Computing for Musicology (0809.F104N5)
4. Map & filter for (quantitative) musical analysis

J.N. Oliveira

Dept. Informática,
Universidade do Minho
Braga, Portugal

May 2009
Licenciatura em Música
(http://www.musica.reitoria.uminho.pt/licenciatura.html)
Universidade do Minho
Braga

Recall word mappings

- Recall the *map* operator, which we've seen being extremely useful in Haskell programming and music processing
- Examples of our use of *map* are
  - words, ... — eg. conversion to uppercase letters:

    $$map\ toUpper\ \texttt{"Mendelssohn"} = \texttt{"MENDELSSOHN"}$$

  - music parts, ... — eg. augmentation, and so on:

    $$map\ (id \times (/n))\ p$$

    augments/diminishes part p depending on whether n is smaller or larger than 1.

Recall word filtering

Further recall the example

$$\textit{filter notVowel}$$
```
"Joseph Haydn died two hundred years ago"
```

yielding

```
"Jsph Hydn dd tw hndrd yrs g"
```

based on **property** 'being a vowel' or not:

$$\textit{notVowel}\ c = \neg\,(c \in \texttt{"aeiouAEIOU"})$$

Filtering extends to any kind of list in Haskell, not just words, eg:

$$\textit{filter odd}\ [1\,..]$$

yields the list of all odd natural numbers

$$[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, \ldots]$$

Implementing *filter*

Let us try and define *filter p* ourselves. As earlier on, three properties of the function being defined need to be identified:

$$\begin{aligned} &filter\ p\ [\,] = ... \\ &filter\ p\ [c] = ... \\ &filter\ p\ (w \mathbin{+\!\!+} y) = \ldots\ (filter\ p\ w)\ \ldots\ (filter\ p\ y)\ \ldots \end{aligned}$$

The first and last aren't too difficult to find:

$$\begin{aligned} &filter\ p\ [\,] = [\,] \\ &filter\ p\ [c] = ... \\ &filter\ p\ (w \mathbin{+\!\!+} y) = (filter\ p\ w) \mathbin{+\!\!+} (filter\ p\ y) \end{aligned}$$

Implementing *filter*

The second requires testing whether c fulfills selection criterion (property) p. Haskell offers special syntax for this, either

$$\begin{array}{l} \mathit{filter}\ p\ [c] \\ \quad \mid p\ c = [c] \\ \quad \mid \mathit{otherwise} = [\,] \end{array}$$

or its inlined version:

$$\mathit{filter}\ p\ [c] = \mathbf{if}\ p\ c\ \mathbf{then}\ [c]\ \mathbf{else}\ [\,]$$

Implementing *filter*

To complete the encoding, we incorporate the middle clause into the third by instantiating w to $[c]$ and simplifying:

$$
\begin{array}{ll}
 & \mathit{filter}\ p\ ([c] +\!\!+\ y) = (\mathit{filter}\ p\ [c]) +\!\!+ (\mathit{filter}\ p\ y) \\
\equiv & \{\ \text{simplification of left hand side}\ \} \\
 & \mathit{filter}\ p\ (c : y) = (\mathit{filter}\ p\ [c]) +\!\!+ (\mathit{filter}\ p\ y) \\
\equiv & \{\ \text{substitution in right hand side}\ \} \\
 & \mathit{filter}\ p\ (c : y) = (\textbf{if}\ p\ c\ \textbf{then}\ [c]\ \textbf{else}\ [\,]) +\!\!+ (\mathit{filter}\ p\ y)
\end{array}
$$

Implementing *filter*

Putting everything together, we obtain the following piece of Haskell:

$$\begin{aligned}&\mathit{filter}\ p\ [\,] = [\,]\\ &\mathit{filter}\ p\ (c : y) = (\textbf{if}\ p\ c\ \textbf{then}\ [c]\ \textbf{else}\ [\,]) +\!\!+ (\mathit{filter}\ p\ y)\end{aligned}$$

By doing a similar exercise one obtains the following Haskell for *map f*:

$$\begin{aligned}&\mathit{map}\ f\ [\,] = [\,]\\ &\mathit{map}\ f\ (c : y) = [f\ \ c] +\!\!+ (\mathit{map}\ f\ \ y)\end{aligned}$$

which simplifies to:

$$\begin{aligned}&\mathit{map}\ f\ [\,] = [\,]\\ &\mathit{map}\ f\ (c : y) = (f\ \ c) : (\mathit{map}\ f\ \ y)\end{aligned}$$

Combining map and filter

These two operations — *map* and *filter* play a major role in programming, in a way such that they complement each other:

- *filter p* **selects** those elements in a list which are of interest according to selection criterion p;
- *map f* **transforms** all elements in a list, one after the other, according to transformation f.

One can combine these two operations in a single operation using **composition**:

$$map\ f \cdot filter\ p$$

This performs a selection **followed by** a transformation. For instance,

$$((map\ toUpper) \cdot (filter\ notVowel))\ \texttt{"Joseph Haydn"}$$

yields `"JSPH HYDN"`.

Comprehending map and filter

Haskell offers an alternative notation for

(map f . filter p)l

in which we easily see **selection** and **transformation** explicitly combined:

$$[f\ c \mid c \leftarrow l, p\ c] = (map\ f \cdot filter\ p)\ l$$

Notation $[f\ c \mid c \leftarrow l, p\ c]$ means:

take those elements from l, one at a time, which satisfy p, and transform them via f.

For instance,

$$[toUpper\ c \mid c \leftarrow \texttt{"Joseph Haydn"}, notVowel\ c]$$

yields the same `"JSPH HYDN"`.

Quantitative analysis

- Maps and filters are also useful in performing **quantitative** analysis.
- Suppose, for instance, that a given list *l* contains all the published works of a given composer and that, for each such work *w*:
  - *date w* yields its date of composition (eg. 1805)
  - *desc w* yields the title, or description of *w* (eg. `"Symphony No. 3 in E flat major 'Eroica'"`)
  - *op w* yields its opus number (eg. opus 55)
- Now suppose we want to count the number of works composed at a given date *d*.

Querying

Clearly, we have to filter list l by selecting only the works composed by such date, eg. by writing

$$[w \mid w \leftarrow l, date\ w \equiv d]$$

Then *counting* amounts to calculating the length of such a list of selections:

$$length\ [o \mid o \leftarrow l, date\ o \equiv d]$$

What we have just done is known in the literature of **information retrieval** as *querying*:

> *Given a particular source of data (list l in our example),* **querying** *such data source consists of obtaining information (eg. statistical) from such information.*

Querying

- As a rule, queries are to be repeated over and over again as the data source evolves (eg. as performed by the National Statistics Institute).
- It is thus a good idea to give queries a *name*, as we do below concerning our query in Haskell is concerned:

$$nrOfWorksByDat\ d\ l = length\,[o \mid o \leftarrow l, date\ o \equiv d]$$

An alternative definition for this query involving *filter* is:

$$nrOfWorksByDat\ d = length \cdot (filter\ ((\equiv d) \cdot date))$$

Exercise

Exercise 1: In the context of the previous slides, complete the query in Haskell

$$getSymphonies\ l = \ldots.$$

which searches data source l and finds the opus number and publishing date of each symphony (should yield the empty list in case the composer wrote none!).
Suggestion: involves checking whether word `"Symphony"` is a prefix of the description of each work. Writing auxiliary predicate

$$isSymphony\ w \equiv \ldots prefix\ \ldots$$

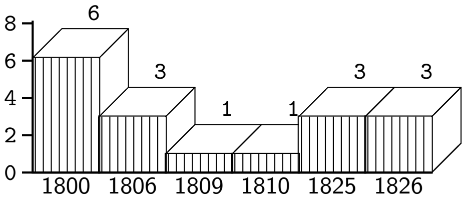
will help. Load library `LvB.hs` (catalogue of woks by L. van Beethoven (1770-1827), WoO's excluded) and test your query.
□

Counting data and building histograms

From the Wikipedia:

> *In statistics, a* **histogram** *is a graphical display of tabulated frequencies, shown as bars. (...) In a more general (...) sense, a histogram is a mapping m_i that counts the number of observations that fall into various disjoint categories (known as bins)*

For instance,



depicts the histogram of the number of string quartets composed per year by L. van Beethoven (1770-1827).

Counting and building histograms

- In Haskell, the contents of histograms are simply lists of pairs,

  [(b1,c1),...,(bn,cn)]

  where the *b*s are *bins* and the *c*s are numbers.
- For example, list

  $$[(1800,6),(1806,3),(1809,1),(1810,1),(1825,3),(1826,3)]$$

  contains the information depicted in the previous slide, where the bins are years 1800,1806,...,1826.
- From any list one can calculate the histogram of its contents by counting how repeated each element in the list is:

  $$\begin{array}{l} \mathit{hist}\ l = \mathit{nub}\ [(x, \mathit{count}\ x\ l) \mid x \leftarrow l] \\ \quad \textbf{where}\ \mathit{count}\ a\ l = \mathit{length}\ [x \mid x \leftarrow l, x \equiv a] \end{array}$$

  (The *nub* function eliminates repetition of pairs.)

Counting and building histograms

- Clearly, in building a histogram most of the work goes into selecting all occurrences of the data of interest in a list.
- Then *hist* yields the histogram from this list.
- Example: we want to produce the histogram of keys in Beethoven's works. For this we assume that $m1\ x$ yields the key of work x.
- Clearly, *map m1* extracts the list of all keys, to be passed on to *hist*. The query to build is then:

$$keyHist = hist \cdot (map\ m1)$$

Exercises

Exercise 2: Write the query which computed the *works per year* histogram of Beethoven's string quartets given above.

□

Exercise 3: Write the query which computes the *works per year* histogram of Beethoven's piano sonatas between 1800 and 1810.

□

Exercise 4: Run *keyHist* for Beethoven's violin and piano sonatas only.

□

Exercises

Exercise 5: Implement the function *concat* which joins a list of lists into a single list by completing and simplifying

$$\begin{aligned}
&concat\ [\,] = \ldots \\
&concat\ [a] = \ldots \\
&concat\ (l + r) = \ldots
\end{aligned}$$

□

A glimpse at multi-dimensional analysis

(Not included in the current version of these slides)

More about Haskell

If you want to know more about Haskell (including its application to music synthesis) have a look at the following (really good) book:

> *P. Hudak: The Haskell School of Expression - Learning Functional Programming Through Multimedia. Cambridge University Press, 2000. ISBN 0-521-64408-9.*