Curriculum for the Imperative vs Functional Programming Teaching Experiment

siiky

2021/05/25

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 must consist of *at least* one language of each of the paradigms.

For the purpose of the experiment, Haskell will be required. 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.

Here follow ideas/suggestions for possibly good language options.

Imperative

Python

Possibly the easiest of the listed 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.

Programming

First thing to know about Haskell is that it's strongly typed, which means every value has a type, and the compiler/interpreter enforces these types.

The syntax is similar to notation used in mathematics: an object x of type A is written $x : A$; in Haskell it's written `x :: A`. A function f that given an A calculates a B is written $f : A \to B$; in Haskell it's written `f :: A -> B`. And so on...

In many cases Haskell can infer ("guess") the type of some value, so specifying types is mostly optional. In these initial examples we will keep the types explicit, but later we will ignore them, unless there's good reason not to.

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 usally 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 *rationals*, *reals*, 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 :: Int
   -1 :: Int
   21 :: Int
   42 :: Int
   ```

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 :: Float
1 :: Float
3.14 :: Float
-6.28 :: Float
1.4142135 :: Float
```

Notation similar to the *scientific notation* is also common on many languages, but the exact notation differs between languages. We won't be using it in this document, so we won't detail it here. 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:

   ```
   [] :: [Int]
   [] :: [Float]
   [] :: [Char]
   "" :: [Char]
   [1, 2, -4] :: [Int]
   [1, 2, -4] :: [Float]
   ['h', 'e', 'l', 'l', 'o'] :: [Char]
   ['h', 'e', 'l', 'l', 'o'] :: String
   [3.14, -6.28, 1.4142135] :: [Float]
   ["hello", "there"] :: [[Char]]
   ["hello", "there"] :: [String]
   ```

   Try to understand these type annotations; it will be useful later on.

3. TODO Tuples?

Basic operations on basic types

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.

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 :: Int
21 * 2 :: Int
66 / 3 :: Float -- Int doesn't work; why?
2 * 2 - 3 :: Int
2 * (2 - 3) :: Int
(2 * 2) - 3 :: Int
```

All of the examples above work as well if you specify `Float` as their type.

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 :: Bool
   not False :: Bool
   True && True :: Bool
   ```

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

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] :: Int
   tail [0, 1, 2, 3] :: [Int]
   head "hello" :: Char
   tail "hello" :: String -- [Char]
   head (tail [0, 1, 2, 3]) :: Int
   tail (tail [0, 1, 2, 3]) :: [Int]
   ```

   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 `(:)`.

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

   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 when used; while usually, for example with `head` and `tail`, functions are *prefix*, i.e., they're placed before the operands when used. 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] :: [Int]
   (++) [0, 1] [2, 3] :: [Int]
   "hello" ++ " " ++ "world" :: String
   ```

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
$<$	`<`
$>$	`>`
$\leq$	`<=`
$\geq$	`>=`
$=$	`==`
$\neq$	`/=`

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

(Pure) Numerical Functions – S^n -> S^m

Let's start now defining our own functions. A very high-level and hand-wavy way to explain is: you translate $f(x) = expr$ into Haskell as `f x = expr`. So, for example, to define the *identity* function, $identity(x) = x$, in Haskell, you 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)`; etc.

Doubles

Define a function in Haskell that given an `Int` doubles it.

```
double :: Int -> Int
double x = 2 * x
```

Question for the learner: how would you define a function that given a `Float` doubles it?

Squares

Define a function in Haskell that given an `Int` squares it.

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

Question for the learner: how would you define a function that given a `Float` squares it?

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:

```
-- What if we want Float?
double_square :: Int -> Int
-- 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

-- Define k by expanding the definitions of f, g, and h; i.e., define k with a
-- single expression.
k' = undefined

-- What happens if you change this expression to another type of number?
k 10 :: Int
```

(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:

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

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 its result is returned as the function's result; otherwise the next condition is tried. This exactly like the `if then else` expressions above. 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".

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

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 to, 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 (x, y) = case (x, y) of
    (0, y) -> 0
    (x, 0) -> 0
    (1, y) -> y
    (x, 1) -> x
    (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 consists 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 its result is returned as the function's result. Otherwise, the next pattern is tried. Also

similarly to 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 (x, y) = case (x, y) of
    (0, _) -> 0
    (_, 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.

Just one more thing before we move forward. Another Haskell syntax we can use is *function clauses*. The "full" definition of `mul` above can be rewritten like this:

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

There's usually no good reason to use one over the other, as they are equivalent. This latter definition is more *idiomatic*, but if you prefer using `case`, then `case` is the way to go!

NOT

```
myNot :: Bool -> Bool
myNot True = False
myNot False = True
```

AND

```
myAnd :: (Bool, Bool) -> Bool
myAnd (True, True) = True
myAnd (_, _) = False
```

OR

```
myOr :: (Bool, Bool) -> Bool
myOr (False, False) = False
myOr (_, _) = True
```

XOR

```
myXor :: (Bool, Bool) -> Bool
myXor (True, False) = True
```

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

Haskell Boolean Operators

Haskell already has most of these operations defined. Here's the table:

Math	Haskell
$\wedge$	`&&`
$\vee$	`||`
$\neg$	`not`

Define the following boolean function, first using the `my*` functions defined above, and then using the standard Haskell boolean operators:

$$h : (Bool \times Bool \times Bool) \to (Bool \times Bool)$$
$$h(a, b, c) = ((a \wedge b) \oplus c, a \vee b \vee c)$$

```
h :: (Bool, Bool, Bool) -> (Bool, Bool)
h (a, b, c) = (myXor (myAnd (a, b)) c, myOr a (myOr b c))
h (a, b, c) = (myXor (a && b) c, a || b || c)
```

(Pure) Predicates on Numbers

Is even/odd? – in terms of division

The functions `div`, `mod`, and `divMod` may be useful.

```
-- All of the following definitions are valid.
isEven :: Int -> Bool

isEven n = (a `mod` 2) == 0

isEven n = (a `mod` 2) /= 1

isEven n = case a `mod` 2 of
    0 -> True
    1 -> False

isEven n = not isOdd n

isEven = not . isOdd

-- All of the following definitions are valid.
isOdd :: Int -> Bool
```

```haskell
isOdd n = (a `mod` 2) == 1

isOdd n = (a `mod` 2) /= 0

isOdd n = case a `mod` 2 of
    0 -> False
    1 -> True

isOdd n = not isEven n

isOdd = not . isEven
```

The only pair of definitions that wouldn't work is that of the mutually recursive definitions, that is, the one where each function calls the other. More details about recursive functions will come next.

Is multiple? – in terms of division

The functions `div`, `mod`, and `divMod` may be useful.

```haskell
isMultiple :: Int -> Int -> Bool
isMultiple a b = (a `mod` b) == 0

isMultiple a b = case a `mod` b of
    0 -> True
    _ -> False
```

etc

(Pure) Predicates on Chars?

Is digit?

```haskell
isDigit '0' = True
isDigit '1' = True
isDigit '2' = True
isDigit '3' = True
isDigit '4' = True
isDigit '5' = True
isDigit '6' = True
isDigit '7' = True
isDigit '8' = True
isDigit '9' = True
isDigit _ = False
```

Is whitespace?

```
isWhitespace ' ' = True
isWhitespace '\t' = True
isWhitespace '\v' = True
isWhitespace _ = False
```

Is symbol?

Is alpha?

etc

(Pure) Recursive Functions on Numbers

In short, recursive functions are functions that, directly or indirectly, call themselves. A popular one is that of the Fibonacci number:

$$fib : \mathbb{N}_{\nu} \to \mathbb{N}$$

$$fib(n) = \begin{cases} 1 & \text{if } n < 2 \\ fib(n-1) + fib(n-2) & \text{otherwise} \end{cases}$$

Fibonacci – the mathematical definition (recursive process)

The definition above translates to:

```
fib :: Int -> Int
fib n
    | n < 2 = 1
    | otherwise = fib (n-1) + fib (n-2)
```

Is even/odd?

Here we're going to define `isEven` and `isOdd` in a mutually recursive manner, but neither will use the `mod` function. Instead, we'll say that a number n is even if it 0, or if $n-1$ is odd. Likewise, we'll say that a number n is odd if it is not 0, or if $n-1$ is even.

```
isEven :: Int -> Bool
isEven 0 = True
isEven n = isOdd (n-1)

isOdd :: Int -> Bool
isOdd 0 = False
isOdd n = isEven (n-1)
```

Is multiple?

```
isMultiple :: Int -> Int -> Bool
isMultiple a b
    | a == b = True
    | a > b = isMultiple (a - b) b
    | a < b = False
```

Sum, product, ... – recursive process

Given two numbers a and b such that $a < b$, we'll define the following expressions:

$\sum_{i=a}^{b} i$

$\prod_{i=a}^{b} i$

```
-- What happens if a > b?
sum a b
    | a == b = a
    | a < b = a + sum (a + 1) b

-- What happens if a > b?
prod a b
    | a == b = a
    | a < b = a * prod (a + 1) b
```

Sum, product, ... – iterative process

Because of how the computers work, the definitions of `sum` and `prod` above are very inefficient, and may even not work for a large enough $b - a$. That is because the definitions above evolve into a *recursive process*; while computers are *iterative* machines. To explain it better, let's look at an example `sum` call, and manually evaluate it.

```
sum 0 4
==
0 + sum 1 4
==
0 + (1 + sum 2 4)
==
0 + (1 + (2 + sum 3 4))
==
0 + (1 + (2 + (3 + sum 4 4)))
==
0 + (1 + (2 + (3 + 4)))
==
0 + (1 + (2 + 7))
==
```

```
0 + (1 + 9)
==
0 + 10
==
10
```

It's possible to do better, however. Let's look at the iterative definition, and then see the difference between the two by also manually evaluating it.

```
sum :: Int -> Int -> Int
sum a b = sumAux 0 a b

sum :: Int -> Int -> Int -> Int
sumAux ret a b
    | a == b = ret + a
    | a < b = sumAux (ret + a) (a + 1) b
```

Here's the manual evaluation of `sum 0 4`, according to this new definition:

```
sum 0 4
==
sumAux 0 0 4
==
sumAux (0 + 0) 1 4
==
sumAux 0 1 4
==
sumAux (0 + 1) 2 4
==
sumAux 1 2 4
==
sumAux (1 + 2) 3 4
==
sumAux 3 3 4
==
sumAux (3 + 3) 4 4
==
sumAux 6 4 4
==
6 + 4
==
10
```

Notice that this function results in roughly the same number of steps, but it doesn't grow "to the right", in a triangle, as with the other definition. That is the big difference. If $b - a = n$, then the first definition would grow into a triangle of height $n + 1$, that is, it would evolve eventually into a bunch of "pending" operations. With the second definition, the number of "pending" operations can

be considered constant.

This new definition has a couple more steps, but that can be improved, for example like this:

```
sum :: Int -> Int -> Int
sum a b
    | a == b = a
    | a < b = sumAux a (a + 1) b

sumAux :: Int -> Int -> Int -> Int
sumAux ret a b
    | a == b = ret + a
    | a < b = sumAux (ret + a) (a + 1) b
```

Exercise: show that the two definitions are equivalent.

Fibonacci – iterative process

A similar iterative "conversion" can be applied to the Fibonacci function. It isn't as obvious, but here's the definition:

```
fib n = fibAux 1 1 n

-- n1 corresponds to fib(n-1); and n2 correspends to fib(n-2)
fibAux n2 _ 0 = n2
fibAux n2 n1 s = fibAux n1 (n1 + n2) (s - 1)
```

To try and explain by visualizing it, let's build a diagram. The parentheses surround `n2` and `n1` (i.e., $fib(n-2)$ and $fib(n-1)$).

```
fib(n=2) = 2
(1 1) 2 3 5 8 13

fib(n=3) = 3
1 (1 2) 3 5 8 13

fib(n=4) = 5
1 1 (2 3) 5 8 13

fib(n=5) = 8
1 1 2 (3 5) 8 13

fib(n=6) = 13
1 1 2 3 (5 8) 13
```

Exercise: manually evaluate $fib(n)$, according to the first mathematical definition and this new definition, for a small n ($n < 5$).

etc

(Pure) Functions on Sequences

Is empty?

```
empty :: [a] -> Bool
empty [] = True
empty _ = False
```

Has member?

```
member :: Eq a => a -> [a] -> Bool
member _ [] = False
member x (y:ys) = (x == y) || member x ys
```

Length – recursive and iterative processes

```
length :: [a] -> Int
length [] = 0
length (_:xs) = 1 + length xs
```

This definition evolves into a recursive process, as we've seen before, but it's possible to turn it into one that evolves into an iterative process.

Reverse – recursive and iterative processes

```
reverse [] = []
reverse (h:t) = reverse t ++ [h]
```

This definition too, evolves into a recursive process, but there's a better one evolving into an iterative process.

etc

(Pure) Functions over Sequences (Explicit Recursion)

Compare different definitions with recursive and iterative processes.

Double, add 1, . . .

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

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

Notice that the two functions are essentially the same:

```
func [] = []
func (h:t) = (f h):(func t) -- for some given f
```

For `double`, `f` would be `(*2)`; and for `add1`, `(+1)`.

`sum`, `product`, ...

```
sum :: [Int] -> Int
sum [] = 0
sum (h:t) = h + sum t

product :: [Int] -> Int
product [] = 1
product (h:t) = h * product t
```

Notice that here too, the two functions are essentially the same:

```
func [] = ret -- for some given ret
func (h:t) = h `f` func t -- for some given f
```

For `sum`, `ret` would be 0, and `f` would be `(+)`; and for `product`, `ret` would be 1, and `f` would be `(*)`.

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

We saw in the previous chapter that some computations we want to compute are very similar. In this chapter we'll see that they can indeed be abstracted, that is, be made more general.

`map` – double, add 1, triple, ...

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (h:t) = (f h):(map f t)

double = map (*2)

add1 = map (+1)

triple = map (*3)
```

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

`sum`, `product`

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ ret [] = ret
foldr f ret (h:t) = f h (foldr ret f t)

sum = foldr (+) 0

product = foldr (*) 1
```

```haskell
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl _ ret [] = ret
foldl f ret (h:t) = foldl f (f ret h) t

sum = foldl (+) 0

product = foldl (*) 1
```

`map` in terms of `fold`

On top of these, `map` can also be written as a `fold?`.

```haskell
map f = foldr (\x ret -> (f x) : ret) []
map f = foldl (\ret x -> (f x) : ret) []
```

How do these two definitions differ from each other? How do the definitions for `sum` and `product` using `foldr` and `foldl` differ from each other? How does `foldr` differ from `foldl`? Is one better than the other? Should one be preferred over the other?