xs) = (foldlQ f $! f z x) xs
```

We can write a tail recursive function to sum the elements of the list as follows:

```
sum4 :: [Integer] -> Integer  -- sum in Prelude
sum4 xs = sumIter xs 0
    where sumIter []     acc = acc
          sumIter (x:xs) acc = sumIter xs (acc+x)
```

We can then redefine `sum4` to force eager evaluation of the accumulating parameter of `sumIter` as follows:

```
sum5 :: [Integer] -> Integer -- sum in Prelude
sum5 xs = sumIter xs 0
    where sumIter []     acc = acc
          sumIter (x:xs) acc = sumIter xs $! acc + x
```

However, we need to be careful in applying `seq` and `$!`. They change the semantics of the lazily evaluated language in the case where the argument is nonstrict. They may force a program to terminate abnormally and/or cause it to take unnecessary evaluation steps.

5.4 Additional Higher-Order List Functions

5.4.1 List-breaking operations

In a previous chapter we looked at the list-breaking functions `take` and `drop`. The Prelude also includes several higher-order list-breaking functions that take two arguments, a predicate that determines where the list is to be broken and the list to be broken.

Here we look at Prelude functions `takeWhile` and `dropWhile`. As you would expect, function `takeWhile` "takes" elements from the beginning of the list "while" the elements satisfy the predicate and `dropWhile` "drops" elements from the beginning of the list "while" the elements satisfy the predicate. The Prelude definitions are similar to the following:

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

dropWhile' :: (a -> Bool) -> [a] -> [a] -- dropWhile in Prelude
dropWhile'  p []  = []
dropWhile'  p xs@(x:xs')
    | p x         = dropWhile' p xs'
    | otherwise   = xs
```

Note the use of the pattern `xs@(x:xs')` in `dropWhile'`. This pattern matches a non-nil list with `x` and `xs'` binding to the head and tail, respectively, as usual. Variable `xs` binds to the entire list.

As an example, suppose we want to remove the leading blanks from a string. We can do that with the expression:

```
dropWhile ((==) ' ')
```

As with `take` and `drop`, the above functions can also be related by a "law". For all finite lists `xs` and predicates `p` on the same type:

```
takeWhile p xs ++ dropWhile p xs = xs
```

Prelude function `span` combines the functionality of `takeWhile` and `dropWhile` into one function. It takes a predicate `p` and a list `xs` and returns a tuple where the first element is the longest prefix (possibly empty) of `xs` that satisfies `p` and the second element is the remainder of the list.

```
span' :: (a -> Bool) -> [a] -> ([a],[a]) -- span in Prelude
span' _ xs@[]      =  (xs, xs)
span' p xs@(x:xs')
    | p x          =  let (ys,zs) = span' p xs' in (x:ys,zs)
    | otherwise    =  ([],xs)
```

Thus the following "law" holds for all finite lists `xs` and predicates `p` on same type:

```
span p xs == (takeWhile p xs, dropWhile p xs)
```

The Prelude also includes the function `break`, defined as follows:

```
break' :: (a -> Bool) -> [a] -> ([a],[a]) -- break in Prelude
break' p =  span (not . p)
```

5.4.2 List-combining operations

In a previous chapter, we also looked at the function `zip`, which takes two lists and returns a list of pairs of the corresponding elements. Function `zip` applies an operation, in this case *tuple-construction*, to the corresponding elements of two lists.

We can generalize this pattern of computation with the function `zipWith` in which the operation is an argument to the function.

```
zipWith' :: (a->b->c) -> [a]->[b]->[c] -- zipWith in Prelude
zipWith' z  (x:xs)  (y:ys) = z x y : zipWith' z xs ys
zipWith' _  _     _        = []
```

Using a lambda expression to state the tuple-forming operation, the Prelude defines `zip` in terms of `zipWith`:

```
zip'' :: [a] -> [b] -> [(a,b)]  -- zip
zip'' = zipWith' (\x y -> (x,y))
```

Or can be written more simply as:

```
zip''' :: [a] -> [b] -> [(a,b)]  -- zip
zip''' = zipWith' (,)
```

The `zipWith` function also enables us to define operations such as the scalar product of two vectors in a concise way.

```
sp :: Num a => [a] -> [a] -> a
sp xs ys = sum' (zipWith' (*) xs ys)
```

The Prelude includes versions of `zipWith` that take up to seven input lists: `zipWith3` ··· `zipWith7`.

5.5 Rational Arithmetic Revisited

Remember the rational number arithmetic package developed in an earlier chapter. In that package's `Rational` module, we defined a function `eqRat` to compare two rational numbers for equality using the appropriate set of integer comparisons.

```
eqRat :: Rat -> Rat -> Bool
eqRat x y = (numer x) * (denom y) == (numer y) * (denom x)
```

We could have implemented the other comparison operations similarly.

Because the comparison operations are similar, they are good candidates for us to use a higher-order function. We can abstract out the common pattern of comparisons into a function that takes the corresponding integer comparison as an argument.

To compare two rational numbers, we can express their values in terms of a common denominator (e.g., `denom x * denom y`) and then compare the numerators using the integer comparisons. We can thus abstract the comparison into a higher-order function `compareRat` that takes an appropriate integer relational operator and the two rational numbers.

```
compareRat :: (Int -> Int -> Bool) -> Rat -> Rat -> Bool
compareRat r x y = r (numer x * denom y) (denom x * numer y)
```

Then we can define the rational number comparisons in terms of `compareRat`. (Note that we redefine function `eqRat` from the package in the earlier chapter.)

```
eqRat,neqRat,ltRat,leqRat,gtRat,geqRat :: Rat -> Rat -> Bool
eqRat   = compareRat (==)
neqRat  = compareRat (/=)
ltRat   = compareRat (<)
leqRat  = compareRat (<=)
gtRat   = compareRat (>)
geqRat  = compareRat (>=)
```

5.6 Merge Sort

We defined the insertion sort in a previous chapter. It has an average-case time complexity of $O(n^2)$ where n is the length of the input list.

We now consider a more efficient function to sort the elements of a list into ascending order: *merge sort*. Merge sort works as follows:

- If the list has fewer than two elements, then it is already sorted.
- If the list has two or more elements, then we split it into two sublists, each with about half the elements, and sort each recursively.
- We merge the two ascending sublists into an ascending list.

We define function `msort` to be a polymorphic, higher-order function that has two parameters. The first (`less`) is the comparison operator and the second (`xs`) is the list to be sorted. Function `less` must be defined for every element that appears in the list to be sorted.

```
msort :: Ord a => (a -> a -> Bool) -> [a] -> [a]
msort _    [] = []
msort less xs = merge (msort less ls) (msort less rs)
    where  n       = (length xs) `div` 2
           (ls,rs) = splitAt n xs
           merge [] ys     = ys
           merge xs []     = x
           merge ls@(x:xs) rs@(y:ys)
               | less x y  = x : merge xs rs
               | otherwise = y : merge ls ys
```

By nesting the definition of `merge`, we enabled it to directly access the the parameters of `msort`. In particular, we did not need to pass the comparison function to `merge`.

Assuming that `less` evaluates in constant time, the time complexity of `msort` is $O(n \log(n))$, where n is the length of the input list.

- Each call level requires splitting of the list in half and merging of the two sorted lists. This takes time proportional to the length of the list argument.
- Each call of `msort` for lists longer than one results in two recursive calls of `msort`.
- But each successive call of `msort` halves the number of elements in its input, so there are $O(\log(n))$ recursive calls.

So the total cost is $O(n \log(n))$. The cost is independent of distribution of elements in the original list.

We can apply `msort` as follows:

```
msort (<) [5, 7, 1, 3]
```

Function `msort` is defined in curried form with the comparison function first. This enables us to conveniently specialize `msort` with a specific comparison function. For example,

```
descendSort :: Ord a => [a] -> [a]
descendSort = msort (\ x y -> x > y)
```

5.7 Exercises

1. Suppose you need a Haskell function `times` that takes a list of integers (type `Integer`) and returns the product of the elements (e.g, `times [2,3,4]` returns `24`). Define the following Haskell functions.

   a. Function `times1` that uses the Prelude function `foldr`(or `foldr'` from this chapter).

   b. Function `times2` that uses backward recursion to compute the product. (Use recursion directly. Do not use the list-folding Prelude functions such as `foldr` or `product`.)

   c. Function `times3` that uses forward recursion to compute the product. (Hint: use a tail-recursive auxiliary function with an accumulating parameter.)

   d. Function `times4` that uses function `foldl'` from the Haskell library `Data.List`.

2. For each of the following specifications, define a Haskell function that has the given arguments and result. Use the higher order library functions (from this chapter) such as `map`, `filter`, `foldr`, and `foldl` as appropriate.

   a. Function `numof` takes a value and a list and returns the number of occurrences of the value in the list.

   b. Function `ellen` takes a list of character strings and returns a list of the lengths of the corresponding strings.

   c. Function `ssp` takes a list of integers and returns the sum of the squares of the positive elements of the list.

3. Suppose you need a Haskell function `sumSqNeg` that takes a list of integers (type `Integer`) and returns the sum of the squares of the negative values in the list.

   Define the following Haskell functions. Use the higher order library functions (from this chapter) such as `map`, `filter`, `foldr`, and `foldl` as appropriate.

a. Function `sumSqNeg1` that is backward recursive. (Use recursion directly. Do not use the list-folding Prelude functions such as `foldr` or `sum`.)

b. Function `sumSqNeg2` that is tail recursive. (Use recursion directly. Do not use the list-folding Prelude functions such as `foldr` or `sum`.)

c. Function `sumSqNeg3` that uses standard prelude functions such as `map`, `filter`, `foldr`, and `foldl`.

d. Function `sumSqNeg4` that uses list comprehensions (Chapter 7).

4. Define a Haskell function

```
total :: (Integer -> Integer) -> Integer -> Integer
```

so that `total f n` gives `f 0 + f 1 + f 2 + ... + f n`.

5. Define a Haskell function

```
removeFirst :: (a -> Bool) -> [a] -> [a}
```

so that `removeFirst p xs` removes the first element of `xs` that does has the property `p`.

6. Define a Haskell function

```
removeLast :: (a -> Bool) -> [a] -> [a]
```

so that `removeLast p xs` removes the last occurrence of element of `xs` that has the property `p`.

How could you define it using `removeFirst`?

7. Define a Haskell function `map2` that takes a list of functions and a list of values and returns the list of results of applying each function in the first list to the corresponding value in the second list.

8. Define a Haskell function `fmap` that takes a value and a list of functions and returns the list of results from applying each function to the argument value. (For example, `fmap 3 [((*) 2), ((+) 2)]` yields `[6,5]`.)

9. Define a Haskell function `composeList` that takes a list of functions and composes them into a single function. (Be sure to give the type signature.)

10. A list `s` is a *prefix* of a list `t` if there is some list `u` (perhaps nil) such that `s ++ u == t`. For example, the prefixes of string `abc` are ‘,a,ab,abc‘.

    A list `s` is a *suffix* of a list `t` if there is some list `u` (perhaps nil) such that `u ++ s == t`. For example, the suffixes of `abc` are `abc`, `bc`, `c`, and “.

    A list `s` is a *segment* of a list `t` if there are some (perhaps nil) lists `u` and `v` such that . For example, the segments of string `abc` consist of the prefixes and the suffixes plus `b`.

Define the following Haskell functions. You may use functions appearing early in the list to implement later ones.

a. Define a function `prefix` such that `prefix xs ys` returns `True` if `xs` is a prefix of `ys` and returns `False` otherwise.

b. Define a function `suffixes` such that `suffixes xs` returns the list of all suffixes of list `xs`. (Hint: Generate them in the order given in the example of `abc` above.)

c. Define a function `indexes` such that `indexes xs ys` returns a list of all the positions at which list `xs` appears in list `ys`. Consider the first character of `ys` as being at position 0. For example, `indexes ab abaabbab` returns `[1,4,7]`. (Hint: Remember functions like `map`, `filter`, `zip`, and the functions you just defined?)

d. Define a function `sublist` such that `sublist xs ys` returns `True` if list `xs` appears as a segment of list `ys` and returns `False` otherwise.

11. Assume that the following Haskell type synonyms have been defined:

```
type Word = String -- word, characters left-to-right
type Line = [Word] -- line, words left-to-right
type Page = [Line] -- page, lines top-to-bottom
type Doc  = [Page] -- document, pages front-to-back
```

Further assume that values of type `Word` do not contain any space characters. Implement the following Haskell text-handling functions:

a. `npages` that takes a `Doc` and returns the number of `Page`s in the document.

b. `nlines` that takes a `Doc` and returns the number of `Line`s in the document.

c. `nwords` that takes a `Doc` and returns the number of `Word`s in the document.

d. `nchars` that takes a `Doc` and returns the number of `Char`s in the document (not including spaces of course).

e. `deblank` that takes a `Doc` and returns the `Doc` with all blank lines removed. A blank line is a line that contains no words.

f. `linetext` that takes a `Line` and returns the line as a `String` with the words appended together in left-to-right order separated by space characters and with a newline character '`\n`' appended to the right end of the line. (For example, `linetext [Robert, Khayat]` yields `"Robert Khayat\n"`.)

g. `pagetext` that takes a `Page` and returns the page as a `String`–applies `linetext` to its component lines and appends the result in a top-to-bottom order.

h. `doctext` that takes a `Doc` and returns the document as a `String`– applies `pagetext` to its component lines and appends the result in a top-to-bottom order.

i. `wordeq` that takes a two `Docs` and returns `True` if the two documents are *word equivalent* and `False` otherwise. Two documents are word equivalent if they contain exactly the same words in exactly the same order regardless of page and line structure. For example, `[[["Robert"],["Khayat"]]]` is word equivalent to `[[["Robert","Khayat"]]]`.

5.8 References

[**Abelson-Sussman 1996**] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs* (SICP), Second Edition, MIT Press, 1996.

[**Bird 2015**] Richard Bird. *Thinking Functionally with Haskell* Cambridge University Press, 2015.

[**Bird-Wadler 1998**] Richard Bird and Philip Wadler. *Introduction to Functional Programming*, Second Edition, Addison Wesley, 1998. [First Edition, 1988]

[**Chiusano-Bjarnason 2015**] Paul Chiusano and Runar Bjarnason, *Functional Programming in Scala*, Manning, 2015.

[**Cunningham 2014**] H. Conrad Cunningham. *Notes on Functional Programming with Haskell*, 1994-2014.

[**Thompson 2011**] Simon Thompon. *Haskell: The Craft of Programming, Third Edition*, Pearson, 2011.

[**Wikipedia 2017**] *Wikipedia* articles on "Polymorphism", "Ad Hoc Polymorphism", "Parametric Polymorphism", "Subtyping", and "Function Overloading", 2017.

5.9 Terms and Concepts

Procedural abstraction, functions (first-class, higher-order), modularity, interface, function generalization and specialization, scope-commonality-variability (SCV) analysis, hot and frozen spots, think like a functional programmer, common functional programming patterns (map, filter, fold, concatMap), duality theorem, strict and nonstrict functions, bottom, strictness analysis, currying, partial application, operator sections, combinators, functional composition, property of extensionality, pointful and point-free styles, plumbing, function pipeline, lambda expression, application operator `$`, eager evaluation operators `seq` and `$!`, head-normal form, merge sort