Wally World POP Project

H. Conrad Cunningham

11 October 2018

Professor of Computer and Information Science
University of Mississippi
211 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-5358

Browser Advisory: The HTML version of this document requires use of a browser that supports the display of MathML. A good choice as of October 2018 is a recent version of Firefox from Mozilla.

Wally World POP Project

I wrote this project description for CSci 450/503 Assignment #2 in Fall 2018.

2018-10-01: Changed format slightly, no changes in detail.

2018-10-08: Removed spurious "{.haskell}" from Exercise #5 statement

2018-10-11: Added Note to Exercise 7 (about unknown barcode)

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 following exercises guide you to develop the above functions incrementally.

Useful Prelude Functions

In the exercises in the next subsection, 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 13-14) 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 15-17) from the Prelude such as `map`, `filter`, `foldr`, `foldl`, and `concatMap`
- list comprehensions (Chapter 18)

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.

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.

   (2018-10-11) Note: 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.)