CS 4: Fundamentals Of Computer Programming, Winter 2017

Assignment 2: Asymptotic complexity and higher-order functions

Due: *Friday, January 27, 02:00:00*

Coverage

This assignment covers the material up to lecture 7, corresponding to section 1.3.4 of SICP.

What to hand in

All of your code should be saved to a file named `lab2.ml`. This file should be submitted to csman as usual. Do not submit the interface file (`lab2.mli`), or the test script, or the `Makefile` we supply for you below.

Part @: OCaml notes

Testing

For this assignment, we are supplying you with these support files:

1. a `.mli` OCaml interface file (for this assignment: `lab2.mli`)
2. a test script (for this assignment: `tests_lab2.ml`)
3. a `Makefile`.)

You should download all of these files into the directory in which you are writing your `lab2.ml` code. You should not change them, and you should not submit them as part of your assignment submission to csman.

In order to run the test script, you will need to install one new OCaml packages using the `opam` package manager. Enter the following commands into a terminal (the `$` is a prompt; don't enter that):

```
$ opam update
$ opam install ounit
```

This will install the `ounit` unit testing framework. `ounit` is a set of libraries which make it easy to write unit tests for OCaml code (sort of like the `nose` library for Python).

Once your assignment is done, you should compile it and check that it conforms to the interface file by entering this command:

```
$ make
```

Of course, you can also compile your code from inside the OCaml interpreter using the `#use` directive. Using `#use` is recommended while developing the code, to check that the code has no type errors. Using the `make` command is useful when the code is finished to make sure that your functions *etc.* have the correct type signatures (which they should, if you have been following the directions).

Finally, to run the test script, type this:

```
$ make test
```

This will compile the `tests_lab2.ml` file (which contains the unit tests) and output an executable program called `tests_lab2`. Then it will run that program and report all test failures. If the test program doesn't compile, you probably forgot to install the `ounit` library; see above. If you did and it still doesn't work, see your TA.

If you want to compile the code and immediately run the test script (useful during development/debugging), type:

```
$ make all
```

Running the tests generates some log files; to get rid of them (as well as all compiled OCaml files), type:

```
$ make clean
```

It's worthwhile taking a look at the code in the `tests_lab2.ml` file, even though you don't have to change it. Most of the tests are very straightforward; they use a function called `assert_equal` to test that a particular function call gives a particular result. There are some interesting operators in the file (such as `>:::` and `>::`) which are defined in the `ounit` libraries; one of the cool things about OCaml is that you can define new operators!

Finally, be aware that the test scripts are in no way exhaustive! Some functions are just inherently hard to test, or else hard to test in a way that wouldn't give away the answer to a student who looked at the test script code. We recommend that you experiment with the code on your own as well as run the test script. Don't assume that just because your code passes all the tests that everything is perfect!

Using libraries in OCaml

Some of the problems below require the use of arbitrary-precision integers and rational numbers. The easiest way to get this in OCaml is by using the `Num` library. This is a library which is part of the OCaml standard libraries, so you don't have to install any extra packages to use it. Here we will show you how to use OCaml libraries in two different ways: in the interactive interpreter (useful for debugging and testing code) and while compiling code. We'll use the `Num` library as an example, but the information will apply to any OCaml library.

In the interactive interpreter

The most basic way to use an OCaml library inside the interactive interpreter is to use the `#load` command. This requires that you know the full name of the compiled library file. This means that you have to enter the following commands before using the `Num` library interactively:

```
# #load "nums.cma";;
# open Num;;
```

(In all the interactive OCaml examples, the first `#` is the interactive prompt that you shouldn't type.)

Here, we have to know that the library file is called `"nums.cma"`, which is a bit annoying. The line `open Num;;` brings all the contents of the `Num` package into the top-level namespace, so you don't have to use the `Num` prefix when using any of the functions in the `Num` library. (This is like entering `from Num import *` in Python.)

If the library file is in a non-standard location, it's even worse; you have to add this line before the `#load` line:

```
#directory /path/to/the/library/directory
```

(where you should substitute `/path/to/the/library/directory` with the real path) or else start `ocaml` with the `-I` argument:

```
$ ocaml -I /path/to/the/library/directory
```

Either way, it's a nuisance.

A better way to use a library in the interactive OCaml interpreter is to use the `topfind` tool, which is part of the `ocamlfind` package. This package comes pre-installed on the course VM; you can check for it by typing:

```
$ opam list
```

and seeing if `ocamlfind` is listed. If not, type:

```
$ opam install ocamlfind
```

To use `topfind`, get into the OCaml interpreter and enter the following lines:

```
# #use "topfind";;
# #require "num";;
# open Num;;
```

The first line (`#use "topfind";;`) only needs to be done once per session. After you enter this line, the following lines will be printed out:

```
- : unit = ()
Findlib has been successfully loaded. Additional directives:
  #require "package";;      to load a package
  #list;;                   to list the available packages
  #camlp4o;;                to load camlp4 (standard syntax)
  #camlp4r;;                to load camlp4 (revised syntax)
  #predicates "p,q,...";;   to set these predicates
  Topfind.reset();;         to force that packages will be reloaded
  #thread;;                 to enable threads
```

When this is done, you get a bunch of new commands you can use in the interactive interpreter. Try this one:

```
# #list;;
```

It will print a list of the installed packages, which can be quite useful. One of these packages should be `"num"`. The second line we told you to execute (`#require "num";;`) loads that package. Note that you don't have to know the full name of the compiled library file or its location; the `topfind` tool takes care of that for you. The last line (`open Num;;`) works as before.

We'd like to reiterate here that all the `#`-commands like `#use`, `#require`, `#list` *etc.* are *not* part of the OCaml language but are specific to the interactive interpreter. Please do not use these commands in compiled OCaml code (*i.e.* `.ml` and `.mli` files)!

The only disadvantage of using `topfind` is that you have to enter the line `#use "topfind";;` every time you start the OCaml interpreter. However, we can easily automate this by adding that line to the end of a file called `.ocamlinit`. This file is automatically loaded whenever the `ocaml` interpreter is started up. If you have followed all the setup instructions carefully, there should already be a file called `.ocamlinit` in your home directory. It should contain the following lines:

```
(* Added by OPAM. *)
let () =
  try Topdirs.dir_directory (Sys.getenv "OCAML_TOPLEVEL_PATH")
```

```
    with Not_found -> ()
  ;;
```

If this isn't the case, you should probably see a TA. Otherwise, load this file into your text editor and add these lines:

```
Sys.interactive := false;;
#use "topfind";;
Sys.interactive := true;;
```

at the end. Now, every time you launch the `ocaml` interpreter, `topfind` will automatically be loaded. Sweet! The `Sys.interactive` lines are only there to suppress the messages that the `#use "topfind";;` command would normally print. Make sure you type `:=` and not `=` in those lines! We will explain why later on in the course.

Note also that we seemingly violated the rule we gave above and entered `#`-commands directly into a file. That's because the `.ocamlinit` file is not a compiled `.ml` or `.mli` file; it's just loaded directly into the OCaml interpreter.

The `ocaml` program looks for the `.ocamlinit` file in the current directory first and then in your home directory, so don't have a `.ocamlinit` file in your current directory unless you want to override the one in your home directory.

For the rest of this course we will almost always use `#require` instead of `#load` to load libraries into the interactive interpreter.

To reiterate: once you've set up the `.ocamlinit` file the way we've described above, you can load the `Num` library into an interactive `ocaml` session by typing these lines after starting the `ocaml` interpreter:

```
# #require "num";;
# open Num;;
```

In other words, you now no longer have to type in:

```
# #use "topfind";;
```

when you start the interactive OCaml interpreter.

In compiled code

If you want to compile an executable program that uses the `Num` library, there are two ways: the hard way and the easy way. The hard way is to specify the library information explicitly when you compile your code. For instance, if you have a file `"myprog.ml"` that uses the `Num` library, you can compile it like this:

```
$ ocamlc nums.cma myprog.mli myprog.ml -o myprog
```

Note again that you have to specify the library name in full. If the directory where `nums.cma` is located is not a standard location (*i.e.* not `/usr/lib/ocaml`), you would additionally have to specify the directory with the `-I` option, *e.g.*:

```
$ ocamlc -I /path/to/the/library/directory nums.cma myprog.mli myprog.ml -o myprog
```

Clearly, this is a pain, and perhaps unsurprisingly, the easy way to do this involves using the `ocamlfind` package again, just as in the interactive case. In this case we will use the `ocamlfind` program (which is part of the `ocamlfind` package; don't confuse the two distinct uses of `ocamlfind`):

```
$ ocamlfind ocamlc -package num myprog.mli myprog.ml -o myprog -linkpkg
```

You can also use `ocamlfind` for compiling non-executables (*e.g.* object code files or libraries); in that case you leave off the `-linkpkg` argument and adjust the `ocamlc` arguments as necessary. For instance, if we were compiling a library `mylib.ml` that used the `Num` library we would compile it like this:

```
$ ocamlfind ocamlc -c  -package num mylib.mli mylib.ml
```

We will be using `ocamlfind` for compiling code which uses libraries from now on. Most of the time you won't have to worry about this because the commands will be in the `Makefile` that we will supply to you.

Using the `Num` library

The `Num` library defines a data type called `num` which can represent arbitrarily-large integers and rational numbers composed of arbitrarily-large integers. The library documentation is here. The `num` type is abstract, so you have to convert *e.g.* integers to and from `num`s in order to use them. To convert an integer to a `num`, use the `int_of_num` function and to convert back use the `num_of_int` function. The `string_of_num` function converts a `num` to a string, while `float_of_num` converts a `num` to a floating-point number. The numeric operators in the `Num` library all have a `/` suffix, so instead of `+ - * /` *etc.* we have `+/ -/ */ //` *etc.*. This is analogous to floating-point numbers which use `+. -. *. /.` *etc.*. It's not terribly pleasant to use but it works. Note that you can use the relational operators `< <= > >= = <>` on any two numeric values as long as they have the same type (whether they are `int`s, `float`s, or `num`s). In other words, relational operators are overloaded but arithmetic ones are not.

Here are some examples of the `Num` library in use in the interactive interpreter:

```
$ ocaml
# #require "num";;
# open Num;;
# let n1 = num_of_int 42;;
val n1 : Num.num = <num 42>
# let n2 = num_of_int 57;;
val n2 : Num.num = <num 57>
# n1 +/ n2;;
- : Num.num = <num 99>
# n1 */ n2;;
- : Num.num = <num 2394>
# n1 // n2;;
- : Num.num = <num 14/19>
# float_of_num (n1 // n2);;
- : float = 0.736842105263
# string_of_num (n1 // n2);;
- : string = "14/19"
# int_of_num (n1 // n2);;
Exception: Failure "integer argument required".
# int_of_num ((num_of_int 42) // (num_of_int 21));;
- : int = 2
```

This should be enough to get you through this assignment.

Part A: Orders of growth

As usual, write essay-type answers as OCaml comments.

1. [**20**] Consider the tree-recursive fibonacci function discussed in class:

```
let rec fib n =
  if n < 2
     then n
     else fib (n - 1) + fib (n - 2)
```

You know that the time complexity for this function is `O(2^n)`. (Specifically, it is `Θ(g^n)`, where `g` is the golden ratio (`1.618...`).) If we assume that OCaml is using applicative-order evaluation (the normal OCaml evaluation rule), then what is its *space* complexity? Explain why this is different from the time complexity. *Hint:* consider what the largest number of pending operations would have to be when evaluating `fib 7`. You may assume that once an expression is fully evaluated, all the memory used in evaluating that expression is returned to the system.

2. [**30**] SICP, exercise 1.15

The sine of an angle (specified in radians) can be computed by making use of the approximation `sin x = x` if `x` is sufficiently small, and the trigonometric identity:

```
sin x = 3 sin(x/3) - 4 sin^3(x/3)
```

to reduce the size of the argument of `sin`. (For purposes of this exercise an angle is considered "sufficiently small" if its magnitude is less than 0.1 radians.) These ideas are incorporated in the following functions (using floating-point arithmetic throughout):

```
let cube x = x *. x *. x
let p x = 3.0 *. x -. 4.0 *. cube x
let rec sine angle =
  if abs_float angle < 0.1
     then angle
     else p (sine (angle /. 3.0))
```

a. How many times is the function `p` applied when `sine 12.15` is evaluated?

b. What is the order of growth in space and number of steps (as a function of `a`) used by the process generated by the `sine` function when `sine a` is evaluated?

By "growth in number of steps", we mean the time complexity of the `sine` function as a function of the size of the input. Explain the reason for the space/time complexities; don't just state an answer.

3. [**30**] SICP, exercise 1.16

SICP describes a non-iterative function called `fast_expt` that does exponentiation using successive squaring (when possible). Translated into OCaml, that function looks like this:

```
let rec fast_expt b n =
  let is_even m = m mod 2 = 0 in
  let square m = m * m in
    if n = 0 then 1
    else if is_even n then square (fast_expt b (n / 2))
    else b * fast_expt b (n - 1)
```

Note that `mod` is a predefined infix operator in OCaml (not a function!) which computes remainders; you use it like this: `5 mod 2` (which will return `1`).

1. This function uses nested `if`/`then`/`else` forms, which are a bit ugly and error-prone (since OCaml doesn't actually have an `else if` syntax). Rewrite the function using pattern matching on the `n` argument (a `match` expression); use `when` clauses in pattern matches when you need to test for non-structural conditions. Your function should not have any `if` expressions.

   Here is a skeleton version of the function you should write:

```
let rec fast_expt b n =
  let is_even m = m mod 2 = 0 in
  let square m = m * m in
```

```
match n with
  (* fill in the rest here *)
```

N.B. the "wildcard" pattern _ may be useful to you.

2. Write a function called `ifast_expt` that evolves an *iterative* exponentiation process that uses successive squaring and uses a logarithmic number of steps.

   Hint: Using the observation that $b^n = (b^{n/2})^2 = (b^2)^{n/2}$, keep, along with the exponent `n` and the base `b`, an additional state variable `a`, and define the state transformation in such a way that the product $a * b^n$ is unchanged from state to state. At the beginning of the process `a` is taken to be `1`, and the answer is given by the value of `a` at the end of the process. In general, the technique of defining an invariant quantity that remains unchanged from state to state is a powerful way to think about the design of iterative algorithms.

   Use integer arithmetic for this problem. You may assume that all the (integer) arguments to your function are non-negative.

   You will need some helper functions in the implementation of your `ifast_expt` function. You should make these internal to your `ifast_expt` function, as was done with the `fast_expt` function above. One of these will need to be recursive; call it `iter`. Only use `let rec` with that function; use `let` for all other internal definitions and for the `ifast_expr` function as a whole.

   Use pattern matching instead of nested `if`/`then`/`else` forms as you did for the `fast_expt` function.

4. [**15**] SICP, exercise 1.17

The exponentiation algorithms in this section are based on performing exponentiation by means of repeated multiplication. The simplest such function is this:

```
let rec expt a b =
  if b = 0
     then 1
     else a * expt a (b - 1)
```

In a similar way, one can perform integer multiplication by means of repeated addition. The following multiplication function (in which it is assumed that our language can only add, not multiply), is analogous to the `expt` function:

```
let rec mult a b =
  if b = 0
     then 0
     else a + mult a (b - 1)
```

This algorithm takes a number of steps that is linear in `b`. Now suppose we include, together with addition, the operations `double`, which doubles an integer, and `halve`, which divides an (even) integer by 2. Using these, design a multiplication function analogous to `fast_expt` that uses a logarithmic number of steps.

Use integer arithmetic for this problem. The multiplication function you write should generate a recursive process.

For this problem, write all helper functions (including `double` and `halve`) inside the `fast_mult` function, and again use pattern matching on the `n` argument instead of nested `if`/`then`/`else` expressions.

5. [**15**] SICP, exercise 1.18

Using the results of the previous exercises, devise a function called `ifast_mult` that generates an iterative process for multiplying two integers in terms of adding, doubling, and halving and uses a logarithmic number of steps.

This multiplication function should generate an iterative process. If you are multiplying integer `b` by integer `n`, you will need another state variable `a` such that the invariant is `(a + b*n)`, and `n` will decrease to zero, at which point `a` will be the answer.

Again, use pattern matching instead of nested `if`/`then`/`else` expressions. And again, only use `let rec` where it's absolutely necessary.

6. [**30**] Consider the following (higher-order) function:

```
let rec foo f n =
  if n <= 1
     then f 0
     else foo f (n / 2) + foo f (n / 2)
```

Note that function calls have the highest precedence in OCaml, so the last expression is the same as `(foo f (n / 2)) + (foo f (n / 2))`.

If we assume that the function `f` can compute its result in constant time and constant space, what are the (worst-case) time and space complexities of the function `foo`? Justify your answer. Assume that the integer input `n` is always non-negative, and assume the usual applicative-order evaluation rule.

7. [**15**] Consider this function to compute fibonacci numbers:

```
let fib n =
  let rec last_two n =
    if n < 1
      then (0, 1)
      else
        let p0, p1 = last_two (n - 1) in
          (p1, p0 + p1)
  in
    fst (last_two n)
```

A couple of OCaml notes:

- It's legal to assign more than one value at a time in a `let` expression as shown above. Effectively you are doing a pattern match that cannot fail.
- `fst` is a function which extracts the first value of a two-tuple.

Please answer the following two questions in OCaml comments:

1. What kind of process does this function represent (linear recursive, linear iterative, tree recursive *etc.*) and why?
2. What is the space and time complexity of this function with respect to its argument `n`?

Part B: Evaluation

In this section, write all essay-question-type answers inside OCaml comments.

1. [**30**] Desugar the following (nonrecursive) `let` expressions to the equivalent `fun` expressions applied to arguments. You do not need to evaluate the resulting `fun` expressions. Use the OCaml interpreter to test

that your desugared versions are equivalent to the original versions.

Note: A non-recursive `let`/`and` form binds multiple values to the result of evaluating the corresponding expressions, but none of the expressions can depend on the bindings. (In a recursive `let`/`and` form, any or all of the expressions can depend on the bindings.) A non-recursive `let`/`and` form is therefore equivalent to a function of more than one argument applied to its arguments.

a.
```
let x = 20
and y = 2 * 4
in x * (2 + y)
```

b.
```
let a = 1.0
and b = 20.0
and c = 3.0
in sqrt (b *. b -. 4.0 *. a *. c)
```

c. For this problem, desugar all of the `let` expressions. Note that successive `let`/`in` forms are *not* the same as a `let`/`and` form with multiple `and`s, because expressions can depend on earlier bindings.

```
let x = 1 in
let y = 2 in
let z = 3 in
  x * y * z
```

d. For this problem, desugar all of the `let` expressions.

```
let x = 1 in
let x = 2 in
let x = 3 in
  x * x * x
```

2. [**30**] Using the substitution model (including desugaring `let` to `fun`), evaluate the following expression. You may skip obvious steps (for instance, you can reduce `2 + 2` to `4` in a single step). *Hint:* Desugar all the `let`s to `fun`s before doing anything else. Our evaluation took about 35 lines. Watch out for shielding!

```
let x = 2 * 10
and y = 3 + 4
in
  let y = 14 in
    let z = 22 in
      x * y * z
```

3. [**20**] Ben Bitdiddle can't understand why the following code gives an error:

```
let x = 10
and y = x * 2
and z = y + 3
in x + y + z
```

When Ben runs this (expecting the result to be 53), OCaml complains about `x` being an unbound value. "That's not true!" cries Ben angrily, "`x` was bound on the first line!". Explain why this doesn't work by first desugaring the `let` to a `fun`, and then explain in words why it can't work, by referring to the way expressions get evaluated (you don't need to evaluate the expression explicitly). Then show Ben a simple way to fix this code to make it do what he wants.

Part C: Higher-order functions

In this section, use `num` as the numeric type for all operations unless otherwise indicated. For convenience, put this at the top of the file:

```
open Num
```

and define this helper function:

```
let ni = num_of_int     (* convert int -> num *)
```

We will use `num` as the numeric type when we want either or both of arbitrarily large integers and rational numbers. Note that doing arithmetic on `num`s requires the use of the special `num` operators `+/`, `-/`, `*/`, `//` *etc.* (but the usual equality and relational operators still work).

1. [**10**] [SICP, exercise 1.30]

   The following `sum` function generates a linear recursion:

   ```
   let rec sum term a next b =
     if a > b
       then (ni 0)
       else term a +/ (sum term (next a) next b)
   ```

   (Recall that the `+/` operator is addition of `num`s.) `term` is a function of one argument which generates the current term in a sequence given a sequence value, while `next` is a function of one argument which generates the next value in the sequence. For instance:

   ```
   sum (fun x -> x */ x) (ni 1) (fun n -> n +/ (ni 1)) (ni 10)
   ```

   will compute the sum of all squares of the numbers 1 through 10 (expressed as `num`s).

   The function can be rewritten so that the sum is performed iteratively. Show how to do this by filling in the missing expressions in the following definition:

   ```
   let isum term a next b =
     let rec iter a result =
       if <??>
         then <??>
         else iter <??> <??>
     in
       iter <??> <??>
   ```

   Assume that `term` is a function of type `num -> num`.

   Examples:

   ```
   # let square n = n */ n ;;
   val square : Num.num -> Num.num = <fun>
   # let step1 n = n +/ (ni 1) ;;
   val step1 : Num.num -> Num.num = <fun>
   # isum square (ni 10) step1 (ni 0);;
   - : Num.num = <num 0>
   # isum square (ni 4) step1 (ni 4);;
   - : Num.num = <num 16>
   # isum square (ni 0) step1 (ni 10);;
   - : Num.num = <num 385>
   ```

2. [**30**] [SICP, exercise 1.31]

   a. The `sum` function is only the simplest of a vast number of similar abstractions that can be captured as higher-order functions. Write an analogous function called `product` that returns the product of the values of a function at points over a given range. Show how to define `factorial` in terms of product. Also use `product` to compute approximations to π (3.1415926...) using the formula.

```
         2 * 4 * 4 * 6 * 6 * 8
pi/4 =  --------------------- ...
         3 * 3 * 5 * 5 * 7 * 7
```

b. If your product function generates a recursive process, write one that generates an iterative process. If it generates an iterative process, write one that generates a recursive process.

Call your recursive `product` function `product_rec` and your iterative one `product_iter`. Define a version of `factorial` using both forms of `product`, calling one `factorial_rec` and the other `factorial_iter`.

Examples:

```
# factorial_rec (ni 0)
- : Num.num = <num 1>
# factorial_iter (ni 0)
- : Num.num = <num 1>
# factorial_rec (ni 10)
- : Num.num = <num 3628800>
# factorial_iter (ni 10)
- : Num.num = <num 3628800>
```

Also write the code to generate an approximation to pi (using either the recursive or iterative version of `product`) by filling in the following definitions using the formula described above. Use at least 1000 terms from the product.

```
let pi_product n = <??>   (* infinite product expansion up to n terms *)
let pi_approx = <??>      (* defined in terms of pi_product *)
```

Use `num` as the numeric type for all operations except for the `pi_approx` value, which should be a `float`. Use the `float_of_num` function to convert from a rational approximation to pi (obtained by the formula given above) to a `float`. Note that we're using `nums` in this case because we want arbitrarily-precise rational numbers. Be careful to use `num` operators throughout!

Note that none of these functions need to be more than a few lines long. (Our longest function for this problem is 7 lines long.)

3. [**30**] SICP, exercise 1.32

a. Show that `sum` and `product` from the previous problems are both special cases of a still more general notion called `accumulate` that combines a collection of terms, using some general accumulation function:

```
accumulate combiner null_value term a next b
```

`accumulate` takes as arguments the same term and range specifications as `sum` and `product`, together with a combiner function (of two arguments) that specifies how the current term is to be combined with the accumulation of the preceding terms and a `null_value` that specifies what base value to use when the terms run out. Write `accumulate` and show how `sum` and `product` can both be defined as simple calls to `accumulate`. Assume that all numeric types are `num`.

b. If your `accumulate` function generates a recursive process, write one that generates an iterative process. If it generates an iterative process, write one that generates a recursive process.

Call the recursive `accumulate` function `accumulate_rec` and the iterative version `accumulate_iter`. You can use either form to define `sum` and `product`. Note that in order to use an operator as a function, you must wrap it in parentheses (this is useful when passing an operator as an argument to a function). If the operator name starts with an asterisk, you have to put a space between it and the open parenthesis so OCaml doesn't mistake it for a comment! In other words, write `( */ )` instead of `(*/)`.

4. [**10**] SICP, exercise 1.42

   Let `f` and `g` be two one-argument functions. The *composition* `f` after `g` is defined to be the function `x -> f(g(x))`. Define a function `compose` that implements composition.

   In the examples below, we use `int` instead of `num` as the numeric type. The type of `compose` doesn't depend on what numeric type we use.

   Examples:

```
# let square n = n * n;;
# let inc n = n + 1;;
# (compose square inc) 6
- : int = 49
# (compose inc square) 6
- : int = 37
```

5. [**20**] SICP, exercise 1.43.

   If `f` is a numerical function and `n` is a positive integer, then we can form the `n`th repeated application of `f`, which is defined to be the function whose value at `x` is `f(f(...(f(x))...))`. For example, if `f` is the function `x -> x + 1`, then the `n`th repeated application of `f` is the function `x -> x + n`. If `f` is the operation of squaring a number, then the `n`th repeated application of `f` is the function that raises its argument to the 2^nth power. Write a function that takes as inputs a function that computes `f` and a positive integer `n` and returns the function that computes the `n`th repeated application of `f`. Your function should be able to be used as follows:

```
# (repeated square 2) 5
- : int = 625
```

   Hint: You may find it convenient to use `compose` from the previous exercise.

   In the examples below, we use `int` instead of `num` as the numeric type. The type of `repeated` doesn't depend on what numeric type we use.

   Examples:

```
# let square n = n * n;;
# (repeated square 0) 6
- : int = 6
# (repeated square 1) 6
- : int = 36
# (repeated square 2) 6
- : int = 1296
```

   Note that a function repeated `0` times is the identity function. If you do this right, the solution will be very short.

6. [**25**] SICP, exercise 1.44

   The idea of *smoothing* a function is an important concept in signal processing. If `f` is a function of one (numerical) argument and `dx` is some small number, then the smoothed version of `f` is the function whose value at a point `x` is the average of `f(x - dx)`, `f(x)`, and `f(x + dx)`. Write a function `smooth` that takes as input a function that computes `f` and a `dx` value and returns a function (of one numerical argument) that computes the smoothed `f`. It is sometimes valuable to repeatedly smooth a function (that is, smooth the smoothed function, and so on) to obtained the `n`-fold smoothed function. Show how to generate the `n`-fold smoothed function of any given function using `smooth` and the `repeated` function you defined in the previous problem.

For this problem, we use `float` instead of `num` as the numeric type.

In case the instructions aren't clear, you need to write two functions for this problem: `smooth` and `nsmoothed`, where `nsmoothed` represents the repeated application of the `smooth` function `n` times. Both functions have a `dx` argument representing the step size.

Hint: Be careful with the `dx` argument to `nsmoothed`! You may find it useful to define a helper function called `smooth-dx` which takes in a single function `f` and returns a smoothed function which uses a particular `dx` value.

Note that both `smooth` and `nsmoothed` are very short if you write them the right way.

Part D: Additional problems

In this section, we'll write some functions that were referred to in the lectures but not defined.

1. [**20**] In lecture 5, we referred to a function called `is_prime` which returns `true` if an integer is a prime number and `false` otherwise. Write this function. For the purposes of the function, consider all negative numbers, zero, and 1 to be non-prime (and thus return `false` instead of *e.g.* raising an exception). Your function doesn't have to be maximally efficient, but try to give it a time complexity of at most Θ(`sqrt n`) for input `n`. Note that `sqrt` works on `float`s only; you may find the functions `float_of_int` and `int_of_float` to be useful.

2. [**10**] In lecture 5, we also referred to a function called `smallest_prime_factor` which returns the smallest prime factor of a composite (non-prime) positive integer. Write this function. (You can use the `is_prime` function you defined above in the definition.) If the input number is prime or less than 2, raise an exception using `invalid_arg`. Again, we're not worried about maximal efficiency here, but your function must work properly.
