# The supermarket billing example

The example of the supermarket bill:

- A supermarket billing process starts with a database of items.
- An item consists of an integer bar code, a string name and an integer price, in pennies.
- We also receive a list of bar codes that forms the “till”.
- We must look up those bar codes to find the corresponding items, and get the name and price of the items in the till. This will form a “bill”, which will consists of this list of name/price pairs.
- Lastly, we must format that bill, including computation of a total, to produce a string.

Here is a start for our file. We will be using HUnit tests to test the program. We explicitly import the prelude while hiding the particular `lookup` function, as we will implement our own.

```
module Billing where

import Test.HUnit
import Prelude hiding (lookup)

tests = TestList [
    -- will add tests here
    ]
```

We can run the tests in the file via:

```
:load Billing
runTestTT tests
```

Now let’s set up some types, a sample database and a special “unknown item”:

```
type Name     = String
type Price    = Int
type BarCode  = Int
type Database = [(BarCode,Name, Price)]
type Till     = [BarCode]
type BillItem = (Name, Price)
type Bill     = [BillItem]

codeIndex :: Database
codeIndex = [ (4719, "Fish␣Fingers" , 121),
              (5643, "Nappies" , 1010),
              (3814, "Orange␣Jelly", 56),
              (1111, "Hula␣Hoops", 21),
              (1112, "Hula␣Hoops␣(Giant)", 133),
              (1234, "Dry␣Sherry,␣1lt", 540) ]

unknownItem :: BillItem
unknownItem = ("Unknown␣Item", 0)
```

We also fix a value for the length of the bill report lines:

```
lineLength :: Int
lineLength = 30
```

Now we can think of breaking our problem up in functions that perform the various steps. In functional programming languages one important step is to consider functions that produce intermediate results, then compose those functions. In our case we can start with a Till value and use it to produce a Bill value. Then we can separately worry about turning that Bill value into a printable string. So at a high level our program can be decomposed thus:

```
produceBill :: Till -> String
produceBill till = formatBill (makeBill till)
-- Shortcut for this:  produceBill = formatBill . makeBill

makeBill :: Till -> Bill
makeBill till = [] -- Need to fix this

formatBill :: Bill -> String
formatBill bill = "" -- Need to fix this
```

We put some dummy implementations for now.

This **function composition** is one of the tools at our disposal for breaking down a complex problem into steps.

Another common pattern is that of a **list transformation**. Our list comprehension work is great for that. In our case, we can implement makeBill by looking up each code in the till into the database. We can offload the work of that lookup to another functions, which we will call lookup:

```
makeBill till = [lookup code | code <- till]

lookup :: BarCode -> BillItem
lookup code = look codeIndex code

look :: Database -> BarCode -> BillItem
look db code = unknownItem   -- Need to fix this
```

Now the look function has to do some real work. We will use two tests for it, and add them to the tests list from earlier:

```
tests = TestList [
    TestCase $ assertEqual "lookExists"
          ("Orange␣Jelly", 56)
          (look codeIndex 3814),
    TestCase $ assertEqual "lookMissing"
          ("Unknown␣Item", 0)
          (look codeIndex 3815)
    ]
```

Don't worry about the dollar signs just yet.

Equipped with those tests, we can now work through the look function's implementation. We can think of that again as a 2-step process:

- Use a list comprehension to find matches for the bar code in the list of items.
- Interpret the results.

We can use a where clause to hold the resulting comprehension, like so:

```
look db code = if null results then unknownItem else results !! 0
   where results = [(name, price) | (code2, name, price) <- db, code2 == code]
```

Here list !! index returns the entry in the list at the given index (it's loosely equivalent to the array indexing operator in other languages). The list comprehension looks through the list of code triples in search of one that matches the given code.

Now that we have all the tools for generating a bill from a till, we can focus on converting the bill to a string. We can break that up in steps as well:

- Format the items lines
- Compute and format the total
- Combine the two

The formatBill method puts all those together:

```
formatBill :: Bill -> String
formatBill bill = formatLines bill ++ formatTotal total
   where total = makeTotal bill

formatLines :: Bill -> String
formatLines bill = ""        -- Need to fix this

formatTotal :: Price -> String
formatTotal total = ""       -- Need to fix this

makeTotal :: Bill -> Price
makeTotal bill = 0
```

We can add some tests:

```
   TestCase $ assertEqual "makeTotal"
         (23 + 45)
         (makeTotal [("something", 23), ("else", 45)]),
   TestCase $ assertEqual "formatTotal"
         "\nTotal....................6.61"
         (formatTotal 661),
   TestCase $ assertEqual "formatBill"
         ("Dry␣Sherry,␣1lt..........5.40\n" ++
          "Fish␣Fingers.............1.21\n" ++
          "\nTotal....................6.61")
         (formatBill [("Dry␣Sherry,␣1lt", 540), ("Fish␣Fingers", 121)]),
```

Let's start with makeTotal. We can consider it as a composition of two steps:

- Getting all the prices out of the bill items.

- Adding up those prices.

We can combine the two steps as follows:

```
makeTotal :: Bill -> Price
makeTotal bill = sum prices
    where prices = [p | (_, p) <- bill]
```

Next let's format the total. The format we are after for all items is as follows:

- The name of the item on the left side
- The price of the item on the right side
- An appropriate number of dots inbetween to reach a total length of 30

We will write a function that accomplishes this, then return to writing the rest of the total format:

```
formatLine :: BillItem -> String
formatLine (name, price) = name ++ filler ++ formattedPrice
   where formattedPrice = formatPrice price
         space = lineLength - length name - length formattedPrice
         filler = replicate space '.'

formatPrice :: Price -> String
formatPrice price = ""  -- Need to fix this
```

And let's add a test for it, as well as tests for the formatPrice method:

```
   TestCase $ assertEqual "normal␣amount" "12.53" (formatPrice 1253),
   TestCase $ assertEqual "single␣digit␣pennies" "12.03" (formatPrice 1203),
   TestCase $ assertEqual "no␣dollars" "0.53" (formatPrice 53),
   TestCase $ assertEqual "whole␣line"
         "Dry␣Sherry,␣1lt..........5.40"
         (formatLine ("Dry␣Sherry,␣1lt", 540)),
```

Formatting the price consists of printing the dollar part (using show), using a dot, then the pennies using two digits (so 09 instead of just 9).

```
formatPrice :: Price -> String
formatPrice price = show dollars ++ "." ++ space ++ show pennies
   where dollars = price `div` 100
         pennies = price `mod` 100
         space = if pennies < 10 then "0" else ""
```

TODO:

```
formatLines :: Bill -> String
formatLines lines = join "\n" [formatLine l | l <- lines]
    where join sep lines = intercalate sep lines

formatTotal :: Price -> String
formatTotal total = "\n" ++ formatPrice ("Total", total)
```