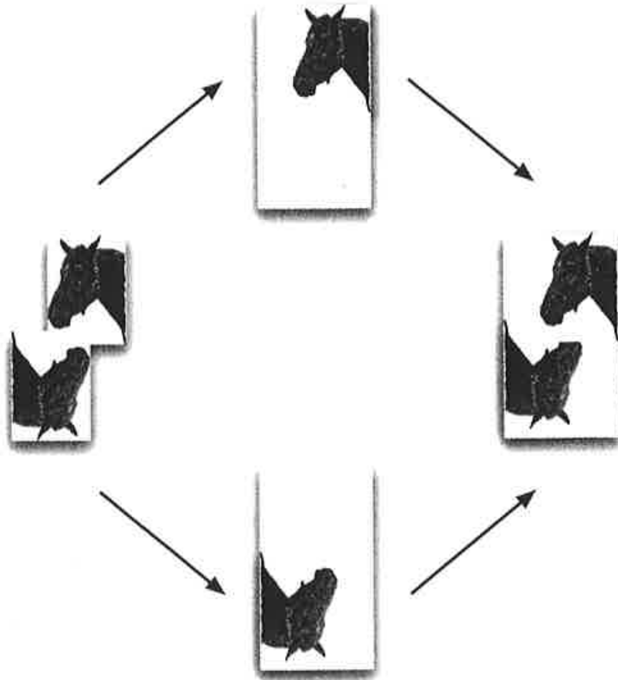


Figure 6.11 Superimposing two `Images`

6.38 Define QuickCheck properties to check the implementation of the functions over the `Image` type. How many carry over from the `Picture` type, and how many have to be re-defined?

6.7 Extended exercise: supermarket billing

This collection of exercises looks at supermarket billing.[3] The idea is to use the list-manipulating techniques presented in Chapter 5. In particular we will be using list comprehensions and also the prelude functions mentioned there. We will also expect local definitions – as explained in Section 4.2 – to be used when appropriate.

The problem

A scanner at a supermarket checkout will produce from a basket of shopping a list of bar codes, like

```
[1234,4719,3814,1112,1113,1234]
```

which has to be converted to a bill as shown in Figure 6.12. We have to decide first how to model the objects involved. Bar codes and prices (in pence) can be modelled by integers; names of goods by strings. We therefore say that

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

[3] I am grateful to Peter Lindsay *et al.* of the Department of Computer Science at the University of New South Wales, Australia, for the inspiration for this example, which was suggested by their lecture notes.

```
                 Haskell Stores

Dry Sherry, 1lt...........5.40
Fish Fingers..............1.21
Orange Jelly..............0.56
Hula Hoops (Giant)........1.33
Unknown Item..............0.00
Dry Sherry, 1lt...........5.40

Total....................13.90
```

Figure 6.12 A supermarket bill

The conversion will be based on a **database** which links bar codes, names and prices. As in the library, we use a list to model the relationship.

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

The example database we use is

```
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)]
```

The object of the script will be to convert a list of bar codes into a list of `(Name,Price)` pairs; this then has to be converted into a string for printing as above. We make the type definitions

```
type TillType = [BarCode]
type BillType = [(Name,Price)]
```

and then we can say that the functions we wish to define are

```
makeBill    :: TillType -> BillType
```

which takes a list of bar codes to a list of name/price pairs,

```
formatBill  :: BillType -> String
```

which takes a list of name/price pairs into a formatted bill, and

```
produceBill :: TillType -> String
```

which will combine the effects of `makeBill` and `formatBill`, thus

```
produceBill = formatBill . makeBill
```

The length of a line in the bill is decided to be 30. This is made a constant, thus

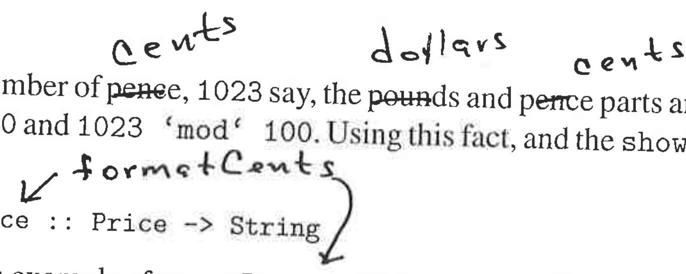
```
lineLength :: Int
lineLength = 30
```

Making `lineLength` a constant in this way means that to change the length of a line in the bill, only one definition needs to be altered; if 30 were used in each of the formatting functions, then each would have to be modified on changing the line length. The rest of the script is developed through the sequences of exercises which follow.

Formatting the bill

First we develop the `formatBill` function from the bottom up: we design functions to format prices, lines, and the total, and using these we finally build the `formatBill` function itself.

Exercises



6.39 Given a number of pence, 1023 say, the pounds and pence parts are given by 1023 `'div' 100` and 1023 `'mod' 100`. Using this fact, and the `show` function, define a function

```
formatPence :: Price -> String
```

so that, for example, `formatPence 1023 = "10.23"`; you need to be careful about cases like `"12.02"`.

6.40 Using the `formatPence` function, define a function

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

which formats a line of a bill, thus

```
formatLine ("Dry Sherry, 1lt",540)
           = "Dry Sherry, 1lt...........5.40\n"
```

Recall that `'\n'` is the newline character, that `++` can be used to join two strings together, and that `length` will give the length of a string. You might also find the `replicate` function useful.

6.41 Using the `formatLine` function, define

```
formatLines :: [ (Name,Price) ] -> String
```

which applies `formatLine` to each `(Name,Price)` pair, and joins the results together.

6.42 Define a function

```
makeTotal :: BillType -> Price
```

which takes a list of `(Name,Price)` pairs, and gives the total of the prices. For instance,

```
makeTotal [(" ... ",540),(" ... ",121)] = 661
```

6.43 Define the function

```
formatTotal :: Price -> String
```

so that, for example,

```
formatTotal 661 = "\nTotal....................6.61"
```

6.44 Using the functions `formatLines`, `makeTotal` and `formatTotal`, define

```
formatBill :: BillType -> String
```

so that on the input

```
[("Dry Sherry, 1lt",540),("Fish Fingers",121),
("Orange Jelly",56),("Hula Hoops (Giant)",133),
("Unknown Item",0),("Dry Sherry, 1lt",540)]
```

the example bill at the start of the section is produced.

Making the bill: bar codes into names and prices

Now we have to look at the database functions which accomplish the conversion of bar codes into names and prices.

Exercises

6.45 Define a function

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

which returns the `(Name,Price)` pair corresponding to the `BarCode` in the `Database`. If the `BarCode` does not appear in the database, then the pair `("Unknown Item", 0)` should be the result.

Hint: using the ideas of the library database you might find that you are returning a *list* of `(Name,Price)` rather than a single value. You can assume that each bar code occurs only once in the database, so you can extract this value by taking the `head` of such a list *if it is non-empty*.

6.46 Define a function

```
lookup :: BarCode -> (Name,Price)
```

may want to name 'lookup'

which uses `look` to look up an item in the particular database `codeIndex`. This function clashes with a function `lookup` defined in the prelude; consult page 53 for details of how to handle this.

6.47 Define the function

```
makeBill   :: TillType -> BillType
```

which applies `lookup` to every item in the input list. For instance, when applied to `[1234,4719,3814,1112,1113,1234]` the result will be the list of `(Name,Price)` pairs given in Exercise 6.25. Note that `1113` does not appear in `codeIndex` and so is converted to `("Unknown Item",0)`.

This completes the definition of `makeBill` and together with `formatBill` gives the conversion program.

Extending the problem

We conclude with some further exercises.

```
                 Haskell Stores

       Dry Sherry, 1lt...........5.40
       Fish Fingers..............1.21
       Orange Jelly..............0.56
       Hula Hoops (Giant)........1.33
       Unknown Item..............0.00
       Dry Sherry, 1lt...........5.40

       Discount..................1.00

       Total....................12.90
```

Figure 6.13 Bills with 'multibuy' discounts

Exercises

6.48 You are asked to add a discount for multiple buys of sherry: for every two bottles bought, there is a 1.00 discount. From the example list of bar codes

```
[1234,4719,3814,1112,1113,1234]
```

the bill should be as illustrated in Figure 6.13. You will probably find it helpful to define functions

```
makeDiscount :: BillType -> Price
formatDiscount :: Price -> String
```

which you can use in a redefined

```
formatBill :: BillType -> String
```

6.49 Design functions which update the database of bar codes. You will need a function to add a `BarCode` and a `(Name,Price)` pair to the `Database`, while at the same time removing any other reference to the bar code already present in the database.

6.50 Re-design your system so that bar codes which do not appear in the database give no entry in the final bill. There are (at least) two ways of doing this.

- Keep the function `makeBill` as it is, and modify the formatting functions, or
- modify the `makeBill` function to remove the 'unknown item' pairs.

6.51 [Harder] How appropriate would it be to test your supermarket billing system using QuickCheck? Could you check parts of the system using QuickCheck? Could you use it to test the whole system, or could you do both?

6.52 [Project] Design a script of functions to analyse collections of sales. Given a list of `TillType`, produce a table showing the total sales of each item. You might also analyse the bills to see which *pairs* of items are bought together; this could assist with placing items in the supermarket.