Computing for Musicology (0809.F104N5)
2. Introduction to Programming with Numbers and Words

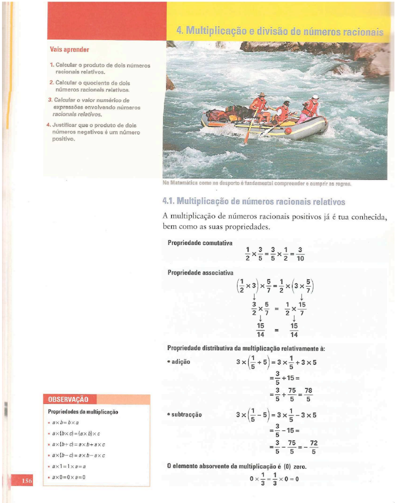
J.N. Oliveira

Dept. Informática,
Universidade do Minho
Braga, Portugal

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

From a middle school textbook

First page of chapter on multiplying and dividing rational numbers (7th year):



4. Multiplicação e divisão de números racionais

Vais aprender

1. Calcular o produto de dois números racionais relativos.
2. Calcular o quociente de dois números racionais relativos.
3. Calcular o valor numérico de expressões envolvendo números racionais relativos.
4. Justificar que o produto de dois números negativos é um número positivo.

Na Matemática como no desporto é fundamental compreender e cumprir as regras.

4.1. Multiplicação de números racionais relativos

A multiplicação de números racionais positivos já é tua conhecida, bem como as suas propriedades.

Propriedade comutativa

$$\frac{1}{2}\times\frac{3}{5}=\frac{3}{5}\times\frac{1}{2}=\frac{3}{10}$$

Propriedade associativa

$$\left(\frac{1}{2}\times 3\right)\times\frac{5}{7}=\frac{1}{2}\times\left(3\times\frac{5}{7}\right)$$
$$\frac{3}{2}\times\frac{5}{7}=\frac{1}{2}\times\frac{15}{7}$$
$$\frac{15}{14}=\frac{15}{14}$$

Propriedade distributiva da multiplicação relativamente à:

- adição $3\times\left(\frac{1}{5}+5\right)=3\times\frac{1}{5}+3\times 5 =\frac{3}{5}+15=\frac{3}{5}+\frac{75}{5}=\frac{78}{5}$
- subtracção $3\times\left(\frac{1}{5}-5\right)=3\times\frac{1}{5}-3\times 5 =\frac{3}{5}-15=\frac{3}{5}-\frac{75}{5}=-\frac{72}{5}$

O elemento absorvente da multiplicação é (0) zero.

$$0\times\frac{1}{3}=\frac{1}{3}\times 0=0$$

OBSERVAÇÃO

Propriedades da multiplicação

- $a\times b=b\times a$
- $a\times(b\times c)=(a\times b)\times c$
- $a\times(b+c)=a\times b+a\times c$
- $a\times(b-c)=a\times b-a\times c$
- $a\times 1=1\times a=a$
- $a\times 0=0\times a=0$

156

From a middle school textbook

Draw your attention to the text-box at the bottom, on the left:

OBSERVAÇÃO

Propriedades da multiplicação

- $a \times b = b \times a$
- $a \times (b \times c) = (a \times b) \times c$
- $a \times (b + c) = a \times b + a \times c$
- $a \times (b - c) = a \times b - a \times c$
- $a \times 1 = 1 \times a = a$
- $a \times 0 = 0 \times a = 0$

• **subtracção**

O elemento absorv

156

First program: multiplication

The **properties** of multiplication are enough for us to start writing programs involving addition and multiplication, eg.

$$a \times 0 = 0 \quad (1)$$
$$a \times 1 = a \quad (2)$$
$$a \times (b + c) = (a \times b) + (a \times c) \quad (3)$$

Let us see how: for $c = 1$ one has

$$a \times 0 = 0$$
$$a \times 1 = a$$
$$a \times (b + 1) = (a \times b) + (a \times 1)$$

First program: multiplication

This runs (eg. in Haskell) and can actually be simplified into

$$a \times 0 = 0$$
$$a \times (b + 1) = (a \times b) + a$$

(Just replace $a \times 1$ by a and delete second clause, which is a consequence of the other two.)

Exercise 1: *From the following properties of exponentials,*

$$\begin{aligned} a^1 &= a \\ a^{(b+c)} &= a^b \times a^c \\ a^0 &= 1 \end{aligned}$$

write an Haskell program which computes a^x*.*

□

Numbers in Haskell

Exercise 2: *From the following properties of addition,*

$$
\begin{aligned}
a + 0 &= a \\
a + (b + c) &= (a + b) + c
\end{aligned}
$$

infer an Haskell program which computes addition itself.

□

Words in Haskell

- Further to numbers, in Haskell we can handle words, that is, objects such as `"Haydn"`, `"Mendelssohn"`, and so on.
- Note the use of `""` in words: in fact, there is a (great!) difference between the *word* `"Mendelssohn"` and the *individual* Felix Mendelssohn, a composer who was born 200 years ago.

F. Mendelssohn-Bartholdy (1809-1847)

In this way, while Haskell is able to tell us that the word `"Mendelssohn"` is made of 11 letters,

$$length\ \texttt{"Mendelssohn"} = 11$$

it is unable to infer that Mendelssohn died in 1847 (you need to ask a music historian about the truth of such fact).

Words in Haskell

- Further to numbers, in Haskell we can handle words, that is, objects such as `"Haydn"`, `"Mendelssohn"`, and so on.
- Note the use of `""` in words: in fact, there is a (great!) difference between the *word* `"Mendelssohn"` and the *individual* Felix Mendelssohn, a composer who was born 200 years ago.

F. Mendelssohn-Bartholdy (1809-1847)

In this way, while Haskell is able to tell us that the word `"Mendelssohn"` is made of 11 letters,

$$\textit{length}\ \texttt{"Mendelssohn"} = 11$$

it is unable to infer that Mendelssohn died in 1847 (you need to ask a music historian about the truth of such fact).

Words in Haskell

Haskell provides a rich set of operations on words. Let us see some of these:

- word **inversion**:

  reverse `"Mendelssohn"` = `"nhossledneM"`

- word **chaining**:

  `"Mendelssohn"` ++ `"Bartholdy"` =
  `"MendelssohnBartholdy"`

  or (if you like)

  `"Mendelssohn"` ++ `"-"` ++ `"Bartholdy"` =
  `"Mendelssohn-Bartholdy"`

Words in Haskell

Haskell provides a rich set of operations on words. Let us see some of these:

- word **inversion**:

  reverse "Mendelssohn" = "nhossledneM"

- word **chaining**:

  "Mendelssohn" ++ "Bartholdy" =
  "MendelssohnBartholdy"

  or (if you like)

  "Mendelssohn" ++ "-" ++ "Bartholdy" =
  "Mendelssohn-Bartholdy"

Words in Haskell

Further operations on words:

- removing **repeated** characters from words:

  nub "Mendelssohn" = "Mendlsoh"

- checking **prefixes**:

  isPrefixOf "Mendel" "Mendelssohn" = *True*
  isPrefixOf "Mendlsoh" "Mendelssohn" = *False*

- **sorting** the characters of words in increasing order:

  sort "Mendelssohn" = "Mdeehlnnoss"

- **equality** of words, eg.

  "Mendel"=="Mendelssohn" *yields False*

Words in Haskell

Further operations on words:

- removing **repeated** characters from words:

  nub "Mendelssohn" = "Mendlsoh"

- checking **prefixes**:

  isPrefixOf "Mendel" "Mendelssohn" = *True*
  isPrefixOf "Mendlsoh" "Mendelssohn" = *False*

- **sorting** the characters of words in increasing order:

  sort "Mendelssohn" = "Mdeehlnnoss"

- **equality** of words, eg.

  "Mendel"=="Mendelssohn" *yields False*

Words in Haskell

Further operations on words:

- removing **repeated** characters from words:

  nub "Mendelssohn" = "Mendlsoh"

- checking **prefixes**:

  isPrefixOf "Mendel" "Mendelssohn" = *True*
  isPrefixOf "Mendlsoh" "Mendelssohn" = *False*

- **sorting** the characters of words in increasing order:

  sort "Mendelssohn" = "Mdeehlnnoss"

- **equality** of words, eg.

  "Mendel"=="Mendelssohn" *yields False*

Words in Haskell

Further operations on words:

- removing **repeated** characters from words:

  nub `"Mendelssohn"` = `"Mendlsoh"`

- checking **prefixes**:

  isPrefixOf `"Mendel"` `"Mendelssohn"` = *True*
  isPrefixOf `"Mendlsoh"` `"Mendelssohn"` = *False*

- **sorting** the characters of words in increasing order:

  sort `"Mendelssohn"` = `"Mdeehlnnoss"`

- **equality** of words, eg.

  `"Mendel"==“Mendelssohn"` *yields False*

Sentences in Haskell

- Words can have characters in them other than lowercase and uppercase letters.
- Words with spaces are better viewed as **sentences**, eg.

  `"Mendelssohn died in 1847"`

  Sentences can be split into sequences of words:

  words `"Mendelssohn died in 1847"` = `["Mendelssohn","died","in","1847"]`

- So, sentence `"Mendelssohn died in 1847"`, which has 24 characters,

  length `"Mendelssohn died in 1847"` = 24

  is made of 4 words:

  length (*words* `"Mendelssohn died in 1847"`) = 4

Sentences in Haskell

- Words can have characters in them other than lowercase and uppercase letters.
- Words with spaces are better viewed as **sentences**, eg.

  `"Mendelssohn died in 1847"`

  Sentences can be split into sequences of words:

  words `"Mendelssohn died in 1847"` =
  `["Mendelssohn", "died", "in", "1847"]`

- So, sentence `"Mendelssohn died in 1847"`, which has 24 characters,

  length `"Mendelssohn died in 1847"` = 24

  is made of 4 words:

  length (*words* `"Mendelssohn died in 1847"`) = 4

Sentences in Haskell

- Words can have characters in them other than lowercase and uppercase letters.
- Words with spaces are better viewed as **sentences**, eg.

  `"Mendelssohn died in 1847"`

  Sentences can be split into sequences of words:

  words `"Mendelssohn died in 1847"` =
  `["Mendelssohn", "died", "in", "1847"]`

- So, sentence `"Mendelssohn died in 1847"`, which has 24 characters,

  length `"Mendelssohn died in 1847"` $= 24$

  is made of 4 words:

  length (*words* `"Mendelssohn died in 1847"`) $= 4$

Numbers versus words

- Also note the difference between 1847 (a *number*) and its denotation "1847" (a *word*).
- We say that word "1847" *shows* (or prints) number 1847. Check this by evaluating

$$\mathit{show}\ 1847$$

Exercise 3: Check the difference between numbers and words by evaluating the following expressions:

a) $1847 + 2$
b) "1847" $+ 2$
c) "died in " $+\!\!+\ 1847$
c) "died in " $+\!\!+\ (\mathit{show}\ 1847)$

Numbers versus words

- Also note the difference between 1847 (a *number*) and its denotation `"1847"` (a *word*).
- We say that word `"1847"` *shows* (or prints) number 1847. Check this by evaluating

$$\mathit{show}\ 1847$$

Exercise 4: Check the difference between numbers and words by evaluating the following expressions:

a) $1847 + 2$
b) `"1847"` $+ 2$
c) `"died in "` $+\!\!+\ 1847$
c) `"died in "` $+\!\!+\ (\mathit{show}\ 1847)$

Numbers versus words

- Also note the difference between 1847 (a *number*) and its denotation `"1847"` (a *word*).
- We say that word `"1847"` *shows* (or prints) number 1847. Check this by evaluating

  show 1847

Exercise 5: Check the difference between numbers and words by evaluating the following expressions:

a) $1847 + 2$
b) `"1847"` $+ 2$
c) `"died in "` $+\!\!+\ 1847$
c) `"died in "` $+\!\!+$ (*show* 1847)

□

Empty words and empty sentences

Words can have only one character, cf.

$$\textit{length}\ \texttt{"H"} = 1$$

and even no characters at all:

$$\textit{length}\ \texttt{""} = 0$$

This last word — the **empty word** — adds nothing to any other given word w:

$$w ++ \texttt{""} = \texttt{""} ++ w = w$$

This leads us to the operator which yields all **prefixes** of a given word,

$$\textit{inits}\ \texttt{"Haydn"} = [\texttt{""}, \texttt{"H"}, \texttt{"Ha"}, \texttt{"Hay"}, \texttt{"Hayd"}, \texttt{"Haydn"}]$$

sorted by dictionary order, in which "" is smallest.

F.J. Haydn (1732-1809)

Empty words and empty sentences

Words can have only one character, cf.

$$\textit{length}\ \texttt{"H"} = 1$$

and even no characters at all:

$$\textit{length}\ \texttt{""} = 0$$

This last word — the **empty word** — adds nothing to any other given word w:

$$w +\!\!+ \texttt{""} = \texttt{""} +\!\!+ w = w$$

F.J. Haydn (1732-1809)

This leads us to the operator which yields all **prefixes** of a given word,

$$\textit{inits}\ \texttt{"Haydn"} = [\texttt{""}, \texttt{"H"}, \texttt{"Ha"}, \texttt{"Hay"}, \texttt{"Hayd"}, \texttt{"Haydn"}]$$

sorted by dictionary order, in which `""` is smallest.

Words are made of characters

- By evaluating

  head "Mendelssohn"

  you run the operation *head* which yields the first letter of a given word, if it exists (thus never evaluate *head* ""...)
- Note that 'M' = *head* "Mendelssohn" is a letter (character), not a word.
- So, letter 'M' is different from "M", the singleton word which contains only character 'M'.
- To check that words are sequences of characters check

  "Haydn"==['H','a','y','d','n']

Words are made of characters

- By evaluating

  head "Mendelssohn"

  you run the operation *head* which yields the first letter of a given word, if it exists (thus never evaluate *head* ""...)
- Note that 'M' = *head* "Mendelssohn" is a letter (character), not a word.
- So, letter 'M' is different from "M", the singleton word which contains only character 'M'.
- To check that words are sequences of characters check

  "Haydn"==['H','a','y','d','n']

Words are made of characters

- By evaluating

  $$\textit{head}\ \texttt{"Mendelssohn"}$$

  you run the operation *head* which yields the first letter of a given word, if it exists (thus never evaluate *head* `""`...)
- Note that `'M'` = *head* `"Mendelssohn"` is a letter (character), not a word.
- So, letter `'M'` is different from `"M"`, the singleton word which contains only character `'M'`.
- To check that words are sequences of characters check

  $$\texttt{"Haydn"} == [\texttt{'H'}, \texttt{'a'}, \texttt{'y'}, \texttt{'d'}, \texttt{'n'}]$$

Building words out of characters

- How do you add a character, say 'F', at the front of a given word, say "Mendelssohn"? You have two ways: either typing

  $$\texttt{"F"} \mathbin{+\!\!+} \texttt{"Mendelssohn"}$$

  or

  $$\texttt{'F'} : \texttt{"Mendelssohn"}$$

  Both yield "FMendelssohn".
- The (:) operator is known as *cons*, a prefix of *construct*, which is such that

  $$c : w = \texttt{"c"} \mathbin{+\!\!+} w$$

  meaning that it can be used to build words by adding characters to the empty word:

  $$\texttt{'H'} : (\texttt{'a'} : (\texttt{'y'} : (\texttt{'d'} : (\texttt{'n'} : \texttt{""})))) = \texttt{"Haydn"}$$

Building words out of characters

- How do you add a character, say `'F'`, at the front of a given word, say `"Mendelssohn"`? You have two ways: either typing

  $$\texttt{"F"} +\!\!+ \texttt{"Mendelssohn"}$$

  or

  $$\texttt{'F'} : \texttt{"Mendelssohn"}$$

  Both yield `"FMendelssohn"`.
- The (:) op-
  erator is known as *cons*, a prefix of *construct*, which is such that

  $$c : w \;=\; \texttt{"c"} +\!\!+ w$$

  meaning that it can
  be used to build words by adding characters to the empty word:

  $$\texttt{'H'} : (\texttt{'a'} : (\texttt{'y'} : (\texttt{'d'} : (\texttt{'n'} : \texttt{""})))) = \texttt{"Haydn"}$$

Words which are 'rondos'

- Suppose "ABCD" is a word describing a particular piece of music made of parts 'A', 'B', 'C' and 'D'.
- Now run

  intersperse 'R' "ABCD"

  in your Haskell calculator, where 'R' describes yet another part. You will obtain

  "ARBRCRD"

  — that is, the *rondo* word where episodes 'A', 'B', 'C' and 'D' alternate with refrain 'R'.

Words which are ‘rondos’

- Suppose `"ABCD"` is a word describing a particular piece of music made of parts `'A'`, `'B'`, `'C'` and `'D'`.
- Now run

  intersperse `'R' "ABCD"`

  in your Haskell calculator, where `'R'` describes yet another part. You will obtain

  `"ARBRCRD"`

  — that is, the *rondo* word where episodes `'A'`, `'B'`, `'C'` and `'D'` alternate with refrain `'R'`.

'Canon perpetuus' kind of words

- Take your rondo "ARBRCRD" word and run

  cycle "ARBRCRD"

  in your Haskell calculator. You will see you little rondo repeated forever,

  "ARBRCRDARBRCRDARBRCRDARBRCRDARBRCRDARB..."

  (The only way to stop this is to type Ctr-c.)
- Note the mathematical property

  $$intersperse\ x\ (cycle\ w) = cycle\ (intersperse\ x\ w)$$
- **Infinite** words such as the one just built above will be very useful in our formalization of **music notation** to come up soon.

Word filtering

Suppose that, from a rondo-word, you want to extract the episodes in the order they take place. You can write

filter ($\neq$ 'R') "ARBRCRD"

to recover word "ABCD" without refrain 'R'. This literally means:

filter out all instances of 'R' *from* "ARBRCRD"

Put in other words:

filter word "ARBRCRD" *so as to keep* **only** *the letters* **different** *from* 'R'

If you wish to *keep* the 'R's instead of deleting them just type

filter (=='R') "ARBRCRD"

to obtain the word "RRR" containing the three instances of the refrain.

Word filtering

Suppose that, from a rondo-word, you want to extract the episodes in the order they take place. You can write

filter ($\not\equiv$ 'R') "ARBRCRD"

to recover word "ABCD" without refrain 'R'. This literally means:

filter out all instances of 'R' *from* "ARBRCRD"

Put in other words:

filter word "ARBRCRD" *so as to keep* **only** *the letters* **different** *from* 'R'

If you wish to *keep* the 'R's instead of deleting them just type

filter (=='R') "ARBRCRD"

to obtain the word "RRR" containing the three instances of the refrain.

Word filtering

Suppose that, from a rondo-word, you want to extract the episodes in the order they take place. You can write

filter $(\neq$ `'R'`$)$ `"ARBRCRD"`

to recover word `"ABCD"` without refrain `'R'`. This literally means:

filter out all instances of `'R'` *from* `"ARBRCRD"`

Put in other words:

filter word `"ARBRCRD"` *so as to keep* **only** *the letters* **different** *from* `'R'`

If you wish to *keep* the `'R'`s instead of deleting them just type

filter $(==$`'R'`$)$ `"ARBRCRD"`

to obtain the word `"RRR"` containing the three instances of the refrain.

Word filtering

- As another example of word filtering, let us see how to drop vowels from words:

  filter notVowel
  `"Joseph Haydn died two hundred years ago"`

  obtaining

  `"Jsph Hydn dd tw hndrd yrs g"`

  The key in this process is the specification of the **property** ‘being a vowel’ or not:

  $$notVowel\ c = not\ (c \in \texttt{"aeiouAEIOU"})$$

  Here $c \in w$ checks whether a particular c can be found in word w.

Taking and dropping

Further (standard) operations on words:

- **selecting** *n*-first letters:

  $take\ 7$ `"Mendelssohn"` $=$ `"Mendels"`

  Case of not enough letters:

  $take\ 7$ `"Haydn"` $=$ `"Haydn"`

- **dropping** *n*-first letters:

  $drop\ 7$ `"Mendelssohn"` $=$ `"sohn"`

  Case of not enough letters:

  $drop\ 7$ `"Haydn"` $=$ `""`

Note the **mathematical** property:

$$take\ n\ w \mathbin{+\!\!+} drop\ n\ w \quad = \quad w \tag{4}$$

Taking and dropping

Further (standard) operations on words:

- **selecting** n-first letters:

  $$take\ 7\ \texttt{"Mendelssohn"} = \texttt{"Mendels"}$$

  Case of not enough letters:

  $$take\ 7\ \texttt{"Haydn"} = \texttt{"Haydn"}$$

- **dropping** n-first letters:

  $$drop\ 7\ \texttt{"Mendelssohn"} = \texttt{"sohn"}$$

  Case of not enough letters:

  $$drop\ 7\ \texttt{"Haydn"} = \texttt{""}$$

Note the **mathematical** property:

$$take\ n\ w +\!\!+\ drop\ n\ w \quad = \quad w \tag{4}$$

Taking and dropping

Further (standard) operations on words:

- **selecting** n-first letters:

  $$take\ 7\ \texttt{"Mendelssohn"} = \texttt{"Mendels"}$$

  Case of not enough letters:

  $$take\ 7\ \texttt{"Haydn"} = \texttt{"Haydn"}$$

- **dropping** n-first letters:

  $$drop\ 7\ \texttt{"Mendelssohn"} = \texttt{"sohn"}$$

  Case of not enough letters:

  $$drop\ 7\ \texttt{"Haydn"} = \texttt{""}$$

Note the **mathematical** property:

$$take\ n\ w +\!\!\!+\ drop\ n\ w \quad = \quad w \tag{4}$$

Ciphering words

Julius Caesar (100BC-44BC) is known to have used the following trick to hide the contents of his messages to his army from the enemy by *ciphering* the words:

- **Ciphering:** replace each letter by its **successor** in the Latin alphabet, eg. `"WeAreReadyToAttack"` converted to `"XfBsfSfbezUpBuubdl"`.
- **Deciphering:** replace each letter by its **predecessor** in the Latin alphabet.

Exercise 6: Check that Haskell knows about the Latin alphabet by running

$$
\begin{aligned}
&succ\ \texttt{'A'} = \texttt{'B'}\\
&succ\ \texttt{'B'} = \texttt{'C'}\ ,\ etc\\
&pred\ \texttt{'k'} = \texttt{'j'}\\
&pred\ \texttt{'d'} = \texttt{'c'}\ ,\ etc
\end{aligned}
$$

Ciphering words

Julius Caesar (100BC-44BC) is known to have used the following trick to hide the contents of his messages to his army from the enemy by *ciphering* the words:

- **Ciphering:** replace each letter by its **successor** in the Latin alphabet, eg. `"WeAreReadyToAttack"` converted to `"XfBsfSfbezUpBuubdl"`.
- **Deciphering:** replace each letter by its **predecessor** in the Latin alphabet.

Exercise 7: Check that Haskell knows about the Latin alphabet by running

$$\begin{aligned} &succ\ \texttt{'A'} = \texttt{'B'} \\ &succ\ \texttt{'B'} = \texttt{'C'}\ ,\ etc \\ &pred\ \texttt{'k'} = \texttt{'j'} \\ &pred\ \texttt{'d'} = \texttt{'c'}\ ,\ etc \end{aligned}$$

Word mappings

The effect of applying *succ* or *pred* to **every** letter in a word or sentence is obtained in Haskell by typing, for instance

map succ `"WeAreReadyToAttack"` = `"XfBsfSfbezUpBuubdl"`
map pred `"PlXfBsfSfbezUpp"` = `"OkWeAreReadyToo"`

The *map* operator is extremely useful in Haskell programming, as the following illustration shows:

- conversion to uppercase letters:

  map toUpper `"Mendelssohn"` = `"MENDELSSOHN"`

- conversion to lowercase letters:

  map toLower `"Haydn"` = `"haydn"`

where *toUpper* and *toLower* are the obvious case-conversion operations.

Rebuilding sentences from their words

We have seen how to split a sentence into a sequence of words, recall

words `"Mendelssohn died in 1847"` =
`["Mendelssohn", "died", "in", "1847"]`

Is there the **converse** operation of rebuilding the original sentence from its words? Let us try it:

concat `["Haydn", "died", "in", "1809"]` =
"Haydndiedin1809"

So *concat* merges a sequence of words into a single word. (Can be thought of $(+\!\!+)$ generalized to more than two arguments.)

Rebuilding sentences from their words

However, "Haydndiedin1809" is not what we started from: the spaces are missing. We thus need something else:

$$concat\ (intersperse\ \texttt{" "}\ [\texttt{"Haydn"}, \texttt{"died"}, \texttt{"in"}, \texttt{"1809"}])$$

Mind the following **mathematical** property:

$$concat\ (intersperse\ \texttt{" "}\ (words\ s)) = s$$

Exercise 8: Run *take* 16 (*cycle* "ARBRCRD"). Conclude that Haskell is able to select from infinite words.

□

Exercise 9: Check that *concat* [""] = "" but *concat* "" yelds an error. Why is this so?

□

Let's program with words, not numbers

How difficult is it to write **programs** which handle **words** instead of numbers?

- Conceptually, programs handling words (sentences, etc) are as easy to write as those which handle numbers
- The design principle is the same: programs always arise from (mathematical) properties of the operators we want to write.

Example:

We want to re-invent the $(++)$ *operator which concatenates words.*

Programming with words, not numbers

First of all, we record properties of this operator. Further to the ones already written up,

$$\texttt{""} +\!\!+\, w = w$$
$$\texttt{"a"} +\!\!+\, w = a : w$$

we add the one which tells that you can join words from both ends:

$$(w +\!\!+\, y) +\!\!+\, z = w +\!\!+\, (y +\!\!+\, z)$$

NB: the standard name for this is the **associative** property.

Programming with words, not numbers

Now, substitute w in the third property of

$$"" +\!\!+ w = w$$
$$"a" +\!\!+ w = a : w$$
$$(w +\!\!+ y) +\!\!+ z = w +\!\!+ (y +\!\!+ z)$$

by "a", obtaining:

$$"" +\!\!+ w = w$$
$$"a" +\!\!+ w = a : w$$
$$("a" +\!\!+ y) +\!\!+ z = "a" +\!\!+ (y +\!\!+ z)$$

Then use the second equation to simplify the third (twice):

$$"" +\!\!+ w = w$$
$$"a" +\!\!+ w = a : w$$
$$(a : y) +\!\!+ z = a : (y +\!\!+ z)$$

Programming with words, not numbers

As the second equation is no longer needed, remove it from the program. You are done:

$$\texttt{""} + w = w$$
$$(a : y) + z = a : (y + z)$$

Exercise 10: Knowing that properties

$$length\ \texttt{""} = 0$$
$$length\ (w + y) = length\ w + length\ y$$
$$length\ \texttt{"c"} = 1$$

hold, provide your own version of *length*.

□

Exercises

Exercise 11: From the following properties of $\in$,

$$
\begin{aligned}
&c \in \texttt{""} = \mathit{False} \\
&c \in (w +\!\!+ y) = c \in w \mid c \in w \\
&c \in \texttt{"d"} = c == d
\end{aligned}
$$

provide you own version of this operator.

□

Exercise 12: Complete the following properties of the word reversal operation:

$$
\begin{aligned}
&\mathit{reverse}\ \texttt{""} = \texttt{""} \\
&\mathit{reverse}\ (w +\!\!+ y) = \ldots.. \\
&\mathit{reverse}\ \texttt{"c"} = \ldots.
\end{aligned}
$$

Hence provide your own version of *reverse*.

□

Exercises

Exercise 13: Complete the following properties of the *map f* operator:

$$map\ f\ \texttt{""} = \texttt{""}$$
$$map\ f\ (w \mathbin{+\!\!+} y) = ...$$
$$map\ f\ \texttt{"c"} = ...$$

Hence provide your own version of *map f*.

□

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.