

Curriculum for the Imperative vs Functional Programming Teaching Experiment

You can find this document in [HTML](#), [PDF](#), and the [original Org](#).

Introduction

The functional paradigm has many advantages over other more common, popular, and "traditional" paradigms, which are more popular to the detriment of the programmers using them, the functional programmers forced to use them, and the world at large, because the functional style is deemed to be complicated, hard to learn, understand, use, and targeted only at academic researchers.

This document will outline the curriculum for an experiment with the end goal of asserting whether the functional paradigm is indeed more difficult to learn, use, or understand, when compared to the imperative/procedural paradigm.

The experiment is somewhat biased towards the functional approach, and after the basic concepts that are common to all paradigms, those concepts that don't actually relate to programming itself, almost all concepts come from the functional world. As such, the imperative style taught will not be idiomatic imperative style, but bent to fit the functional style.

The Languages

Due to the purpose of the experiment, and the target learners, the languages given as alternatives should satisfy these requisites:

- Should be high-level – no knowledge of the inner workings of a computer, of memory, of memory management, etc, should be required to use it effectively – C, C++, and similar are out of the question.
- No program structure required – requiring a program in its entirety to live inside an artificial block is insane! – Java, C#, and similar are out of the question.
- A REPL is a **big** plus! It facilitates and encourages experimentation with the language.

A learner's choice of languages should consist of *at least* one language of each of the paradigms.

Imperative

Python

Possibly the easiest of these three imperative languages, used mostly in science. Has lots of magic.

Lua

Very simple, small, and fast interpreted language, used a lot on game development and as an embedded language. No magic whatsoever.

TODO Go?

The most difficult of the three, but also possibly the fastest, useful for concurrent systems. No magic. The only language of all of the listed in this document with no REPL.

TODO Pascal?

Functional

Haskell

A must due to its type system. Not much magic.

Scheme

Very simple general-purpose language with advanced meta programming, mainly due to its straightforward syntax. No magic whatsoever.

Elixir

Powerful language that runs on the *BEAM* VM, suited for parallel systems, with an actor model for its concurrency model. Brings lots of new features to the Erlang table, including proper modules and modules hierarchy, a decent polymorphic system, meta programming capabilities, and good tooling. Not much magic.

Conclusion

For the purpose of the experiment, Haskell will be required, and as such, it will be used and assumed throughout this text in examples, due to it being strongly typed and so straightforward in this regard.

Programming

Basic types

The types that we will call *basic* are types provided by the base language. These vary between languages, but there's usually a set of *basic types* shared by most programming languages. Here we will cover the most common types that are available to all the languages discussed above.

Atomics

These are so called *atomics* because they cannot usually be *decomposed* in smaller parts. With these alone it's already possible to get a lot out of any programming language.

1. Numbers

Nothing to explain here, other than that in computer programming languages it's common for integer numbers to be a distinct type from the *non-integer* numbers – such as *rational*s, *real*s, and *complex*.

In truth, a computer cannot represent *real* numbers, only approximations. The details aren't important – just keep in mind that when doing number calculations with reals on a computer, if you get unexpected results, it's very likely for this to be the cause.

The reasons to call non-integer numbers reals are that a computer may still be useful to do calculations on (approximate) reals, and that most programming languages don't have exact rational numbers – they just "fake" it.

(a) Integers

Just what one would expect – some examples:

```
0
-1
21
42
```

Some languages provide *unsigned* integer types, i.e., the naturals (including 0), non-negative integers.

(b) Reals

Mostly what one would expect as well:

```
0
1
3.14
-6.28
1.4142135
```

Notation similar to the *scientific notation* is also common on many languages, but the exact notation differs between languages, so we won't detail it too much. However, they usually go something like this: `0.1234e5` is the same as 0.1234×10^5 . Check the official documentation of each language for the exact supported notations.

2. Booleans

Used for logic – **True** and **False**. The exact words or symbols used in each language varies, but these are common enough to be good guesses.

3. Chars?

Characters are values that may represent a letter, a number, a symbol, etc, such as: `'a'`, `'3'`, `'!'`, etc. Exact details vary wildly between languages too, so check the language's official documentation.

The most common notation is the one used above – surrounding the character with a single quote (`'`).

Sequences

These types are *collections* of other types, atomic or not, with an order, and they may be empty.

1. Strings

These are sequences of characters. Examples of strings are `"hello"`, `"0 + 1 = 2"`, `""`.

Although other notations exist, the most common of all is surrounding the characters of the string with double quotes (`"`), like in the examples above.

This poses a problem, because by using double quotes to denote a string, makes it impossible to use double quotes themselves inside the string. To fix this, languages allow programmers to *escape* certain specific characters inside a string, by placing a single backslash (`\`) right before the character that's to be escaped – like this, a string with a single character, the double quote itself: `"\""`; or this: `"And then they said: \"are you gonna escape or not?\""`.

In Haskell, strings are actually just lists of characters, so the following section also applies to strings.

2. Lists/Arrays

These, in a way, can be thought of as a generalization of strings, but for elements other than characters. If a string is a list or array of characters, or something else entirely, depends on the language. However, such details aren't usually matter for concern.

Again, notation varies wildly between languages, but the most common among the languages discussed previously (about half of them) is surrounding the elements with square brackets (`[]`), and separating the elements with commas (`,`).

Some examples follow:

```
[]  
[1, 2, -4]  
['h', 'e', 'l', 'l', 'o']  
[3.14, -6.28, 1.4142135]  
["hello", "there"]
```

3. TODO Tuples?

Conclusion

Now that you know how to create, define, write, read, and understand the basic types, you're ready to get your hands dirty and do something with them.

Basic operations on basic types

Atomics

1. Numbers

(a) Arithmetic

Possibly the thing numbers are most useful for. All (almost) of the arithmetic operations you're already familiar with from mathematics are available, and most basic with familiar names too: `+`, `*`, `-`, `/`. Precedence is also the most common in mathematics: `*` and `/` take precedence over `+` and `-`; but otherwise, operations are applied from left to right. Nonetheless, it's possible to force operation precedence and clarify ambiguities with parentheses (`()`).

```
1 + 1  
21 * 2  
66 / 3  
2 * 2 - 3  
2 * (2 - 3)  
(2 * 2) - 3
```

2. Booleans

(a) Logic

The most basic logical operators from mathematics are also available: *not* (`¬`, `not`), *and* (`∧`, `&&`), *or* (`∨`, `||`). The order of precedence, from the most precedent to the least one is `not`, `&&`, `||`.

```
True || False
not False
True && True
```

A note on implementation details: most programming languages evaluate both arithmetic and logical operators from left to right. However, computers are at essence sequential machines, and therefore cannot compute the value of two expressions simultaneously (a bit of a stretch here). Because of this, and for performance reasons, apart from `not` which is unary, logical operators are *short circuiting* – this is just a fancy way of saying that it'll try to do the least amount of work to get to the resulting value. This *short circuiting* is possible in these two cases:

- `False && B`, which evaluates to `False`
- `True || B`, which evaluates to `True`

It may sound like a small detail, but it's actually an important one. And depending on the language, the operands' order may actually change the program's behavior!

Sequences

1. Indexing

In Haskell, to index a list (consequently strings too) you use the `!!` function – indexes start at 0:

```
[0, 1, 2, 3] !! 2
"hello" !! 4
[[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]] !! 1 !! 1
```

2. Destructuring

Lists are defined as either being empty (`[]`), or having a *head* and a *tail*, where the *head* is an element of the list, and the *tail* is the rest of the list. So, in order to destruct (i.e., separate) a list in its components, you use the creatively named functions `head` and `tail`:

```
head [0, 1, 2, 3]
tail [0, 1, 2, 3]
head "hello"
tail "hello"
head (tail [0, 1, 2, 3])
tail (tail [0, 1, 2, 3])
```

Given that we can *destruct* a list into both its components, we should also be able to *construct* a list given its components – and that's what we'll learn now. You can construct a list with its so-called *constructors*. As mentioned above, a list can be the empty list, or a *head* and a *tail* put

together. So we need a way to create an empty list, and a way to create a list from its *head* and its *tail*.

The empty list is easy, because it is itself – `[]` is the empty list, there's no need to complicate.

And to put a *head* and a *tail* together to form a new list you can use the *cons* operator `(:)`.

```
[]
(:) 1 []
1: []
1: (2: (3: []))
1:2:3: []
[] : []
```

What you saw above with `(:)` is an important Haskell convention to keep in mind. A function (or operator) that's defined (called) as `(fun)` (notice the parentheses) is an *infix* operator, i.e., it's placed in between the operands; while usually, for example with `head` and `tail`, functions are *prefix*, i.e., they're placed before the operands. The most common examples of *infix* operators are the arithmetic operators `((+)`, `(/)`, etc). To turn an *infix* operator into a *prefix* operator, all you have to do is surround the operator with parentheses. So, `(+) 1 2` is the same as `1 + 2`.

There's also a convention to turn *prefix* operators into *infix* operators, which is to surround the operator with backticks (```) – we don't have an example yet, but it goes like this: `op arg1 arg2` is equivalent to `arg1 `op` arg2`. Later on we'll get to see examples of this.

3. Concatenation

The operation that takes two sequences of the same type and "glues" them together is called *concatenation*.

```
[0, 1] ++ [2, 3]
(++) [0, 1] [2, 3]
"hello" ++ " " ++ "world"
```

4. TODO Interpolation?

Order – comparison, equality, etc

Something else that's common in mathematics is comparing or equating things. For example, we can say that $2 < 3$, that $2 + 2 = 4$, that $3 \cdot 3 > 3$, that $1 + 1 \neq 1$, etc. When programming, being able to compare and equate things is also very useful. So here's the table:

Mathematics	Haskell
$<$	<code><</code>
$>$	<code>></code>
\leq	<code><=</code>
\geq	<code>>=</code>
$=$	<code>==</code>
\neq	<code>/=</code>

The reason to use `==` instead of `=` for equality will be clear later.

(Pure) Numerical Functions – $S^n \rightarrow S^m$

Let's start now defining our own functions. A very high-level and hand-wavy way to explain is: whenever you would write $f(x) = \text{expr}$, in Haskell you translate that by writing `f x = expr`. So, for example, to define the *identity* function, $\text{identity}(x) = x$, in Haskell, you just write `identity x = x`. For multivariable functions, you just need to add the parentheses in Haskell: $f(x, y) = x \cdot y$ translates to `f (x, y) = x * y`; $f(x, y) = (y, x)$ translates to `f (x, y) = (y, x)`; $f(x) = (x, x)$ translates to `f x = (x, x)`.

Doubles

```
double x = 2 * x
```

Squares

```
square x = x * x
```

etc

Function composition

Like in mathematics (calculus), it's possible to compose functions to define a new function. The notation is similar, and so are the semantics: $(f \circ g)(x)$ is the same as $f(g(x))$. And in Haskell:

```
-- double_square x = double (square x)
double_square = double . square
```

As subtly implied at the beginning of this section, in the case of multivariable functions, composition also *just works*, as long as the types match.

```
h x = (x, x + 1)
g (x, y) = (x * 3, y * 2, x + y)
f (x, y, z) = x * y + z
k = f . g . h
k 10
```


(Pure) Logical Functions

Before *(Pure) Predicates on Numbers* for background.

This is going to be a packed section, with several important bits. Let's start with *flow control*.

Flow Control

It sometimes may happen that we need or want a function to do different things depending on some condition. Imagine we're defining the *absolute* function, i.e., the function that given a (signed) number always returns a positive number, that is the input number itself, or its symmetric.

$$\text{abs}(x) \begin{cases} -x & \text{if } x < 0 \\ x & \text{otherwise} \end{cases}$$

In Haskell, the most basic statement we have for this is the `if then else`. We could translate the function above to this:

```
abs x = if x < 0
      then -x
      else x
```

```
-- Note that the line breaks aren't necessary; this is also OK:
abs x = if x < 0 then -x else x
```

This is already enough to get everything needing flow control done. However, with more clauses it quickly grows in size:

```
f (x, y, z) = if cond1
              then expr1
              else if cond2
                    then expr2
                    else if cond3
                          then expr3
                          else if cond4
                                then expr4
                                else expr5
```

This is hard to type, and when the expressions span several lines it gets hard to read and understand the code. To remediate this problem, we have *guards*:

```
f (x, y, z) -- Notice that there's no equal sign here!
| cond1 = expr1
| cond2 = expr2
| cond3 = expr3
| cond4 = expr4
| otherwise = expr5
```

The conditions are evaluated one by one, in the order defined; if a condition evaluates to true, then the corresponding expression is evaluated and the result is returned as the function's result; otherwise the next condition is tried. This is exactly like the `if then else` expressions before. Because of that, you should consider the order of the conditions when using guards.

The `otherwise` clause isn't necessary, but if all the different conditions don't correspond to all the possible "states", that is, if it's possible for all of the conditions to be false, then the program will crash if there's no `otherwise` clause.

Next we'll learn about a major Haskell feature, available on many functional programming languages, but not as much in imperative languages.

Pattern Matching and Function Clauses

Pattern matching allows us to *match* values according to patterns. For example, if we were to define arithmetic operators, we should probably add one or more clauses to take care of 0 or 1, because they're usually "special".

$$mul(x, y) \begin{cases} 0 & \text{if } x = 0 \\ 0 & \text{if } y = 0 \\ y & \text{if } x = 1 \\ x & \text{if } y = 1 \\ \text{The common case...} & \text{otherwise} \end{cases}$$

We can already define an equivalent function in Haskell using either `if then else` or guards:

```
mul (x, y)
  | x == 0 = 0
  | y == 0 = 0
  | x == 1 = y
  | y == 1 = x
  -- `undefined` can be used to "make holes" when we don't know
  -- how, or don't want, to define some expression.
  | otherwise = undefined
```

And you might be able to guess that *pattern matching* (together with several function clauses) can be used to define this function even more succinctly:

```
mul (0, y) = 0
mul (x, 0) = 0
mul (1, y) = y
mul (x, 1) = x
mul (x, y) = undefined
```

A function clause is analogous to a guard clause – each one will be tried in order, and the first one to "work" is chosen. Each of the clauses is composed of a *pattern*, and when the function is called, the arguments are matched with the pattern. If they do match, then the corresponding expression is evaluated, and the result is returned as the function's result. Otherwise, the next pattern is tried. Similar to the case of guards, if the arguments don't match any of the patterns, then an error is thrown.

One last tip on pattern matching: if you don't care about a particular value, you can give it the pattern `_`, which will match *anything*, but won't be given a name. Thus, the first two clauses of the `mul` function could be rewritten like this:

```
mul (0, _) = 0
mul (_, 0) = 0
```

Pattern matching isn't limited to numbers, however – you can pattern match on values of any type. And that's what you'll practice next.

NOT

```
myNot True = False
myNot False = True
```

AND

```
myAnd (True, True) = True
myAnd (_, _) = False
```

OR

```
myOr (False, False) = False
myOr (_, _) = True
```

XOR

```
myXor (True, False) = True
myXor (False, True) = True
myXor (_, _) = False
```

etc

(Pure) Predicates on Numbers

Is multiple? – in terms of division

Is even/odd? – in terms of division

etc

(Pure) Predicates on Chars?

Is number?

Is alpha?

Is symbol?

etc

(Pure) Recursive Functions on Numbers

Is even/odd?

Is multiple?

Sum, product, ... – recursive process

Fibonacci – the mathematical definition (recursive process)

Sum, product, ... – iterative process

Fibonacci – iterative process

etc

(Pure) Functions on Sequences

Is empty?

Has member?

Length – recursive and iterative processes

Reverse – recursive and iterative processes

etc

(Pure) Functions over Sequences

With explicit recursion.

Double, add 1, ...

```
double [] = []
```

```
double (h:t) = (2 * h):(double t)
```

```
add1 [] = []  
add1 (h:t) = (h + 1):(add1 t)
```

sum, product, ...

(Pure) Functions over Sequences (Higher-order Functions)

Using higher-order functions.

map – double, add 1, triple, ...

fold – length, reverse, has member?, ...

map in terms of **fold**