“Functional programming combines the flexibility and power of abstract mathematics with the intuitive clarity of abstract mathematics.”
make_list

**input:** an integer \( n \)

**output:** a list consisting of the integers 1 through \( n \)

Traditional recursion strategy

``` Ocaml
let rec make_list = function 0 -> [] | n -> n::make_list (n-1)
```

Try it out.

**Quiz:** Write a tail-recursive version of `make_list`

``` Ocaml
let rec make_list_helper = function
  (0, lst) -> lst
| (n, lst) -> make_list_helper (n-1,n::lst)

let make_list = function n -> make_list_helper (n, [])
```
tail-recursive reverse

Last time, we wrote this:

```
let rec rev_helper = function
    | [], result -> result
    | first::rest, result -> rev_helper (rest, first::result)

let rec rev = function lst -> rev_helper (lst, [])
```

Example:
```
# rev_helper ([1;2;3], [4;5;6]);;  
- : int list = [3; 2; 1; 4; 5; 6]
```

Quiz: What is the spec for `rev_helper`?

```
input: a pair consisting of a list `lst` and a list `partial`
output: a list consisting of the elements of `lst` in reverse order followed by the elements of `partial`.
```

Is this spec consistent with the code?

- with base case?
  
```
| [], result -> result
```

- with recursive case?
  
```
first::rest, result -> rev_helper (rest, first::result)
```
**input**: a pair consisting of a list \( lst \) and a list \( partial \)

**output**: a list consisting of the elements of \( lst \) in reverse order followed by the elements of \( partial \).

Is this spec consistent with the code?

\[
[], \text{ result } \rightarrow \text{ result}
\]

\[
\text{first::rest, result } \rightarrow \text{ rev_helper (rest, first::result)}
\]

According to spec, value of \( \text{rev_helper (rest, first::result)} \) is elements of \( rest \) in reverse order,
followed by elements of \( first::result \) in forward order

Same as elements of \( first::rest \) in reverse order,
followed by elements of \( result \) in forward order

This is what spec requires.
Quiz: Write a tail-recursive version of append

Hint 1: You don't need an extra argument
Hint 2: result might not come out the way you expect or want.

let rec tail_append_helper = function
  | [], lst -> lst
  | first::rest, lst -> tail_append_helper (rest, first::lst)

How to write append in terms of this helper?

let append = function
  lst1, lst2 -> tail_append_helper (rev lst1, lst2)

Try it out.
Review of algorithm analysis

Big O and big Omega

If there are constants $c$ and $n_0$ such that $f(n) \leq c \cdot g(n)$ for $n \geq n_0$
then we say $f$ is $O(g)$.

Informally, "$f(n)$ is at most a constant times $g(n)$ as $n$ grows" or
"$f(n)$ is eventually at most a constant times $g(n)$.

If there are constants $c$ and $n_0$ such that $f(n) \geq c \cdot g(n)$ for $n \geq n_0$
then we say $f$ is $\Omega(g)$

Informally, "$f(n)$ is at least a constant times $g(n)$ as $n$ grows." or
"$f(n)$ is eventually at least a constant times $g(n)$.

Only makes sense if $n$ is allowed to be arbitrarily large.

"$f(n)$ is $O(g(n))$" same as "$g(n)$ is $\Omega(f(n))$"

If $f(n)$ is $O(g(n))$ and $g(n)$ is $O(f(n))$, we say $f(n)$ is $\Theta(g(n))$