Functional Programming 2014/2015
Assignment 1: Lists

Ruud Koot

September 16, 2014

In this exercise we will read in a database, perform a simple query on it and present the results to the user in an aesthetically pleasing form. Most exercises can be completed by combing functions from the *Prelude* and the libraries *Data.Char*, *Data.List* and *Data.Maybe* and contain a hint on which functions you could use from these libraries; often a completely different solution, not using these functions, is also possible. A starting framework and the sample database can be found on the Assignments page on the course website.

1 Parsing

A plain text database consists of a number of lines (each line is called a *row*), with on each line a fixed number of *fieds* separated by a single space. The first row a database table is called the *header* and contains the names of the columns in the table. An example of such a database would be:

```
first last gender salary
Alice Allen female 82000
Bob Baker male 70000
Carol Clarke female 50000
Dan Davies male 45000
Eve Evans female 275000
```

One way of modeling such databases in Haskell would be using the following types:

$$\begin{aligned}&\textbf{type } \mathit{Field} = \mathit{String}\\&\textbf{type } \mathit{Row} \;\; = [\mathit{Field}]\\&\textbf{type } \mathit{Table} = [\mathit{Row}]\end{aligned}$$

A field is always modeled as a string (even though the database may contain strings that look very much like numbers), a row is a list of fields and a table a list of rows. The head of this list corresponds to the header of the table. (A valid table always has a header and always has at least one column.)

There are several "problems" with this model: for example, it does not enforce that each of the rows in the table must have the same number of fields. However, for the purposes of this first assignment it will suffice. You may assume that all the databases that are presented to program will be well-formed, that is to say, they will always have the same number of fields on each line.

The form in which data is stored inside a file, printed or written on paper, or entered from the keyboard is called its *concrete syntax*. The form in which data is manipulated inside a program is called its *abstract syntax*. The process of transforming some object represented in its concrete syntax into its representation in abstract syntax is called *parsing*.

Exercise 1. *Write a function* $parseTable :: [String] \to Table$ *that parses a table represented in its concrete syntax as a list of strings (each corresponding to a single line in the input) into its abstract syntax. (Hint: use the function* $words$ *from the* $Prelude$*.)*

2 Pretty printing

In the previous exercise we have seen how we can turn concrete syntax into abstract syntax. The reverse operation—turning abstract syntax into concrete syntax—is often called *pretty printing* or *compilation*. In our case we do not want to convert our abstract syntax into the original concrete syntax, but into a different concrete syntax that is easier to read for humans:

```
+-----+------+------+------+
|FIRST|LAST  |GENDER|SALARY|
+-----+------+------+------+
|Alice|Allen |female| 82000|
|Bob  |Baker |male  | 70000|
|Carol|Clarke|female| 50000|
|Dan  |Davies|male  | 45000|
|Eve  |Evans |female|275000|
+-----+------+------+------+
```

An apt name for this process might be "prettier printing". Note that we have done several things to make the result look nice:

1. We have made the width of each column exactly as wide as the widest field in this column (including the name in the header).
2. We have added a very fancy looking border around the table, the header and columns.
3. We have typeset the names of the columns in the header in uppercase.
4. We have right-aligned fields that look like (whole) numbers.

Exercise 2. *Write a function* $printLine :: [Int] \to String$ *that, given a list of widths of columns, returns a string containing a horizontal line. For example,* $printLine\ [5, 6, 6, 6]$ *should return the line* `"+-----+------+------+------+"`*. (Hint: use the function* $replicate$*.)*

If you can write this function using $foldr$ you will get more points for style.

Exercise 3. *Write a function* $printField :: Int \to String \to String$ *that, given a desired width for a field and the contents of a fields, returns a formatted field by adding additional whitespace. If the field only consists of numerical digits, the field should be right-aligned, otherwise it should be left-aligned. (Hint: use the functions* all*,* $isDigit$ *and* $replicate$*.)*

The function *printField* should satisfy the property:

$$\forall n\ s.n \geqslant length\ s \Rightarrow length\ (printField\ n\ s) \equiv n$$

Later in the course we shall see how we can use these properties to test the correctness of a program, or even proved that such properties must always hold for a given program.

Exercise 4. *Write a function* $printRow :: [(Int, String)] \to String$ *that, given a list of pairs—the left element giving the desired length of a field and the right element its contents—formats one row in the table. For example,*

$$printRow\ [(5, \texttt{"Alice"}), (6, \texttt{"Allen"}), (6, \texttt{"female"}), (6, \texttt{"82000"})]$$

should return the formatted row

```
"|Alice|Allen |female| 82000|"
```

(Hint: use the functions $intercalate$, map *and* $uncurry$*.)*

Exercise 5. *Write a function* $columnWidths :: Table \to [Int]$ *that, given a table, computes the necessary widths of all the columns. (Hint: use the functions* $length$, map, $maximum$ *and* $transpose$*.)*

Exercise 6. *Write a function* $printTable :: Table \to [String]$ *that pretty prints the whole table. (Hint: use the functions* map, $toUpper$ *and* zip*.)*

3 Querying

Finally we will write a few simple query operations to extract data from the tables.

Exercise 7. *Write a function* $select :: Field \to Field \to Table \to Table$ *that given a column name and a field value, selects only those rows from the table that have the given field value in the given column. For example, applying the query operation*

$$select\ \texttt{"gender"}\ \texttt{"male"}$$

to the table

```
+-----+------+
|FIRST|GENDER|
+-----+------+
|Alice|female|
|Bob  |male  |
|Carol|female|
|Dan  |male  |
|Eve  |female|
+-----+------+
```

should result in the table

```
+-----+------+
|FIRST|GENDER|
+-----+------+
|Bob  |male  |
|Dan  |male  |
+-----+------+
```

If the given column is not present in the table then the table should be returned unchanged. (Hint: use the functions (!!)*,* $elemIndex$*,* $filter$ *and* $maybe$*.)*

Exercise 8. *Write a function* $project :: [Field] \to Table \to Table$ *that projects several columns from a table. For example, applying the query operation*

$$project\ [\texttt{"last"}, \texttt{"first"}, \texttt{"salary"}]$$

to the table

```
+-----+------+------+------+
|FIRST|LAST  |GENDER|SALARY|
+-----+------+------+------+
|Alice|Allen |female| 82000|
|Carol|Clarke|female| 50000|
|Eve  |Evans |female|275000|
+-----+------+------+------+
```

should result in the table

```
+------+-----+------+
|LAST  |FIRST|SALARY|
+------+-----+------+
|Allen |Alice| 82000|
|Clarke|Carol| 50000|
|Evans |Eve  |275000|
+------+-----+------+
```

If a given column is not present in the original table it should be omitted from the resulting table. (Hint: use the functions (!!)*,* $elemIndex$*,* map*,* $mapMaybe$*,* $transpose$*.)*

4 Wrapping up

We can tie parsing, printing and two query operations together using:

$$\begin{aligned} &exercise :: [String] \to [String] \\ &exercise = parseTable \ggg select\ \texttt{"gender"}\ \texttt{"male"} \\ &\qquad\qquad \ggg project\ [\texttt{"last"}, \texttt{"first"}, \texttt{"salary"}] \ggg printTable \end{aligned}$$

and have the program reads and write from and to standard input and standard output using:

$$\begin{aligned} &main :: IO\ () \\ &main = interact\ (lines \ggg exercise \ggg unlines) \end{aligned}$$

5 Reflection

The following questions are intended to help provoke some reflective thoughts about the exercise you just made and what you just learned. You can leave the answers in a comment at the end of your source code. Write clearly, but briefly; say neither more, nor less, than is necessary.

Question 1. *Assume you were asked to implement this program in an imperative programming language such as C♯ or Java, but were not provided with as much guidance on how to structure your program, nor had made this exercise in a functional language before. How would you have structured you program?*

Question 2. *Now that you have made this exercise in a functional programming language, do you think that you would implement this program differently in an imperative language than if you had not? Especially think about the amount of work a particular function does, the types of the functions, the use of higher-order functions, and the use of side-effects ($System.Console.WriteLine$ is a side-effecting function!)*

Question 3. *In Exercise 3 we mentioned that $printField$ should satisfy the property $\forall n\ s.n \geqslant length\ s \Rightarrow length\ (printField\ n\ s) \equiv n$. Does it also satisfy the property $\forall n\ s.length\ (printField\ n\ s) \equiv n$? Would it be a problem in this program if it would not?*

Question 4. *The property $\forall n\ s.n \geqslant length\ s \Rightarrow length\ (printField\ n\ s) \equiv n$ is written in a mixture of predicate logic and Haskell. Could you express it as a mixture of predicate logic and C♯ or Java? How would you have phrased this property if you had implemented $printField$ using $System.Console.WriteLine$? If I wrongly claimed that you program does not meet its specification, which of the formulations would you prefer to use to argue that you deserve a higher grade?*

Question 5. *In the function $printTable$, how often do you compute the required widths of the fields?*

Question 6. *While we guaranteed the input database contains at least one column, it is actually possible to create an "empty" table with no columns using the $project$ operation (We will be nice and not test for this corner case, however.) How do you think such an empty table should be represented in both its abstract and its concrete syntax? Could someone else have a different opinion on this matter? Which of your functions would you have to modify to correctly handle this case?*