HOW TO WRITE GOOD CODE:

START PROJECT.

DO THINGS RIGHT OR DO THEM FAST?

FAST

CODE FAST

DOES IT WORK YET?

NO

ALMOST, BUT IT'S BECOME A MASS OF KLUDGES AND SPAGHETTI CODE.

NO, AND THE REQUIREMENTS HAVE CHANGED.

THROW IT ALL OUT AND START OVER.

GOOD CODE

https://xkcd.com/844/
list is not a type; it is analogous to a mathematical function that takes a type argument and returns a type. Some types are:

- `int list`, which is the type consisting of lists of ints.
- `(int*int) list`, which is the type consisting of lists of pairs of ints, and
- `int list list`, which is the type consisting of lists of lists of ints.

Recall the length procedure:

```
let rec length = function
    [] -> 0
  | _:::rest -> 1 + length rest
```

This single procedure can be applied to all kinds of lists. In Scheme, `length` could be applied to arbitrary lists.

How does it work in Ocaml? Length is a polymorphic procedure.
Parameterized types and polymorphic procedures and type variables

Different kinds of polymorphism in programming languages:

• parametric polymorphism, e.g. `length`

• ad hoc polymorphism, e.g. Scheme’s addition procedure and Ocaml’s `=`

• subtype polymorphism (in object-oriented programming)
Parameterized types and polymorphic procedures and type variables

Simpler example of polymorphic procedure:
let one = function x -> 1

Accepts argument of any type, returns an \texttt{int}
What is the type of this procedure?
Ocaml says \texttt{‘a -> int}
\texttt{‘a} is not a type. It is a type variable.

What is type of procedure \texttt{length}? \texttt{‘a list -> int}
This means the procedure accepts argument that is a list of any type.

Recall \texttt{car} procedure: let car = function x:::_ -> x
Type is \texttt{‘a list -> ‘a}
Procedure can accept a list with any element type, but return type is that element type.
Parameterized types and polymorphic procedures and type variables

Can have multiple type variables in a type.

let swap = function x,y -> y,x

Type is \( \text{'a} \times \text{'b} \rightarrow \text{'b} \times \text{'a} \)

Given a pair consisting of an int and a bool, returns a pair consisting of a bool and an int.

Given a pair consisting of two bools, it returns a pair of bools.

You might think every value would have a concrete type, but empty list has the potential to be part of any list, so its type is \( \text{'a list} \)

Quiz: Guess the type of \([\], [], [true]\)
Answer: \( \text{'a list} \times \text{'b list} \times \text{bool list} \)
containsP is a polymorphic procedure

**input:** pair consisting of $x$ and list

**output:** $true$ if $x$ is in the list

**Quiz:** Write it.

```ocaml
let rec containsP = function
  _, [] -> false
| x, first::rest ->
  if x=first then true else containsP (x, rest)
```

Logic…. use Ocaml’s short-circuiting “or”: ||

```ocaml
let rec containsP = function
  _, [] -> false
| x, first::rest -> x=first || containsP (x, rest)
```

Corresponding “and” is &&
containsP is a polymorphic procedure

**input:** pair consisting of $x$ and list  
**output:** $true$ if $x$ is in the list

**Quiz:** Write it.

```ocaml
definition containsP = function
   _, [] -> false
   | x, first::rest -> if x=first then true else containsP (x, rest)
```

What is the type of `containsP`?  
$x$ can be any type, so use type variable `‘a`  
The list can have any element type …  
but it must agree with $x$  
so type is `‘a list`  
Whole type is (`‘a * ‘a list`) -> `bool`

Logic…. use Ocaml’s short-circuiting “or”: `||`

```ocaml
definition containsP = function
   _, [] -> false
   | x, first::rest -> x=first || containsP (x, rest)
```

Corresponding “and” is `&&`
curried version of `containsP`

**input:** pair consisting of $x$ and list  
**output:** `true` if $x$ is in the list

**input:** $x$  
**output:** procedure that takes in a list and returns `true` if $x$ is in the list

```ocaml
let rec containsP = function
  _, [] -> false
  | x, first::rest -> x=first || containsP (x, rest)
```
curried version of \texttt{containsP}

\textbf{input}: pair consisting of $x$ and list  
\textbf{output}: $true$ if $x$ is in the list

\textbf{input}: $x$  
\textbf{output}: procedure that takes in a list and returns $true$ if $x$ is in the list

Maybe you want a \texttt{find\_doctor} procedure for \texttt{string} lists and a \texttt{find\_17} procedure for \texttt{int} lists.

Instead of writing each one separately, make them:

\begin{verbatim}
# (curried_containsP 17) [1; 2; 3; 4];;
- : bool = false
# let contains_doctor = curried_containsP "doctor";;
val contains_doctor : string list -> bool = <fun>
\end{verbatim}
curried version of containsP

**input:** pair consisting of $x$ and list  
**output:** $true$ if $x$ is in the list

Maybe you want a `find_doctor` procedure for string lists and a `find_17` procedure for int lists.

Instead of writing each one separately, make them:

```ocaml
# (curried_containsP 17) [1; 2; 3; 4];;
- : bool = false
# let contains_doctor = curried_containsP "doctor";;
val contains_doctor : string list -> bool = <fun>
```

```ocaml
let rec curried_containsP = function x ->
  function
    [] -> false
  | first::rest -> x=first || (curried_containsP x) rest
```
map is a polymorphic procedure

```ml
let rec map = function
    f, [] -> []
  | f, a::rest -> f a::map (f, rest)
```

What is the type of the procedure?
It takes in a procedure \( f \) and a list
Type of \( f \) can be anything.
Element type of the list can be anything.
However, they have to agree:
input type of \( f \) must equal element type of the list.
Reflected in type by using same type variable:
\((\text{'a -> 'b}) * \text{'a list} \to \text{'b list}\)
Output of \text{map} is a list whose element type is
output type of \( f \).
Curried map

Want to construct a procedure that specifically added one to each element of list. Therefore want a curried version of `map`.

```ocaml
# let one_adder = curried_map (function x->1+x);;
val one_adder : int list -> int list = <fun>
# one_adder [10; 20; 30];;
- : int list = [11; 21; 31]
```

```ocaml
let rec curried_map = fun f ->
  function
  [] -> []
| a::rest -> f a::curried_map f rest
```

Quiz: What is the type of `curried_map`?
Answer: (`'a -> 'b`) -> `'a list` -> `'b list`