Computing for Musicology (Course code: F104N5) 2. Introduction to Programming with Numbers and Words

J.N. Oliveira

Dept. Informática,
Universidade do Minho
Braga, Portugal

March 2009 (last update: September 2024)

Licenciatura em Música
(http://www.musica.ilch.uminho.pt/)
Universidade do Minho
Braga

Computing — hardware, software, ...

Computing:

- **Hardware** — the physical machine itself
- **Software** — the tools, programs, applications which run on top of the hardware

Questions:

- What is **software**?
- Where do **programs** come from?
- Programming: **invention**? **construction**?

Computing — hardware, software, ...

Computing:

- **Hardware** — the physical machine itself
- **Software** — the tools, programs, applications which run on top of the hardware

Questions:

- What is **software**?
- Where do **programs** come from?
- Programming: **invention**? **construction**?

Software: where do programs come from

There are a number of misconceptions concerning computer programming, in particular:

- Programming is (very) **difficult**
- Only bright people can **program** a computer
- Programming is sheer "**art**" — you have to be one of the elected few who understand it...

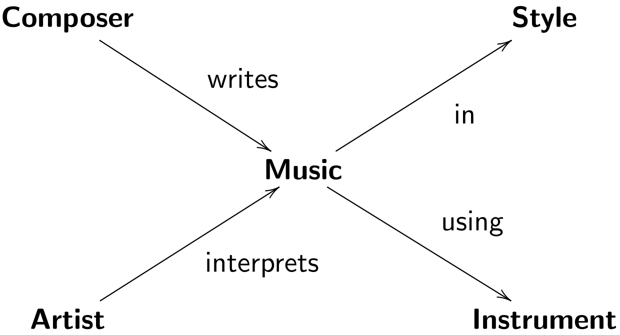
Teaching programming

Nonsense! It turns out that

- Computer **programs** emerge from ordinary mathematics
- Teaching **computer programming** could start around K12, if not earlier, as an activity close to mathematics training

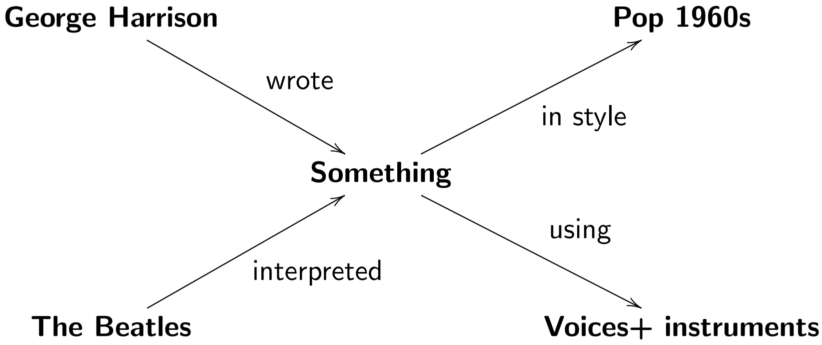
What does it mean to program?

Analogy:



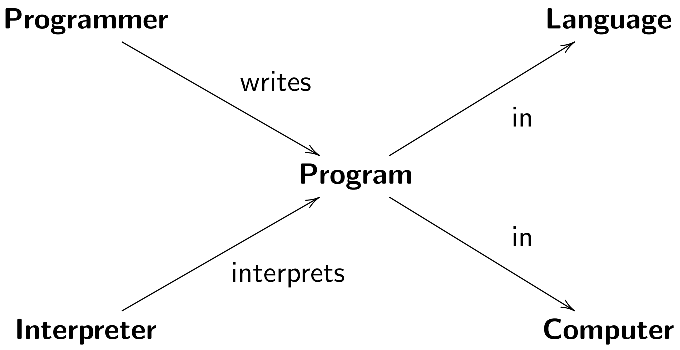
Computer programming

Example:



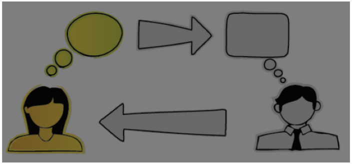
Computer programming

Programming:



Languages?

Humans talk to each other using natural **languages**.



Talking to a **machine** is not very different — we also need "languages".

But such languages need to be **understandable** by machines.

Kinds of Language

We need languages

- to describe **objects**
- to give the machines **instructions** for them to perform **actions** which we regard as useful.

Thus the classification:

- Domain specific languages (**DSL**s) — which describe **objects**, eg. **music**, **text**, **videos**, **web sites** and so on
- **Programming languages** — which instruct machines how to replace humans and perform **actions**.

DSLs

We shall get in touch with the following DSLs/systems:

- **ABC** — for describing music
- **MarkDown** — for describing websites
- **LaTeX** — for describing text
- **OpenShot** — for describing videos

Everything will be web-based: no need for installing anything.

Programming

Concerning programming languages, we will resort to one called

Haskell

available from the **Jupyter** server at DIUM.

Programming

Let us try it!

Numbers

We could see that it all behaves like an ordinary **calculator** as far as **numbers** are concerned.

Is this all Haskell and Jupyter have to offer?

No...

Words in Haskell

- Further to numbers, it 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 more than 200 years ago.

F. Mendelssohn-Bartholdy (1809-1847)

Words in Haskell

- Further to numbers, it 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 more than 200 years ago.

F. Mendelssohn-Bartholdy (1809-1847)

Words in Haskell

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

```
length "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).

Basic operators on words

Word **inversion**:

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

Word **chaining**:

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

or (if you like)

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

Basic operators on words

Word **inversion**:

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

Word **chaining**:

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

or (if you like)

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

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"
```

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"
```

From words to sentences

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"]
```

From words to sentences

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"]
```

From words to sentences

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

$$\mathit{length}\ \texttt{"Mendelssohn died in 1847"} = 24$$

is made of 4 words:

$$\mathit{length}\ (\mathit{words}\ \texttt{"Mendelssohn died in 1847"}) = 4$$

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

`show 1847`

Exercise 1: Check the difference between numbers and words by evaluating the following expressions:

a) `1847 + 2`
b) `"1847" + 2`
c) `"died in " ++ 1847`
d) `"died in " ++ (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 2: Check the difference between numbers and words by evaluating the following expressions:

a) $1847 + 2$
b) `"1847"` $+ 2$
c) `"died in "` $++$ 1847
d) `"died in "` $++$ (*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 3: Check the difference between numbers and words by evaluating the following expressions:

a) `1847 + 2`
b) `"1847" + 2`
c) `"died in " ++ 1847`
d) `"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:

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

F.J. H

$$\textit{inits}\ \texttt{"Haydn"} =$$

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

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

Empty words and empty sentences

Words can have only one character, cf.

length "H" $= 1$

and even no characters at all:

length "" $= 0$

F.J. H

inits "Haydn" $=$

["", "H", "Ha", "Hay", "Hayd", "Haydn"]

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

Empty words and empty sentences

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

$$w ++ "" = "" ++ w = w$$

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

F.J. H

$$inits\ "Haydn" =$$
$$["", "H", "Ha", "Hay", "Hayd", "Haydn"]$$

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

  `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']`

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

$$"F" +\!\!+ \ "Mendelssohn"$$

or

$$'F' : "Mendelssohn"$$

Both yield `"FMendelssohn"`.

The (:) operator is known as *cons* (which stands for "*construct*") and is such that

$$c : w = "c" +\!\!+ w$$

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

$$'H' : ('a' : ('y' : ('d' : ('n' : "")))) = "Haydn"$$

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

`"F" ++ "Mendelssohn"`

or

`'F' : "Mendelssohn"`

Both yield `"FMendelssohn"`.

The `(:)` operator is known as *cons* (which stands for "*construct*") and is such that

`c : w` = `"c" ++ w`

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

`'H' : ('a' : ('y' : ('d' : ('n' : "")))) = "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

  $$\textit{intersperse}\ x\ (\textit{cycle}\ w) = \textit{cycle}\ (\textit{intersperse}\ x\ w)$$

- **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 (≠ '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 (≢ '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 (≠ '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:

$$take\ n\ w +\!\!+\ drop\ n\ w \quad = \quad w \tag{1}$$

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:

$$take\ n\ w +\!\!+\ drop\ n\ w \quad = \quad w \qquad (1)$$

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:

$$take\ n\ w \mathbin{+\!\!+} drop\ n\ w \quad = \quad w \qquad (1)$$

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 4: Check that Haskell knows about the Latin alphabet by running

$$succ\ 'A' = 'B'$$
$$succ\ 'B' = 'C'\ ,\ etc$$
$$pred\ 'k' = 'j'$$
$$pred\ 'd' = 'c'\ ,\ etc$$

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 5: Check that Haskell knows about the Latin alphabet by running

```
succ 'A' = 'B'
succ 'B' = 'C'
```
, *etc*

```
pred 'k' = 'j'
pred 'd' = 'c'
```
, *etc*

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"
```

Word mappings

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?

Rebuilding sentences from their words

Let us try it:

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

So $concat$ merges a sequence of words into a single word.

(It can be thought of (++) generalized to more than two arguments.)

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

```
concat (intersperse " " ["Haydn", "died", "in", "1809"])
```

Rebuilding sentences from their words

Exercise 6: Run `take 16 (cycle "ARBRCRD")`. Conclude that Haskell is able to select from infinite words.

□

Exercise 7: Check that `concat [""]` = `""` but `concat ""` yelds an error. Why is this so?

□

Exercise 8: Does the following **mathematical** property

`concat (intersperse " " (words s))` = `s`

hold? Justify.

□

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

(Generalizing in fact to arbitrary sequences.)

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

$$[\,] +\!\!+ w = w$$

$$[a] +\!\!+ w = a : w$$

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:

$$[\,] +\!\!+ w = w$$

$$[a] +\!\!+ w = a : w$$

$$([a] +\!\!+ y) +\!\!+ z = [a] +\!\!+ (y +\!\!+ z)$$

Programming with words, not numbers

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

$$[\,] +\!\!+ w = w$$

$$[a] +\!\!+ w = a : w$$

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

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

$$[\,] +\!\!+ w = w$$

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

Programming with words, not numbers

Exercise 9: Knowing that properties

$$\textit{length}\ [\,] = 0$$

$$\textit{length}\ (w +\!\!+\ y) = \textit{length}\ w + \textit{length}\ y$$

$$\textit{length}\ [c] = \mathit{1}$$

hold, provide your own version of *length*.

□

Exercises

Exercise 10: From the following properties of $\in$,

$$c \in [\,] = False$$

$$c \in (w +\!\!+ y) = c \in w \vee c \in w$$

$$c \in [d] = c == d$$

provide you own version of this operator.

□

Exercise 11: Complete the following properties of the word reversal operation:

$$reverse\ [\,] = [\,]$$

$$reverse\ (w +\!\!+ y) =$$

.....

$$reverse\ [c] = \ldots.$$

Hence provide your own version of *reverse*.

□

Exercises

Exercise 12: Complete the following properties of the $map\ f$ operator:

$map\ f\ [\,] = [\,]$

$map\ f\ (w +\!\!+\ y) = ...$

$map\ f\ [c] = ...$

Hence provide your own version of $map\ f$.

□

Exercise 13: Generalize all functions written in the exercises above from words to arbitrary sequences.

□

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.*