Part C. Higher-order functions (30 points)

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 (really just an alias):

```
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.* The usual relational operators (`<` , `<=` , `=` *etc.*) will work if used on integers, but don't work with rational numbers, so use the `num` equivalents `</` , `<=/` , `=/` in these cases.

1. (SICP exercise 1.30)

[5 points]

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 comparison of `num` s and 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. (SICP exercise 1.31)

[5 points]

1. 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:

$$\frac{\pi}{4} = \frac{2 \cdot 4 \cdot 4 \cdot 6 \cdot 6 \cdot 8}{3 \cdot 3 \cdot 5 \cdot 5 \cdot 7 \cdot 7}$$

2. 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 π (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 *)
```

Tip

For the purposes of this problem, consider a "single term" of the π approximation to be two consecutive numerator numbers and two consecutive denominator numbers. So the first "term" would be

$$\frac{2 \cdot 4}{3 \cdot 3}$$

the second "term" would be

$$\frac{4 \cdot 6}{5 \cdot 5}$$

etc. Multiplying all the terms together gives the desired approximation to π.

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 `num`s 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. (SICP exercise 1.32)

[**5 points**]

1. Show that `sum` and `product` from the previous problems are both special cases of a still more general notion called `accumulate` [1] 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`.

2. 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. (SICP exercise 1.42)

[**5 points**]

Let `f` and `g` be two one-argument functions. The *composition* of function f after g (often written $f \circ g$) is defined to be the function $x \mapsto f(g(x))$. 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. (SICP exercise 1.43)

[5 points]

If f is a numerical function and n is a positive integer, then we can form the nth repeated application of f, which is defined to be the function whose value at x is $f(f(\ldots(f(x))\ldots))$ (with n fs).

For example, if f is the function $x \mapsto x + 1$, then the nth repeated application of f is the function $x \mapsto x + n$. If f is the operation of squaring a number, then the nth repeated application of f is the function that raises its argument to the $(2n)$th 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 nth repeated application of f. Your function should be able to be used as follows:

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

Tip

You will find it convenient to use `compose` from the previous exercise in your definition of `repeated`.

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. (SICP exercise 1.44)

[5 points]

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 `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. Call this second function `nsmoothed`.

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

Hint

Be careful with the `dx` argument to `nsmoothed`! It's actually more convenient to have `smooth` take the `dx` argument as its *first* argument, because you may need to partially apply `smooth` to `dx` in the definition of `nsmoothed` (at least, that's one way to do it).

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

Here are some examples. Note that your results may not be identical; floating-point math is notoriously hard to reproduce between computers. However, your results should be pretty close to these.

Examples

```
(* smooth examples *)

(* smoothed sin function *)
# let ssin = smooth 0.1 sin;;
val ssin : float -> float = <fun>
# sin 0.0;;
- : float = 0.
# ssin 0.0;;
- : float = 0.
# sin 0.1;;
- : float = 0.0998334166468281548
# ssin 0.1;;
- : float = 0.0995009158139631283
# sin 1.0;;
- : float = 0.841470984807896505
# ssin 1.0;;
- : float = 0.838668418165605112
# sin 3.0;;
- : float = 0.141120008059867214
# ssin 3.0;;
- : float = 0.14064999990238003
# let pi = 4.0 *. atan 1.0;;
# sin (pi /. 2.0);;
- : float = 1.
# ssin (pi /. 2.0);;
- : float = 0.996669443518683806

(* nsmoothed examples *)

# let nssin n = nsmoothed 0.1 sin n
val nssin : int -> float -> float = <fun>

(* ssin0 is the same as sin *)
# let ssin0 = nssin 0;;
val ssin0 : float -> float = <fun>
# ssin0 (pi /. 2.0);;
- : float = 1.
# sin (pi /. 2.0);;
- : float = 1.
# ssin0 1.0;;
- : float = 0.841470984807896505
# sin 1.0;;
- : float = 0.841470984807896505

(* ssin1 is the same as ssin *)
# let ssin1 = nssin 1;;
val ssin1 : float -> float = <fun>
# ssin 1.0;;
```

```
- : float = 0.838668418165605112
# ssin1 1.0;;
- : float = 0.838668418165605112
# ssin 2.0;;
- : float = 0.906268960387323297
# ssin1 2.0;;
- : float = 0.906268960387323297

(* ssin10 is flatter than ssin1 *)
# let ssin10 = nssin 10;;
val ssin10 : float -> float = <fun>
# ssin10 1.0;;
- : float = 0.813861644301632103
# ssin1 1.0;;
- : float = 0.838668418165605112
# ssin1 (pi /. 2.0);;
- : float = 0.996669443518683806
# ssin10 (pi /. 2.0);;
- : float = 0.96718919486859356
```

---

1. The recursive and iterative forms of `accumulate` exist in the OCaml standard library as `List.fold_right` and `List.fold_left` respectively. You are not allowed to use these functions in your solution to this problem. ↩