Exploring Languages with Interpreters and Functional Programming Chapter 17

H. Conrad Cunningham

04 April 2022

Contents

Professor of Computer and Information Science
University of Mississippi
214 Weir Hall

P.O. Box 1848
University, MS 38677
(662) 915-7396 (dept. office)

Browser Advisory: The HTML version of this textbook requires a browser that supports the display of MathML. A good choice as of April 2022 is a recent version of Firefox from Mozilla.

17 Higher Order Function Examples

17.1 Chapter Introduction

Chapters 15 and 16 introduced the concepts of first-class and higher-order functions and their implications for Haskell programming.

The goals of this chapter (17) are to:

- continue to explore first-class and higher-order functions by examining additional library functions and examples
- examine how to express general problem-solving strategies as higher-order functions, in particular the divide-and-conquer strategy

17.2 List-Breaking Operations

In Chapter 13, 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)
```

17.3 List-Combining operations

In Chapter 14, 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 it 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 `zipWith3` for triples. Library `Data.List` has versions of `zipWith` that take up to seven input lists: `zipWith3` ··· `zipWith7`.

17.4 Rational Arithmetic Revisited

Remember the rational number arithmetic package developed in Chapter 7. 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 Chapter 7.)

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

The Haskell module for the revised rational arithmetic module is in `RationalHO.hs`. The module `TestRationalHO.hs` is an extended version of the standard test script from Chapter 12 that tests the standard features of the rational arithmetic module plus `eqRat`, `neqRat`, and `ltRat`. (It does not currently test `leqRat`, `gtRat`, or `geqRat`.)

17.5 Mergesort

We defined the insertion sort in Chapter 14. 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: *mergesort*. Mergesort 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 _    [x]        = [x]
msort less xs = merge less (msort less ls) (msort less rs)
    where n       = (length xs) `div` 2
          (ls,rs) = splitAt n xs
          merge _ [] ys     = ys
          merge _ xs []     = xs
          merge less ls@(x:xs) rs@(y:ys)
              | less x y  = x : (merge less xs rs)
              | otherwise = y : (merge less 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 * log2 n`), where `n` is the length of the input list and `log2` is a function that computes the logarithm with base 2.

- 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(`log2 n`) recursive calls.

So the total cost is O(`n * log2 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)     -- or (>)
```

17.6 Divide-and-Conquer Algorithms

The mergesort (`msort`) function in Section 17.5 uses the divide-and-conquer strategy to solve the sorting problem. In this section, we exmine that strategy in more detail.

17.6.1 General strategy

For some problem `P`, the general strategy for *divide-and-conquer algorithms* has the following steps:

1. *Decompose* the problem `P` into subproblems, each like `P` but with a smaller input argument.
2. *Solve* each subproblem, either directly or by recursively applying the strategy.
3. *Assemble* the solution to `P` by combining the solutions to its subproblems.

The advantages of divide-and-conquer algorithms are that they:

- can lead to efficient solutions.
- allow use of a "horizontal" parallelism. Similar problems can be solved simultaneously.

We examined the meregesort algorithm in Section 17.5. Other well-known divide-and-conquer algorithms include quicksort, binary search, and multiplication [3:6.4]. In these algorithms, the divide-and-conquer strategy leads to more efficient algorithms.

For example, consider searching for a value in a list. A simple *sequential search* has a time complexity of O(`n`), where n denotes the length of the list. Application of the divide-and-conquer strategy leads to *binary search*, a more efficient O(`log2 n`) algorithm.

17.6.2 As higher-order function

As a general pattern of computation, the divide and conquer strategy can be expressed as the following higher-order function:

```
divideAndConquer :: (a -> Bool)          -- trivial
                 -> (a -> b)             -- simplySolve
                 -> (a -> [a])           -- decompose
```

```
                        -> (a -> [b] -> b) -- combineSolutions
                        -> a               -- problem
                        -> b

divideAndConquer trivial simplySolve decompose
                         combineSolutions problem
    = solve problem
      where solve p
          | trivial p = simplySolve p
          | otherwise = combineSolutions p
                            (map solve (decompose p))
```

If the problem is trivially simple (i.e., `trivial p` holds), then it can be solved directly using `simplySolve`.

If the problem is not trivially simple, then it is decomposed using the `decompose` function. Each subproblem is then solved separately using `map solve`. The function `combineSolutions` then assembles the subproblem solutions into a solution for the overall problem.

Sometimes `combineSolutions` may require the original problem description to put the solutions back together properly. Hence, the parameter `p` in the function definition.

Note that the solution of each subproblem is completely independent from the solution of all the others.

If all the subproblem solutions are needed by `combineSolutions`, then the language implementation could potentially solve the subproblems simultaneously. The implementation could take advantage of the availability of multiple processors and actually evaluate the expressions in parallel. This is "horizontal" parallelism as described above.

If `combineSolutions` does not require all the subproblem solutions, then the subproblems cannot be safely solved in parallel. If they were, the result of `combineSolutions` might be *nondeterministic*, that is, the result could be dependent upon the relative order in which the subproblem results are completed.

Now let's use the function `divideAndConquer` to define a few functions.

17.6.3 Generating Fibonacci sequence

First, let's define a Fibonacci function. Consider the following definition (adapted from Kelly [9:77–78]). This function is inefficient, so it is given here primarily to illustrate the technique.

```
fib :: Int -> Int
fib n = divideAndConquer trivial simplySolve decompose
                         combineSolutions problem
        where trivial 0                = True
```

```
trivial 1                = True
trivial (m+2)            = False
simplySolve 0            = 0
simplySolve 1            = 1
decompose m              = [m-1,m-2]
combineSolutions _ [x,y] = x + y
```

17.6.4 Folding a list

Next, let's consider a folding function (similar to `foldr` and `foldl`) that uses the function `divideAndConquer`. Consider the following definition (also adapted from Kelly [9:79–80]).

```
fold :: (a -> a -> a) -> a -> [a] -> a
fold op i =
    divideAndConquer trivial simplySolve decompose
                     combineSolutions
    where trivial xs               = length xs <= 1
          simplySolve []           = i
          simplySolve [x]          = x
          decompose xs             = [take m xs, drop m xs]
                                     where m = length xs / 2
          combineSolutions _ [x,y] = op x y
```

This function divides the input list into two almost equal parts, folds each part separately, and then applies the operation to the two partial results to get the combined result.

The `fold` function depends upon the operation `op` being *associative*. That is, the result must not be affected by the order in which the operation is applied to adjacent elements of the input list.

In `foldr` and `foldl`, the operations are not required to be associative. Thus the result might depend upon the right-to-left operation order in `foldr` or left-to-right order in `foldl`.

Function `fold` is thus a bit less general. However, since the operation is associative and `combineSolutions` is strict in all elements of its second argument, the operations on pairs of elements from the list can be safely done in parallel,

Another divide-and-conquer definition of a folding function is the function `fold'` shown below. It is an optimized version of `fold` above.

```
fold' :: (a -> a -> a) -> a -> [a] -> a
fold' op i xs = foldt (length xs) xs
                where foldt _ []  =   i
                      foldt _ [x] =   x
                      foldt n ys  = op (foldt m (take m ys))
                                       (foldt m' (drop m ys))
```

```
                                    where  m  = n / 2
                                           m' = n - m
```

17.6.5 Finding minimum and maximum of a list

Now, consider the problem of finding both the minimum and the maximum values in a nonempty list and returning them as a pair.

First let's look at a definition that uses the left-folding operator.

```
sMinMax :: Ord a => [a] -> (a,a)
sMinMax (x:xs) = foldl' newmm (x,x) xs
                 where newmm (y,z) u = (min y u, max z u)
```

Let's assume that the comparisons of the elements are expensive and base our time measure on the number of comparisons. Let n denote the length of the list argument and `time` be a time function

A singleton list requires no comparisons. Each additional element adds two comparisons (one `min` and one `max`).

```
time n | n == 1 = 0
       | n >= 2 = time (n-1) + 2
```

Thus `time n == 2 * n - 2`.

Now let's look at a divide-and-conquer solution.

```
minMax :: Ord a => [a] -> (a,a)
minMax [x]   = (x,x)
minMax [x,y] = if x < y then (x,y) else (y,x)
minMax xs    = (min a c, max b d)
               where m     = length xs / 2
                     (a,b) = minMax (take m xs)
                     (c,d) = minMax (drop m xs)
```

Again let's count the number of comparisons for a list of length `n`.

```
time n | n == 1 = 0
       | n == 2 = 1
       | n > 2  = time (floor (n/2)) + time (ceiling (n/2)) + 2
```

For convenience suppose `n = 2^k` for some `k >= 1`.

```
time n = 2 * time (n/2) + 2
       = 2 * (2 * time (n/4) + 2) + 2
       = 4 * time (n/4) + 4 + 2
       = ...
       = 2^(k-1) * time 2 + sum [ 2^i | i <- [1..(k-1)] ]
       = 2^(k-1) + 2 * sum [ 2^i | i <- [1..(k-1)] ]
                 -     sum [ 2^i | i <- [1..(k-1)] ]
       = 2^(k-1) + 2^k - 2
```

```
= 3 * 2^(k-1) - 2
= 3 * (n/2) - 2
```

Thus the divide and conquer version takes 25 percent fewer comparisons than the left-folding version.

So, if element comparisons are the expensive in relation to to the `take`, `drop`, and `length` list operations, then the divide-and-conquer version is better. However, if that is not the case, then the left-folding version is probably better.

Of course, we can also express `minMax` in terms of the function `divideAndConquer`.

```
minMax' :: Ord a => [a] -> (a,a)
minMax' = divideAndConquer trivial simplySolve decompose
                           combineSolutions
          where n                 = length xs
                m                 = n/2
                trivial xs        = n <= 2
                simplySolve [x]   = (x,x)
                simplySolve [x,y] =
                    if x < y then (x,y) else (y,x)
                decompose xs =
                    [take m xs, drop m xs]
                combineSolutions _ [(a,b),(c,d)] =
                    (min a c, max b d)
```

17.7 What Next?

Chapters 15, 16, and 17 (this chapter) examined higher-order list programming concepts and features.

Chapter 18 examines list comprehensions, an alternative syntax for higher-order list processing that is likely comfortable for programmers coming from an imperative programming background.

17.8 Chapter Source Code

The Haskell module for list-breaking, list-combining, and mergesort functions is in file `HigherOrderExamples.hs`.

The Haskell module for the revised rational arithmetic module is in `RationalHO.hs`. The module `TestRationalHO.hs` is an extended version of the standard test script from Chapter 12.

TODO: Reconstruct source code for divide-and-conquer functions and place links here and in text above. May also want to break out mergesort into a separate module.

17.9 Exercises

1. Define a Haskell function

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

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

2. Define a Haskell function

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

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

   How could you define it using `removeFirst`?

3. 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"`, and `"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 `u ++ s ++ v = t`. 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.

4. 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 `Doc`s 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"]]`.

17.10 Wally World Marketplace POP Project

17.10.1 Problem description and initial design

Wally World Marketplace (WWM) is a "big box" store selling groceries, dry goods, hardware, electronics, etc. In this project, we develop part of a point-of-purchase (POP) system for WWM.

The barcode scanner at a WWM POP—i.e., checkout counter—generates a list of barcodes for the items in a customer's shopping cart. For example, a cart

with nine items might result in the list:

```
[ 1848, 1620, 1492, 1620, 1773, 2525, 9595, 1945, 1066 ]
```

Note that there are two instances of the item with barcode 1620.

The primary goal of this project is to develop a Haskell module WWMPOP (in file WWMPOP.hs) that takes a list of barcodes corresponding to the items in a shopping cart and generates the corresponding printable receipt. The module consists of several functions that work together. We build these incrementally in a somewhat bottom-up manner.

Let's consider how to model the various kinds of "objects" in our application. The basic objects include:

- barcodes for products, which we represent as integers
- prices of products, which we represent as integers denoting cents
- names of products, which we represent as strings

We introduce the following Haskell type aliases for these basic objects above:

```
type BarCode = Int
type Price   = Int
type Name    = String
```

We associate barcodes with the product names and prices using a "database" represented as a list of tuples. We represent this price list database using the following type alias:

```
type PriceList = [(BarCode,Name,Price)]
```

An example price list database is:

```
database :: PriceList
database = [ (1848, "Vanilla yogurt cups (4)",    188),
             (1620, "Ground turkey (1 lb)",       316),
             (1492, "Corn flakes cereal",         299),
             (1773, "Black tea bags (100)",       307),
             (2525, "Athletic socks (6)",         825),
             (9595, "Claw hammer",                788),
             (1945, "32-in TV",                 13949),
             (1066, "Zero sugar cola (12)",       334),
             (2018, "Haskell programming book",  4495)
           ]
```

To generate a receipt, we need to take a list of barcodes from a shopping cart and generate a list of prices associated with the items in the cart. From this list, we can generate the receipt.

We introduce the type aliases:

```
type CartItems  = [BarCode]
type CartPrices = [(Name,Price)]
```

We thus identify the need for a Haskell function

```
priceCart :: PriceList -> CartItems -> CartPrices
```

that takes a database of product prices (i.e., a price list) and a list of barcodes of the items in a shopping cart and generates the list of item prices.

Of course, we must determine the relevant sales taxes due on the items and determine the total amount owed. We introduce the following type alias for the bill:

```
type Bill = (CartPrices, Price, Price, Price)
```

The three `Price` items above are for Subtotal, Tax, and Total amounts associated with the purchase (printed on the bottom of the receipt).

We thus identify the need for a Haskell function

```
makeBill :: CartPrices -> Bill
```

that takes the list of item prices and constructs a `Bill` tuple. In carrying out this calculation, the function uses the following n constant:

```
taxRate :: Double
taxRate = 0.07
```

Given a bill, we must be able to convert it to a printable receipt. Thus we introduce the Haskell function

```
formatBill ::  Bill -> String
```

that takes a bill tuple and generates the receipt. It uses the following named constant for the width of the line:

```
lineWidth :: Int
lineWidth = 34
```

Given the above functions, we can put the above functionality together with the Haskell function:

```
makeReceipt :: PriceList -> CartItems -> String
```

that does the end-to-end conversion of a list of barcodes to a printed receipt given an applicable price database, tax rate, and line width.

Given the example shopping cart items and price list database, we get the following receipt when printed.

```
      Wally World Marketplace

Vanilla yogurt cups (4).......1.88
Ground turkey (1 lb)..........3.16
```

```
Toasted oat cereal.....2.99
Ground turkey (1 lb)......3.16
Black tea bags (100)......3.07
Athletic socks (6).....8.25
Claw hammer...............7.88
32-in. television.....139.49
Zero sugar cola (12)......3.34

Subtotal................176.26
Tax.....................12.34
Total...................188.60
```

The above Haskell definitions are collected into the source file `WWMPOP_skeleton.hs`.

The exercises in Section 17.10.3 guide you to develop the above functions incrementally.

17.10.2 Prelude functions useful for project

In the exercises in Section 17.10.3, you may want to consider using some of the following:

- numeric functions from the Prelude library such as such as:
  - `div`, integer division truncated toward negative infinity, and `quot`, integer division truncated toward 0
  - `rem` and `mod` satisfy the following for `y /= 0`

    ```
    (x `quot` y)*y + (x `rem` y) == x
    (x `div`  y)*y + (x `mod` y) == x
    ```

  - `floor`, `ceiling`, `truncate`, and `round` that convert real numbers to integers; `truncate` truncates toward 0 and `round` rounds away from 0
  - `fromIntegral` converts integers to `Double` (and from `Integer` to `Int`)
  - `show` converts numbers to strings
- first-order list functions (Chapters 13 and 14) from the Prelude–such as `head`, `tail`, `++`, `-take`, `drop`, `length`, `-sum`, and `product`
- Prelude function `replicate :: Int -> a -> [a]` such that `replicate n e` returns a list of `n` copies of `e`
- higher-order list functions (Chapters 15, 16, and 17) from the Prelude such as `map`, `filter`, `foldr`, `foldl`, and `concatMap`
- list comprehensions (Chapter 18)—not necessary for solution but may be convenient

17.10.3 POP project exercises

Note: Most of the exercises in this project can be programmed without direct recursions. Consider the Prelude functions listed in the previous subsection.

Also remember that the character code `'\n'` is the newline character; it denotes the end of a line in Haskell strings.

This project defines several type aliases and the constants `lineWidth` and `taxRate` that should be defined and used in the exercises. You should start with the template source file `WWMPOP_skeleton.hs` to develop your own `WWMPOP.hs` solution.

1. Develop the Haskell function

   ```
   formatDollars :: Price -> String
   ```

   that takes a `Price` in cents and formats a string in dollars and cents. For example, `formatDollars 1307` returns the string `13.07`. (Note the `0` in `07`.)

2. Using `formatDollars` above, develop the Haskell function

   ```
   formatLine :: (Name, Price) -> String
   ```

   that takes an item and formats a line of the receipt for that item. For example,

   ```
   formatLine ("Claw hammer",788)
   ```

   yields the string:

   ```
   "Claw hammer..................7.88\n"
   ```

   This string has length `lineWidth` not including the newline character. The space between the item's name and cost is filled using `'.'` characters.

3. Using the `formatLine` function above, develop the Haskell function

   ```
   formatLines :: CartPrices -> String
   ```

   that takes a list of priced items and formats a string with a line for each item. (In general, the resulting string will consist of several lines, each ending with a newline.)

4. Develop the Haskell function

   ```
   calcSubtotal :: CartPrices -> Price
   ```

   that takes a list of priced items and calculates the sum of the prices (i.e., the subtotal).

5. Develop the Haskell function

   ```
   formatAmt :: String -> Price -> String
   ```

that takes a label string and a price amount and generates a line of the receipt for that label

For example,

```
formatAmt "Total" 18860
```

generates the string:‘

```
"Total.....188.60\n"`.
```

6. Develop the Haskell function

   ```
   formatBill :: Bill -> String
   ```

   that takes a `Bill` tuple and generates a receipt string.

7. Develop the Haskell function

   ```
   look :: PriceList -> BarCode -> (Name,Price)
   ```

   that takes a price list database and a barcode for an item and looks up the name and price of the item.

   If the `BarCode` argument does not occur in the `PriceList`, then `look` should return the tuple `("None",0)`.

8. Now develop the Haskell function

   ```
   priceCart :: PriceList -> CartItems -> CartPrices
   ```

   defined above.

9. Now develop the Haskell function

   ```
   makeBill :: CartPrices -> Bill
   ```

   defined above. It takes a list of priced items and generates a bill tuple. It uses the `taxRate` constant.

10. Now develop the Haskell function

    ```
    makeReceipt :: PriceList -> CartItems -> String
    ```

    defined above. This function defines the end-to-end processing that takes the list of items from the shopping cart and generates the receipt.

11. Develop Haskell functions

    ```
    addPL    :: PriceList -> BarCode -> (Name,Price)
                  -> PriceList
    removePL :: PriceList -> BarCode -> PriceList
    ```

    Function `removePL` takes an “old” price list and a barcode to remove and returns a “new” price list with any occurrences of that barcode removed.

Function `addPL` takes an "old" price list, a barcode, and a name/price pair to add and returns a price list with the item added. (If the the barcode is already in the list, the old entry should be removed.)

17.11 Acknowledgements

In Summer 2016, I adapted and revised the following to form a chapter on Higher-Order Functions:

- Chapter 6 of my *Notes on Functional Programming with Haskell* [7], which is influenced by Bird [1–3] and Wentworth [11]
- My notes on *Functional Data Structures (Scala)* [8], which are based, in part, on chapter 3 of the book *Functional Programming in Scala* [4] and its associated materials [5,6]

In 2017, I continued to develop this work as Chapter 5, Higher-Order Functions, of my 2017 Haskell-based programming languages textbook.

In Summer 2018, I divided the previous Higher-Order Functions chapter into three chapters in the 2018 version of the textbook, now titled *Exploring Languages with Interpreters and Functional Programming* (ELIFP), Previous sections 5.1-5.2 became the basis for new Chapter 15, Higher-Order Functions, section 5.3 became the basis for new Chapter 16, Haskell Function Concepts, and previous sections 5.4-5.6 became the basis for new Chapter 17 (this chapter), Higher-Order Function Examples.

In Fall 2018, I developed the Wally World Marketplace POP project. It was motivated by a similar project in Thompson's textbook [10] that I had used in my courses. I designed the project and its exercises to allow for the possibility of automatic grading.

In Summer 2018, I also adapted and revised Chapter 14 of my *Notes on Functional Programming with Haskell* [7] to form Chapter 29 (Divide and Conquer Algorithms) of ELIFP. These previous notes drew on the presentations in the 1st edition of the Bird and Wadler textbook [3], Kelly's dissertation [9], and other functional programming sources.

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating additional separately developed materials, reformatting the document (e.g., using CSS), constructing a bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

In 2022, I also merged the previous ELIFP Chapter 29 (Divide and Conquer Algorithms) and the Wally World Marketplace project into an expanded Chapter 17 (this chapter) of the revised ELIFP.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document

to HTML, PDF, and other forms as needed.

17.12 Terms and Concepts

List-breaking (splitting) operators, list-combining operators, rational arithmetic, merge sort, divide and conquer, horizontal parallelism, divide and conquer as higher-order function, sequential search binary search, simply solve, decompose, combine solutions, Fibonacci sequence, nondeterministic, associative.

17.13 References

[1] Richard Bird. 1998. *Introduction to functional programming using Haskell* (Second ed.). Prentice Hall, Englewood Cliffs, New Jersey, USA.

[2] Richard Bird. 2015. *Thinking functionall with Haskell* (First ed.). Cambridge University Press, Cambridge, UK.

[3] Richard Bird and Philip Wadler. 1988. *Introduction to functional programming* (First ed.). Prentice Hall, Englewood Cliffs, New Jersey, USA.

[4] Paul Chiusano and Runar Bjarnason. 2015. *Functional programming in Scala* (First ed.). Manning, Shelter Island, New York, USA.

[5] Paul Chiusano and Runar Bjarnason. 2022. FP in Scala exercises, hints, and answers. Retrieved from https://github.com/fpinscala/fpinscala

[6] Paul Chiusano and Runar Bjarnason. 2022. FP in Scala community guide and chapter notes. Retrieved from https://github.com/fpinscala/fpinscala/wiki

[7] H. Conrad Cunningham. 2014. *Notes on functional programming with Haskell.* University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/csci450/notes/haskell_notes.pdf

[8] H. Conrad Cunningham. 2019. *Functional data structures (Scala).* University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/csci555/notes/FPS03/FunctionalDS.html

[9] Paul H. J. Kelly. 1989. *Functional programming for loosely-coupled multiprocessors.* MIT Press, Cambridge, Massachusetts, USA.

[10] Simon Thompson. 2011. *Haskell: The craft of programming* (Third ed.). Addison-Wesley, Boston, Massachusetts, USA.

[11] E. Peter Wentworth. 1990. *Introduction to functional programming using RUFL.* Rhodes University, Department of Computer Science, Grahamstown, South Africa.