Functions

Exercise 1. Define (implement) and test on a couple of examples functions corresponding to / computing:

1. `c_or` and `c_not`;
2. exponentiation for Church numerals;
3. is-zero predicate for Church numerals;
4. even-number predicate for Church numerals;
5. multiplication for pair-encoded natural numbers;
6. factorial $n!$ for pair-encoded natural numbers.
7. the length of a list (in Church numerals);
8. `cn_max` – maximum of two Church numerals;
9. the depth of a tree (in Church numerals).

Exercise 2. Representing side-effects as an explicitly "passed around" state value, write (higher-order) functions that represent the imperative constructs:

1. `for`...`to`...
2. `for`...`downto`...
3. `while`...`do`...
4. `do`...`while`...
5. `repeat`...`until`...

Rather than writing a λ-term using the encodings that we've learnt, just implement the functions in OCaml / F#, using built-in `int` and `bool` types. You can use `let rec` instead of `fix`.

- For example, in exercise (a), write a function `let rec for_to f beg_i end_i s =`... where `f` takes arguments `i` ranging from `beg_i` to `end_i`, state `s` at given step, and returns state `s` at next step; the `for_to` function returns the state after the last step.
- And in exercise (c), write a function `let rec while_do p f s =`... where both `p` and `f` take state `s` at given step, and if `p s` returns true, then `f s` is computed to obtain state at next step; the `while_do` function returns the state after the last step.

Do not use the imperative features of OCaml and F#, we will not even cover them in this course!

Despite we will not cover them, it is instructive to see the implementation using the imperative features, to better understand what is actually required of a solution to this exercise.

a)
```
let for_to f beg_i end_i s =
  let s = ref s in
  for i = beg_i to end_i do
    s := f i !s
  done;
  !s
```

b)
```
let for_downto f beg_i end_i s =
  let s = ref s in
  for i = beg_i downto end_i do
    s := f i !s
  done;
  !s
```

c)
```
let while_do p f s =
  let s = ref s in
  while p !s do
    s := f !s
  done;
  !s
```

d)
```
let do_while p f s =
  let s = ref (f s) in
  while p !s do
    s := f !s
  done;
  !s
```

e)
```
let repeat_until p f s =
  let s = ref (f s) in
  while not (p !s) do
    s := f !s
  done;
  !s
```