Week 2: Lists and Tuples

Hans Georg Schaathun

25th April 2017

Time	Topic	Reading
8.15-	Recap of tutorials last week	
	Lecture: Lists and tuples	Thompson Chapter 5
9.00-	Tutorial 6: Lists and Tuples	
11.45-	Lunch break	
12.15-	Recap/discussion of Tutorial 6	
	Lecture: List processing	Thompson Chapter 6
13.00-	Tutorial 7–8: More on Lists and Tuples	

This PDF document is available in an HTML version at `http://www.hg.schaathun.net/FPIA/week02.html`.

1 Tutorial 6: Lists and Tuples

Reading: Simon Thompson, Chapter 5 (skip 5.3). This exercise follows the example in Chapter 5.7 of Simon Thompson's book.

We are going to design a simple system to record books borrowed from a library. To do this, we will need both tuples and lists. If you want to, you may use algebraic data types, but if you have enough on your plate already, don't think about algebraic data types until next week.

Primarily, we need a data structure to record loans. A loan in this context is the fact that a given person has borrowed a particular book. Thus we also need to be able to represent books and persons.

1.1 Getting started

You need to create a new Haskell module (file) with all the definitions in this tutorial.

It is useful to run `ghci` in a window beside your editor, and reload the file after every amendment and test functions. Only reloading the file will tell you if it is syntactically correct.

1.2 Books and Persons

Books and persons may be represented in different ways in terms of data types. Think through the following:

1. What information should be included for each book?
2. What information should be included for each person?
3. What data types are appropriate for books and persons?

1.3 The Loan Database

1. Define data types for a Person (borrower) and a book. For instance, as follows:

   ```
   type Person = String
   type Book = String
   ```

   These are simple (and crude) definitions. A book is represented only by its title, and a person by his name.

2. Define a data type `Loan` to represent a loan, recording the borrower (a person) and the book borrowed. E.g.

   ```
   type Loan = (Book, Person)
   ```

3. Finally we need a data type `Database` which records the set of outstanding loans. What data types can be used to represent a set in Haskell?

   We can use a list, as follows:

   ```
   type Database = [Loan]
   ```

1.4 Test Objects

Let's define a couple of objects for the sake of testing.

1. Define the following constants of type Loan in your module:

   a) `loan1` representing the fact that `"John"` has borrowed `"Huckleberry Finn"` I.e.

   ```
   loan1 = ("Huckleberry Finn", "John")
   ```

b) `loan2` representing the fact that `"John"` has borrowed `"Tom Sawyer"`

c) `loan3` representing the fact that `"Jane"` has borrowed `"Tom Sawyer"`

2. Define a database `db1` containing each of the three loans `loan1`, `loan2`, and `loan3`. I.e.

```
db1 = [loan1 ,loan2 ,loan3]
```

3. Check your definitions by evaluating them in `ghci`. I.e. loan1 db1

4. Also evaluate `loan2` and `loan3` to check that they are correct.

1.5 Retrieval functions

A database is nothing without functions to access it. Add the declarations and definitions to you module file as you work through the steps below.

1. Define a function to get all the books borrowed by a particular person. We can declare it as follows:

```
books :: Database -> Person -> [Book]
```

We can define the `books` function using list comprehension:

```
books db person = [ b | (b,p) <- db, p == person ]
```

2. Similar to the above, we define a function to return all people who have borrowed a particular book, i.e.

```
borrowers :: Database -> Book -> [Person]
```

Write a definition for the `borrowers` function.

3. Test both functions in `ghci`:

```
books db1 "John"
books db1 "Jane"
borrowers db1 "Huckleberry Finn"
borrowers db1 "Tom Sawyer"
```

4. What output do you expect? What output do you get? Compare the two.

1.6 More retrieval

The retrieval functions above return lists. Let's look at a couple of other useful retrievel functions.

The first function says whether the given book is borrowed or not. The second function gives the number of books borrowed by the given person.

1. We want to know if a particular book has been borrowed. I.e.

   ```
   borrowed :: Database -> Book -> Bool
   ```

   We can implement this using the `borrowers` function, and see if the result is empty or not, e.g.

   ```
   borrowed db book = length (borrowers db book) > 0
   ```

2. We want to know how many books a given individual has borrowed. I.e.

   ```
   numBorrowed :: Database -> Person -> Int
   ```

   Write a definition using the `books` and `length` functions.

3. Test both functions in `ghci`:

   ```
   numBorrowed db1 "John"
   numBorrowed db1 "Jane"
   borrowed db1 "Huckleberry Finn"
   borrowed db1 "Tom Sawyer"
   ```

   What output do you expect? What output do you get? Compare the two.

1.7 Modifiers

We are able to retrieve data, but we also want to update the database. Since Haskell does not have variables, we cannot *really* change anything. We need functions which take the old database as input and returns a modified database as output.

1. Add a function to add a loan. I.e.

   ```
   makeLoan :: Database -> Person -> Book -> Database
   makeLoan db person book = newloan : db
             where newloan = (book, person) :: Loan
   ```

2. Add a function to remove a loan. I.e.

   ```
   returnLoan :: Database -> Person -> Book -> Database
   returnLoan db person book = [ loan |
                 loan <-db, loan /= (book, person) ]
   ```

3. Test both functions in `ghci`:

```
makeLoan db1 "John" "Oliver Twist"
returnLoan db1 "John" "Huckleberry Finn"
returnLoan db1 "Jane" "Huckleberry Finn"
```

What output do you expect? What output do you get? Compare the two.

1.8 Discussion

1. What happens if you add duplicate loans? E.g. add the following definitions in your module:

   ```
   db2 = makeLoan db1 "John" "Tom Sawyer"
   db3 = returnLoan db2 "John" "Tom Sawyer"
   ```

   Note that `db2` has John borrowing *Tom Sawyer* added twice, as it was already in `db1`. In `db3` this duplicate loan has been removed.

   How many loans are in `db3`? Decide what you expect before you move on.

2. Start GHCi and evaluate the two new databases:

   ```
   db2
   db3
   ```

   Do the results match your expectation?

3. Discuss the following:

   - Should duplicate loans be allowed?
   - How should duplicate loans be handled? Remember that even if it is not allowed, a good program has to handle it as errors and prevent human error from messing up the database.
   - Do you think your/our implementation is satisfactory on this point?

1.9 Tip: Debugging output

Debugging a pure functional program can often feel like fumbling in the dark. If you have programmed imperatively, you are hopefully used to adding extra output instruction, to see what happens when the program runs.

Why don't we just add such output lines in functional programs?

The answer is of course that I/O is side effects, and we don't like side effects. We are going to learn how to do proper I/O tomorrow, but even so it is often desirable to keep core functions pure and without I/O.

However, for debugging purposes, it is possible to cheat, with the Debyg.Trace module. It provides a couple of functions which pretend to be pure, but which still produces output. This is good for debugging, but should otherwise be avoided. Feel free to look it up.

2 Tutorial 7: Pictures

Reading: Simon Thompson, Chapter 6

1. Do the exercises in Chapter 6.4 of Thompson
2. Do the exercises in Chapter 6.6 of Thompson

3 Tutorial 8: Credit card numbers (optional)

This exercise is based on a an exercise given by Richard Eisenberg for the cis194 module at University of Pennsylvania. He in turn adapted it from a practicum assigned at the University of Utrecht functional programming course taught by Doaitse Swierstra, 2008-2009.

Working with long numbers, such as national insurance numbers, account numbers, ISBN numbers or credit card numbers, it is easy to make mistakes. Therefore such numbers are designed using an error correcting codes. The most probable mistakes lead to an invalid number, so that it is easily seen that an error has occurred.

In this exercise, we shall explore credit card numbers and write a functional program to validate such numbers. The validation comprises the following steps:

1. Double the value of every second digit beginning from the right. That is, the last digit is unchanged; the second-to-last digit is doubled; the third-to-last digit is unchanged; and so on. For example, [1,3,8,6] becomes [2,3,16,6].
2. Calculate the digit sum of each number in the list. For the one-digit numbers, this is just the number itself. For two-digit numbers, it is the sum of the two digits. For example, [2,3,16,6] becomes [2,3,7,6].
3. Add all the digit sums together. For example, [2,3,7,6] becomes 18.
4. Calculate the remainder when the sum is divided by 10. For example, 18 becomes 8.
5. If the result equals 0, then the number is valid. In the running example, we got 8, which means the number is invalid.

We will implement the validation through a series of small exercises.

3.1 Exercise 1

We first need to be able to split a number into its last digit and the rest of the number. Write these functions:

```
lastDigit     :: Integer -> Integer
dropLastDigit :: Integer -> Integer
```

Haskell has built-in functions or operators for integer division and remainder (modulo).

Examples:

$$\texttt{lastDigit}123 = 3 \tag{1}$$
$$\texttt{lastDigit}9 = 9 \tag{2}$$
$$\texttt{dropLastDigit}123 = 12 \tag{3}$$
$$\texttt{dropLastDigit}9 = 0 \tag{4}$$

Task: Implement and test the two functions described above.

3.2 Exercise 2

Now, we male a function to break a number into a list of digits. The method is the standard approach to writing a number to base b (with $b = 10$ in our case). You will need the functions from the previous exercise and apply them with recursion. The function should have the following signature,

```
toDigits      :: Integer -> [Integer]
```

Positive numbers must be converted to a list of digits. For 0 or negative inputs, `toDigits` should return the empty list.

Examples:

$$\texttt{toDigits}1234 = [1, 2, 3, 4] \tag{5}$$
$$\texttt{toDigits}0 = [] \tag{6}$$
$$\texttt{toDigits}(-17) = [] \tag{7}$$

Task: Implement and test the `toDigits` function.

3.3 Exercise 3

Once we have the digits in a list, we need to double every other one. Define a function

```
doubleEveryOther :: [Integer] -> [Integer]
```

Remember that doubleEveryOther doubles every other number starting from the right, i.e. the second-to-last, fourth-to-last, and so on. This is easiest to do if the the list is put reverse order, so that one can count from the left instead. Check out the builtin `reverse` function.

Examples:

$$\texttt{doubleEveryOther}[8,7,6,5] = [16,7,12,5] \tag{8}$$
$$\texttt{doubleEveryOther}[1,2,3] = [1,4,3] \tag{9}$$

Task: Implement and test the `doubleEveryOther` function.

3.4 Exercise 4

We need to write a function to calculate the digit sum of an arbitrary integer. The digit sum is the sum of the digits, e.g. 7, 16, 25, and 34 each have 7 as the digit sum. Thus we are looking for a function with the following signature:

```
sumDigits :: Integer -> Integer
```

You probably want to use `toDigits` as an auxiliary.

Task: Implement and test the `sumDigits` function.

3.5 Exercise 5

The output of doubleEveryOther has a mix of one-digit and two-digit numbers. Define the function

```
sumDigits :: [Integer] -> Integer
```

to calculate the sum of all digits.

Examples:

$$\texttt{sumDigits}[16,7,12,5] = 1+6+7+1+2+5 = 22$$

The `sumDigits` can be used as an auxiliary here.

Task: Implement and test the `sumDigits` function.

3.6 Exercise 6

Define the function

```
validate :: Integer -> Bool
```

that indicates whether an Integer could be a valid credit card number. This will use all functions defined in the previous exercises.

Examples:

$$\texttt{validate}4012888888881881 = \text{True} \tag{10}$$
$$\texttt{validate}4012888888881882 = \text{False} \tag{11}$$

Task: Implement and test the `validate` function.