CS 4: Fundamentals Of Computer Programming, Winter 2017

Assignment 3: Compound data, abstraction, and lists

Due: *Friday, February 3, 02:00:00*

Coverage

This assignment covers the material up to lecture 8, corresponding to section 2.2.2 of SICP.

What to hand in

All of your code should be saved to a file named `lab3.ml`. This files should be submitted to csman as usual.

Part @: OCaml notes

Testing

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

1. a `.mli` OCaml interface file (for this assignment: `lab3.mli`)
2. a (partial) test script (for this assignment: `tests_lab3.ml`)
3. a `Makefile`.)

You should download all of these files into the directory in which you are writing your `lab3.ml` code. You should not change them, and you should not submit them as part of your assignment submission to csman.

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, as we've previously described. To run the test script, type this:

```
$ make test
```

This will compile the `tests_lab3.ml` file (which contains the unit tests) and output an executable program called `tests_lab3`. 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 assignment 2 for details on this. 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
```

Records and field punning

We will be using OCaml records extensively in this assignment, so this is a good time to introduce a nifty feature (which we also mentioned in lecture 7) which makes writing code that uses records much easier. This feature is called *field punning* for reasons that will become clear shortly.

Consider a simple record datatype for complex numbers:

```
type complex = { real : float; imag : float }
```

Writing code that pattern-matches against a complex number can be a bit annoying:

```
let magnitude { real = r; imag = i } = sqrt (r *. r +. i *. i)
```

What's annoying about this is that we have to come up with new names for the real and imaginary part of the complex number. Of course, we could do it this way without pattern matching:

```
let magnitude c = sqrt (c.real *. c.real +. c.imag *. c.imag)
```

because records support a "dot syntax" to access their components (like Python objects). However, this isn't really an improvement. Yet another way to write this would be as follows:

```
let magnitude { real = real; imag = imag } = sqrt (real *. real +. imag *. imag)
```

The right-hand side of this definition looks pretty good, but the left-hand side is weird. It's legal, because in OCaml you can use a field name as a variable name (they exist in different *namespaces*, to be technical), but it's ugly. What field punning allows you to do is to write this code like this:

```
let magnitude { real; imag } = sqrt (real *. real +. imag *. imag)
```

What this does is use the (variable) name `real` as the value of the (field) name `real`, and similarly for `imag`. This is a nice way to make functions which use records and which pattern-match on those records easier to write and easier to read.

You can also use field punning when creating values of record types:

```
# let c =
    let real = 1.0 in
    let imag = 3.4 in
      { real; imag } ;;
- : complex = {real = 1.; imag = 3.4}
```

This is a bit more confusing to read, but it works fine. Remember: field names are in a different namespace than variable names. So it's even legal to do this:

```
type complex = { real : float; imag : float }
let real = 42
let imag = "this is wacky"
```

Here, the variables `real` and `imag` aren't even floats, though the corresponding field names do refer to floats.

You don't have to use field punning in this (or any other) assignment, but we recommend it. A good reference on OCaml records is [this chapter](#) from Real World OCaml.

Signalling errors

We will soon start to explore exception handling in OCaml, but in the meantime, if at any time you need to signal an error in a function (for instance, if the function received an invalid input), you can use the `invalid_arg` and `failwith` functions. They both take a single string as their only argument, and raise an exception which uses that string as an error message. The difference is that `invalid_arg` raises an `Invalid_argument` error (which is most appropriate if a function received an invalid argument), and `failwith` raises a `Failure` exception (which is more appropriate for other situations). For instance:

```
$ ocaml
# invalid_arg "argument should be > 0";;
```

```
Exception: Invalid_argument "argument should be > 0".
# raise (Invalid_argument "argument should be > 0");;  (* same thing *)
Exception: Invalid_argument "argument should be > 0".
# failwith "everything is messed up for no good reason!";;
Exception: Failure "everything is messed up for no good reason!".
# raise (Failure "everything is messed up for no good reason!");; (* same thing *)
Exception: Failure "everything is messed up for no good reason!".
```

As you can see, `invalid_arg` and `failwith` are just functions that raise particular exceptions.

The `function` keyword

In OCaml code, it's extremely common to see this pattern:

```
let f n =
  match n with
    | 0 -> 1
    | n' -> n' + 1
```

In other words, you are defining a function with only one argument, and you are immediately pattern-matching on that argument. This is so common that there is a shorthand way of writing this:

```
let f = function
  | 0 -> 1
  | n' -> n' + 1   (* we could have used n here too *)
```

Note that `function` is not the same thing as `fun`! `function` just allows you to leave off the last argument of the function and instead pattern-match on it. `fun` can be used with any number of arguments, and doesn't automatically define a pattern match (though you can add a `match` statement inside the `fun`, of course).

Note also that `function` can be used with a functions of more than one argument. For instance, this (not very useful) function:

```
let f x y =
  match y with
    | 0 -> x + n
    r n -> x + n * n
```

could be written like this:

```
let f x = function
  | 0 -> x + n
  | n -> x + n * n
```

This is harder to read, but the `function` keyword takes the place of the `y` argument and the `match` expression.

> *Fine point:* There are some cases where you can instantly pattern match on a `fun` definition if the argument has only one kind of structure you can pattern match on. For instance:
>
> ```
> let sum_of_pair = fun (x, y) -> x + y
> (* Or equivalently: let sum_of_pair (x, y) = x + y *)
> ```
>
> defines a function `sum_of_pair` which takes a 2-tuple of `int`s as its only argument and immediately pattern matches on it to extract the two components. This is similar to what we did in the `magnitude` function above.

Using `function` where appropriate can make your code more concise and easier to read, and we recommend that you use it in those cases.

Terminology: "cons" and "append"

Sometimes, we will refer to the act of adding an item to the front of a list using the `::` operator as "cons-ing" the item to the front of the list. This terminology comes from the Scheme programming language (which borrowed it from the even older

Lisp language), where the corresponding operator is a function called `cons`. Even though CS 4 doesn't use Scheme (unlike SICP), we will use this term because there is no suitable replacement. This term is also commonly used in the functional programming literature for a variety of languages, whether they have a `cons` function or not.

Similarly, we will refer to the adding of combining two lists together to form a single list using the `@` operator as "appending" the two lists. This is standard terminology.

Part A: Data abstraction

1. [**20**] SICP, exercise 2.2

   Consider the problem of representing points and line segments in a plane. Points will be represented as a record type called `point`, containing two (floating-point) numbers (with field names `x` and `y`). Line segments will be represented as a record type called `segment` containing two points: a starting point (a field called `startp`) and an ending point (a field called `endp`). Define the two datatypes using OCaml `type` definitions.

   Using this representation, define the following functions:

   - `midpoint_segment`: this function will take a single `segment` as its only argument and return a `point` which is the midpoint of the line segment.
   - `segment_length`: this function will take a single `segment` as its only argument and return a `float` which is the length of the line segment.
   - `print_point`: this function will take a single `point` as its only argument and print its representation to the terminal.

   Use the `Printf.printf` function with a suitable format string to print points. Use the `%g` formatting directive to print floating-point numbers without extra trailing zeros. The output should look like this when tested in the interactive OCaml interpreter:

   ```
   # let p1 = ... ;; (* x = 1.0, y = 0.0 *)
   # print_point p1;;
   (1, 0)
   # let p2 = ... ;; (* x = 3.4, y = -4.5 *)
   # print_point p2;;
   (3.4, -4.5)
   ```

   You may find the built-in functions `abs_float` and `sqrt` useful for computing segment lengths.

   Finally, for testing purposes, write the following functions:

   - `make_point`: this function takes two `float` arguments (representing the `x` and `y` coordinates) and returns a `point` in your representation.
   - `make_segment`: this function takes two `point` arguments (representing the start and endpoints) and returns a `segment` in your representation.
   - `get_coords`: this function takes a `point` argument and returns a two-tuple of the (x, y) coordinates.
   - `get_points`: this function takes a `segment` argument and returns a two-tuple of the (start, end) points.

   Note that these define a primitive abstraction layer for `point`s and `segment`s.

2. [**40**] SICP, exercise 2.3

   Implement a representation for rectangles in a plane. Create functions that compute the perimeter and the area of a given rectangle. Now implement a different representation for rectangles. Can you design your system with suitable abstraction barriers, so that the same perimeter and area functions will work using either representation?

The first representation will store only a pair of points, representing the lower-left corner and the upper-right corner. The second representation will store four numbers, representing the lower and upper `x` values, and the lower and upper `y` values. Use the point and segment abstractions developed in the previous problem in your solution of this problem (you don't have to re-type that code). For both representations, define these accessors:

```
rectangle_lower_segment
rectangle_upper_segment
rectangle_left_segment
rectangle_right_segment
```

Note: don't define the accessors for one representation in terms of the accessors for the other representation even if you can. Pretend that the code for each representation was written by a different programmer with no knowledge of the other representation.

Then define the functions `rectangle_perimeter` and `rectangle_area` using *only* these accessors and functions from the point representation described above.

Since we want to be able to test both representations, use the following names for the second representation's functions:

```
rectangle_lower_segment2
rectangle_upper_segment2
rectangle_left_segment2
rectangle_right_segment2
rectangle_perimeter2
rectangle_area2
```

Note, though, that the code for `rectangle_perimeter` and `rectangle_perimeter2` will be identical except for name changes (and similarly for `rectangle_area` and `rectangle_area2`). The code for the other corresponding functions will necessarily be different, because they will use different data representations.

Judicious use of field punning (see the previous section) will make this code much easier to write. Also, don't forget about the dot syntax for accessing record fields, as it can be useful here too.

Finally, for testing purposes, write the following functions:

- `make_rectangle`: this function takes two `point` arguments and creates a rectangle using the first representation.
- `make_rectangle2`: this function takes four `float` arguments and creates a rectangle using the second representation.

3. [**30**] SICP, exercise 2.4

There are many ways to represent data types. For instance, if we want to represent pairs of values, we could write:

```
type ('a, 'b) pair = Pair of 'a * 'b
let first (Pair (x, _)) = x
let second (Pair (_, y)) = y
```

Note that `pair` is a polymorphic type, with two type variables `'a` and `'b`, since the same definition will work for arbitrary types. (OCaml already defines the built-in functions `fst` and `snd` to extract the first and second elements of an arbitrary two-tuple.)

Alternatively, we could just use "bare" two-tuples as our representation, or some kind of record type with two fields. A more exotic representation of pairs uses a function with two arguments as the pair data type. Here is part of the code for this representation:

```
let make_pair x y = fun m -> m x y
(* Or, equivalently: let make_pair x y m = m x y *)
let first z = z (fun x y -> x)
```

What is the corresponding definition of `second`? Write the code in your answer (not in a comment). Then verify (in a comment) that `first (make_pair x y)` yields `x` for any objects `x` and `y`. (You don't need to do a full substitution model evaluation; just enough to show that you get the correct answer.) Then write out (again in a comment) the full

substitution model evaluation of `second (make_pair 1 2)`. (Spoiler alert: the answer should be `2`.) The evaluation isn't too long (about 25 lines).

This representation for pairs is actually how pairs are represented in lambda calculus (LC), which adds strength to the argument that LC is in fact capable of representing arbitrary data and computations on that data.

4. [**20**] SICP, exercise 2.5

   Show that we can represent pairs of nonnegative integers using only numbers and arithmetic operations if we represent the pair `a` and `b` as the integer that is the product 2^a3^b. Give the corresponding definitions of the functions `make_pair`, `first`, and `second`. To avoid name clashes with the previous problem, call them `make_pairi`, `firsti`, and `secondi`.

   Before you do this, define the following two helper functions:

   - `pow`: This function takes two `int` arguments and returns the first raised to the power of the second. So `pow 2 10` would give `1024`.
   - `int_log`: This function takes two `int` arguments and returns the integer logarithm of the second to the base of the first, which we define as being the number of times that the base can be evenly divided into the number. Examples:

     ```
     int_log 2 1024 --> 10
     int_log 3 1024 --> 0    (* 1024 is not divisible by 3 *)
     int_log 2 3888 --> 4
     int_log 3 3888 --> 5    (* 3888 = 2^4 * 3^5 *)
     ```

     In general, if `pow a b` is `c`, then `int_log a c` is `b`.

   Once these functions have been defined, you can (and should) trivially define `make_pairi`, `firsti` and `secondi` in terms of them.

   You may assume that all arguments are positive integers, except that the second argument of `pow` can also be `0`.

5. [**20**] A very simple representation of non-negative integers is called the "unary number" representation. One way to represent a non-negative integer in unary is as follows:

   ```
   0 --> [blank]
   1 --> 1
   2 --> 11
   3 --> 111
   4 --> 1111
   5 --> 11111
   etc.
   ```

   Clearly, the digit `1` is just a placeholder; we could just as well use anything else. In this problem, we'll write some functions to work with unary numbers in OCaml. Our first representation will use lists: zero will be the empty list, one will be the list containing only the `unit` value (`[()]`), two will be the list containing two `unit` values (`[(); ()]`), and so on. So we have:

   ```
   0 --> []
   1 --> [()]
   2 --> [(); ()]
   3 --> [(); (); ()]
   4 --> [(); (); (); ()]
   5 --> [(); (); (); (); ()]
   etc.
   ```

   This can be expressed as the following OCaml code, which is our abstraction layer for working with unary integers:

   ```
   let zero = []

   let is_zero = function
     | [] -> true
     | () :: _ -> false
   ```

```
let succ u = () :: u
```

You need to define the following functions:

- `prev` This function takes one argument (a unary representation of an integer) and returns a unary integer one less than the argument. If the argument is (unary) zero, use the `invalid_arg` function with a reasonable error message to raise an exception.
- `integer_to_unary` This function takes one argument (an OCaml `int`) and returns a unary number representation of the same integer.
- `unary_to_integer` This function takes one argument (a unary representation of an integer) and returns the corresponding OCaml `int`.
- `unary_add` This function takes two arguments, both of which are unary representations of integers. It returns the result of adding the two integers together. The return value is a unary representation of an integer, not an OCaml `int`.

We are going to insist on a few more restrictions. Only the `prev` function should use pattern matching. The rest of the functions should only use `if`/`then`/`else` statements, the abstraction layer functions (`zero`, `is_zero`, `succ`, and `prev`), recursion, integer operations where they are essential, and nothing else.

Here are some examples:

```
# prev [()];;
- : unit list = []

# prev [(); (); (); (); ()];;
- : unit list = [(); (); (); ()]

# integer_to_unary 0;;
- : unit list = []

# integer_to_unary 1;;
- : unit list = [()]

# integer_to_unary 10;;
- : unit list = [(); (); (); (); (); (); (); (); (); ()]

# unary_to_integer [(); (); (); (); (); (); (); (); ()];;
- : int = 9

# unary_to_integer [(); (); ()];;
- : int = 3

# unary_to_integer [()];;
- : int = 1

# unary_to_integer [];;
- : int = 0

# unary_add [(); (); ()] [];;
- : unit list = [(); (); ()]

# unary_add [] [(); (); ()];;
- : unit list = [(); (); ()]

# unary_add [(); ()] [(); (); ()];;
- : unit list = [(); (); (); (); ()]

# unary_to_integer
    (unary_add (integer_to_unary 1001) (integer_to_unary 65535));;
- : int = 66536
```

Now consider changing the the list-of-unit representation to this alternative unary representation:

```
type nat = Zero | Succ of nat

let zero' = Zero

let is_zero' = function
  | Zero -> true
  | Succ _ -> false

let succ' u = Succ u
```

Define `prev'` for this representation (you can use pattern matching). Do the other definitions (`integer_to_unary`, `unary_to_integer`, `unary_add`) have to change from their definitions in the previous representation (other than obvious name changes *e.g.* `is_zero` to `is_zero'`)? Write your answer in a comment. If the answer is yes, indicate which changes other than name changes have to be made.

6. [**45**] SICP, exercise 2.6

   In case representing pairs as functions wasn't mind-boggling enough, consider that, in a language that can manipulate functions as data, we can even get by without numbers (at least insofar as nonnegative integers are concerned) by implementing zero and the operation of adding one as:

   ```
   let zero = fun s -> fun z -> z
     (* or equivalently: let zero = fun s z -> z *)
     (* or equivalently: let zero s z = z *)

   let add1 n = fun s -> fun z -> s (n s z)
     (* or equivalently: let add1 n = fun s z -> s (n s z) *)
     (* or equivalently: let add1 n s z = s (n s z) *)
   ```

   This representation is known as the "Church numeral" representation, after its inventor, Alonzo Church, the logician who invented the lambda calculus. Conceptually, `s` means "successor" and `z` means "zero", though in reality they don't really mean anything since they are just formal arguments of functions.

   Define `one` and `two` directly (not in terms of `zero` and `add1`). (Hint: Use substitution to evaluate `add1 zero` and see what you get.) Continue by additionally defining all the integers from `three` to `ten` as Church numerals. Do not use `add1` or any of the smaller Church numerals to define the Church numerals. All Church numerals should have two arguments: `s` and `z` (like `zero` does).

   Define an addition function called `add` for Church numerals. Don't use `add1` in this definition, and don't use regular OCaml `int`s in this function either (directly or indirectly). It should have the form:

   ```
   let add m n s z = ...
   (* equivalent to: let add m n = fun s z -> ... *)
   ```

   where `m` and `n` are the Church numerals to be added. Note that when you pass Church numerals `m` and `n` to `add` you get a function of two arguments `s` and `z` as the return value; that is the Church numeral which is the sum of `m` and `n`. The definition of `add` is extremely short.

   Then define a function called `church_to_integer` which, given a Church numeral, returns the corresponding non-negative integer. This is also a very short function.

7. [**45**] We actually cheated a little bit on the previous problem. Although we defined Church numerals in OCaml and were able to define useful functions on them, in fact, Church's lambda calculus is *untyped*, whereas OCaml is very definitely typed, as you are well aware of by now. (Actually, Church invented a typed lambda calculus as well, but that's not what we're referring to here.) Therefore, OCaml assigns types to all of the Church numerals (which are actually functions) as well as to the functions on Church numerals (`add1` and `add`). If you look at the `lab1.mli` interface file, you will see some types that you might not have expected:

   ```
   val zero : 'a -> 'b -> 'b
   val add1 : (('a -> 'b) -> 'c -> 'a) -> ('a -> 'b) -> 'c -> 'b
   val one : ('a -> 'b) -> 'a -> 'b
   val two : ('a -> 'a) -> 'a -> 'a
   val three : ('a -> 'a) -> 'a -> 'a
   val four : ('a -> 'a) -> 'a -> 'a
   val five : ('a -> 'a) -> 'a -> 'a
   ```

```
val six : ('a -> 'a) -> 'a -> 'a
val seven : ('a -> 'a) -> 'a -> 'a
val eight : ('a -> 'a) -> 'a -> 'a
val nine : ('a -> 'a) -> 'a -> 'a
val ten : ('a -> 'a) -> 'a -> 'a
val add : ('a -> 'b -> 'c) -> ('a -> 'd -> 'b) -> 'a -> 'd -> 'c
val church_to_integer : ((int -> int) -> int -> 'a) -> 'a
```

These are the types that were automatically inferred for the functions as defined in `lab3.ml`. *Side note:* you can generate these type signatures yourself by running:

```
$ ocamlc -i lab3.ml
```

on your `lab3.ml` file (assuming that it's correct).

One peculiar aspect of this is that `zero` and `one` don't have the same types as the rest of the Church numerals. We might be OK with `one`, because it's more general than the subsequent Church numerals but compatible with them, but `zero` is a completely different type. And `church_to_integer` doesn't even return an `int`! What is *that* all about? Let's compile the code and try it out:

```
$ ocamlc -c lab3.mli  (* the code we supplied you with *)
$ ocamlc -c lab3.ml   (* your code *)
$ ocaml
# #load "lab3.cmo";;
# open Lab3;;
# church_to_integer;;
- : ((int -> int) -> int -> 'a) -> 'a = <fun>
# zero;;
- : 'a -> 'b -> 'b = <fun>
# one;;
- : ('a -> 'b) -> 'a -> 'b = <fun>
# church_to_integer zero;;
- : int = 0
# church_to_integer one;;
- : int = 1
# church_to_integer ten;;
- : int = 10
```

Somehow, it works: `church_to_integer` returns an `int` when given any of the Church numerals we defined. By assigning types to the type variables in `church_to_integer` when the argument of that function is either `zero` or `one`, explain why both `church_to_integer zero` and `church_to_integer one` return an integer. Write your answer as an OCaml comment.

Hint: Since type variables are simply arbitrary names, you can write the type of `church_to_integer` as:

```
val church_to_integer : ((int -> int) -> int -> 'c) -> 'c
```

so that there is no chance of confusing the `'a` in its type with the `'a` in the types of `zero` and `one`, which could refer to different types. Similarly, the `'a` and `'b` types in `zero` don't have to be the same as the `'a` and `'b` types in `one` for expressions involving `zero` or `one`, and specifically for `church_to_integer zero` and `church_to_integer one`.

Your answer should indicate what type the type variables `'a`, `'b`, and `'c` have to be when evaluating `church_to_integer zero` and `church_to_integer one`. They won't all be the same in both cases (which is OK because `church_to_integer` has a polymorphic type!). Then use this to argue that `church_to_integer zero` and `church_to_integer one` both return values of type `int`.

Also, don't forget that the "type arrow" (`->`) associates to the right, so the type `'a -> 'b -> 'c` means `'a -> ('b -> 'c)`. It's a very good idea to "desugar" type expressions in this manner before trying to compute type variable assignments.

This process of assigning types to type variables in a consistent manner is called *type unification* and is one of the things that the OCaml compiler does for us.

This is a somewhat challenging problem, so feel free to ask the TAs for help if you need it.

Part B: Working with lists

1. [**10**] SICP, exercise 2.17

   Define a function `last_sublist` that returns the list that contains only the last element of a given (nonempty) list:

   ```
   # last_sublist [23; 72; 149; 34];;
   - : int list = [34]
   ```

   Your function should signal an error if there is no last pair. Use `invalid_arg` to generate the error; the error message should be `"last_sublist: empty list"` (that *exact* message; we'll check it!).

   You should write this function using pattern matching and the `function` keyword, since there is only one argument and the function pattern matches on it. The resulting function should generate a linear iterative process when run (in other words, the function should be tail recursive.) Also, this function should work on arbitrary lists (the element type shouldn't matter).

2. [**10**] SICP, exercise 2.18

   Define a function `reverse` that takes a list as argument and returns a list of the same elements in reverse order:

   ```
   # reverse [1; 4; 9; 16; 25];;
   - : int list = [25; 16; 9; 4; 1]
   # reverse [];;
   - : 'a list = []
   # reverse [[1; 4]; [9]; [16; 25]];;
   - : int list list = [[16; 25]; [9]; [1; 4]]
   ```

   Note the type of the last example; it is a `list` of `int list`s!

   Your `reverse` function should have a linear time complexity, be tail recursive, and should (naturally) work on lists of any element types. It should *not* use the list append (`@`) operator.

   There is a library function called `List.rev` which reverses lists. Obviously, you shouldn't use it in your answer to this problem.

3. [**10**] SICP, exercise 2.21

   The function `square_list` takes a list of integers as argument and returns a list of the squares of those numbers.

   ```
   # square_list [1; 2; 3; 4]
   - : int list = [1; 4; 9; 16]
   ```

   Here are two different definitions of `square_list`. Complete both of them by filling in the missing expressions:

   ```
   let rec square_list = function
     | [] -> []
     | h :: t -> <??>

   let square_list2 items = List.map <??> <??>
   ```

   Use the OCaml online documentation to find the definition of `List.map`. *Hint:* Do a web search for "OCaml standard library".

4. [**20**] SICP, exercise 2.22

   Louis Reasoner tries to rewrite the first `square_list` function from the previous problem so that it evolves an iterative process:

   ```
   let square_list items =
     let rec iter things answer =
       match things with
         | [] -> answer
   ```

```
      | h :: t -> iter t ((h * h) :: answer)
  in iter items []
```

Unfortunately, defining `square_list` this way produces the answer list in the reverse order of the one desired. Why? Write your answer in a comment.

Louis then tries to fix his bug by interchanging the arguments to the `::` operator:

```
let square_list items =
  let rec iter things answer =
    match things with
      | [] -> answer
      | h :: t -> iter t (answer :: (h * h))
  in iter items []
```

This doesn't work either. Explain why in a comment.

Can you modify Louis' second solution slightly to make it work properly? (By "slightly", we mean changing the `::` operator to a different list operator and making one more small modification? If so, would the resulting function be efficient?

5. Write the following functions which involve lists.

    1. **[15]** A function called `count_negative_numbers` that counts the negative integers in a list and returns the count.

    2. **[10]** A function called `power_of_two_list` that takes in an integer, `n`, and creates a list containing the first `n` powers of 2 starting with 2^0=1 and up to 2^{n-1}. You can use the `pow` function you defined above in this problem.

    3. **[25]** A function called `prefix_sum` that takes in a list of numbers, and returns a list containing the prefix sum of the original list. *e.g.* `prefix-sum [1; 3; 5; 2] ==> [1; 4; 9; 11]`. The prefix sum is the sum of all of the elements in the list up to that point, so for the list `[1; 3; 5; 2]` the prefix sum is `[1; 1+3; 1+3+5; 1+3+5+2]` or `[1; 4; 9; 11]`.

6. [**15**] SICP, exercise 2.27

    Modify the `reverse` function you defined previously in this assignment to produce a `deep_reverse` function that takes a list of lists (of arbitrary type) as argument and returns as its value the same list with its elements reversed and with its immediate sublists reversed as well. For example,

```
# let lst = [[1; 2]; [3; 4]];;
val lst : int list list = [[1; 2]; [3; 4]]
# reverse lst;;
- : int list list = [[3; 4]; [1; 2]]
# deep_reverse lst;;
- : int list list = [[4; 3]; [2; 1]]
# let lst2 = [[[1; 2]; [3; 4]]; [[5; 6]; [7; 8]]];;
- : int list list list = [[[1; 2]; [3; 4]]; [[5; 6]; [7; 8]]]
# deep_reverse lst2;;
- : int list list list = [[[7; 8]; [5; 6]]; [[3; 4]; [1; 2]]]
```

7. [**30**] Sometimes people learning OCaml from a background of dynamically typed languages miss having lists where you can store arbitrary values. In particular, it would be nice to have a list that can store either values of a particular type `'a` or a list of such values, or a list of lists of such values, and so on, with any combination of values and lists. This is easily modeled in OCaml by defining a new datatype:

```
type 'a nested_list =
  | Value of 'a
  | List of 'a nested_list list
```

Now you can have a list-like data structure that can mix together values and lists with arbitrary nesting. (This datatype is actually isomorphic to a datatype called "S-expressions" which are used as the basis of the syntax in Scheme and related languages.) For instance:

```
# Value 10;;
- : int nested_list = Value 10
```

```
# List [Value 10];;
- : int nested_list = List [Value 10]
# List [Value 10; Value 20; Value 30];;
- : int nested_list = List [Value 10; Value 20; Value 30]
# List [Value 10; List [Value 20; List [Value 30; Value 40]; Value 50]; Value 60];;
- : int nested_list =
List
 [Value 10; List [Value 20; List [Value 30; Value 40]; Value 50]; Value 60]
# List [List [Value 1; Value 2]; List [Value 3; Value 4]];;
- : int nested_list = List [List [Value 1; Value 2]; List [Value 3; Value 4]]
```

Define a version of the `deep_reverse` function from the previous problem that works on `nested_list`s. Call it `deep_reverse_nested`. Don't use the `reverse` or `deep_reverse` functions in your definition. Note that `Value`s don't get reversed, because there is no way to reverse them; only the `List` constructor contents get reversed.

Examples:

```
# deep_reverse_nested (Value 10);;
- : int nested_list = Value 10
# deep_reverse_nested (List [Value 10; Value 20; Value 30; Value 40]);;
- : int nested_list = List [Value 40; Value 30; Value 20; Value 10]
# deep_reverse_nested (List [List [Value 10; Value 20]; List [Value 30; Value 40]]);;
- : int nested_list =
List [List [Value 40; Value 30]; List [Value 20; Value 10]]
# deep_reverse_nested (List [Value 10; List [Value 20; Value 30]]);;
- : int nested_list = List [List [Value 30; Value 20]; Value 10]
# deep_reverse_nested (List [List [Value 10; Value 20]; Value 30]);;
- : int nested_list = List [Value 30; List [Value 20; Value 10]]
# deep_reverse_nested (List [Value 10; List [Value 20; List [Value 30; Value 40]; Value 50]; Value 60]);;
- : int Lab3.nested_list =
List
 [Value 60; List [Value 50; List [Value 40; Value 30]; Value 20]; Value 10]
```

Hint: Check for the `Value` case first, because if the input is just a `Value`, it doesn't need to be reversed. Otherwise, extract the (OCaml) list from the `List` constructor and pass it to a recursive helper function which will assemble the result.
