```
(*
                              CS51 Lab 1
                     Basic Functional Programming
 *)
(*======================================================================
Readings:

    This lab builds on material from Chapters 1-6 of the textbook
    <http://book.cs51.io>, which should be read before the lab session.

Objective:

    This lab is intended to get you up and running with the course's
    assignment submission system, and thinking about core concepts
    introduced in class, including:

        * concrete versus abstract syntax
        * atomic types
        * first-order functional programming
======================================================================*)

(*======================================================================
Part 0: Testing your Gradescope Interaction

Labs and problem sets in CS51 are submitted using the Gradescope
system. By now, you should be set up with Gradescope.

........................................................................
Exercise 1: To make sure that the setup works, submit this file,
just as is, under the filename "lab1.ml", to the Lab 1 assignment on
the CS51 Gradescope web site.
........................................................................

When you submit labs (including this one) Gradescope will check that
the submission compiles cleanly, and if so, will run a set of unit
tests on the submission. For this part 0 submission, the submission
should compile cleanly, but most of the unit tests will fail (as you
haven't done the exercises yet). But that's okay. We won't be checking
the correctness of your labs until the "virtual quiz" this
weekend. See the syllabus for more information about virtual quizzes,
our very low stakes method for grading labs.

Now let's get back to doing the remaining exercises so that more of
the unit tests pass.
```

```

     ******************************************************************
     We use the commenting convention in our code throughout the
     course that code snippets within comments are demarcated with
     backquotes, for instance, `x + 3` or `fun x -> x`. You can think
     of this as corresponding to the fixed-width font in the textbook.
     ******************************************************************

.......................................................................
Exercise 2: So that you can see how the unit tests in labs work,
replace the `failwith` expression below with the integer `42`, so that
`exercise2` is a function that returns `42` (instead of failing). When
you submit, the Exercise 2 unit test should then pass.
.....................................................................*)

let exercise2 () = failwith "exercise2 not implemented" ;;

(* From here on, you'll want to test your lab solutions locally before
submitting them at the end of lab to Gradescope. A simple way to do that
is to cut and paste the exercises into an OCaml interpreter, such as
utop, which you run with the command

    % utop

You can also use the more basic version, ocaml:

    % ocaml

We call this kind of interaction a "read-eval-print loop" or
"REPL". Alternatively, you can feed the whole file to OCaml with the
command:

    % ocaml < lab1.ml

to see what happens. We'll introduce other methods soon. *)

(*======================================================================
Part 1: Concrete versus abstract syntax

We've distinguished concrete from abstract syntax. Abstract syntax
corresponds to the substantive tree structuring of expressions;
concrete syntax corresponds to the particulars of how those structures
are made manifest in the language's textual notation.

In the presence of multiple operators, issues of precedence and
associativity become important in constructing the abstract syntax
```

```
from the concrete syntax.

........................................................................
Exercise 3: Consider the following abstract syntax tree:

     ~-
      |
      |
      -
      ^
     / \
    /   \
   5     3

that is, the negation of the result of subtracting 3 from 5.  To
emphasize that the two operators are distinct, we've used the concrete
symbol `~-` (a tilde followed by a hyphen character, an alternative
spelling of the negation operation; see the Stdlib module) to notate
the negation.

How might this *abstract* syntax be expressed in the *concrete* syntax
of OCaml using the fewest parentheses? Replace the `failwith`
expression with the appropriate OCaml expression to assign the value
to the variable `exercise3` below.
......................................................................*)

let exercise3 () : int = failwith "exercise3 not implemented" ;;

(* Hint: The OCaml concrete expression `~- 5 - 3` does *not*
correspond to the abstract syntax above.

........................................................................
Exercise 4: Draw the tree that the concrete syntax `~- 5 - 3` does
correspond to. Check it with a member of the course staff.
......................................................................*)


(*......................................................................
Exercise 5: Associativity plays a role in cases when two operators
used in the concrete syntax have the same precedence. For instance,
the concrete expression `2 + 1 + 0` might have abstract syntax as
reflected in the following two parenthesizations:

    2 + (1 + 0)

or
```

```

    (2 + 1) + 0

As it turns out, both of these parenthesizations evaluate to the same
result (`3`). (That's because addition is an associative operation.)

Construct an expression that uses an arithmetic operator twice, but
evaluates to two different results dependent on the associativity of
the operator. Use this expression to determine the associativity of
the operator. Check your answer with a member of the course staff if
you'd like.
......................................................................*)

(*======================================================================
Part 2: Types and type inference

........................................................................
Exercise 6: What are appropriate types to replace the ??? in the
expressions below? Test your solution by uncommenting the examples
(removing the `(*` and `*)` at start and end) and verifying that no
typing error is generated.
......................................................................*)

(* <-- After you've replaced the ???s, remove this start-comment line...

let exercise6a : ??? = 42 ;;

let exercise6b : ??? =
  let greet y = "Hello " ^ y
  in greet "World!";;

let exercise6c : ??? =
  fun x -> x +. 11.1 ;;

let exercise6d : ??? =
  fun x -> x < x + 1 ;;

let exercise6e : ??? =
  fun x -> fun y -> x + int_of_float y ;;

...and remove this whole end-comment line too. --> *)

(*======================================================================
Part 3: First-order functional programming

For warmup, here are some "finger exercises" defining simple functions
```

```
before moving onto more complex problems.

......................................................................
Exercise 7: Define a function `square` that squares its
argument. We've provided a bit of template code, supplying the first
line of the function definition but the body of the skeleton code just
causes a failure by forcing an error using the built-in `failwith`
function. Edit the code to implement `square` properly.

Test out your implementation of `square` by modifying the template
code below to define `exercise7` to be the `square` function applied
to the integer 5. You'll want to replace the `0` with the correct
function call.

Thorough testing is important in all your work, and we hope to impart
this view to you in CS51. Testing will help you find bugs, avoid
mistakes, and teach you the value of short, clear, testable
functions. In the file `lab1_tests.ml`, we've put some prewritten
tests for `square` using the testing method of Section 6.5 in the
book. Spend some time understanding how the testing function works and
why these tests are comprehensive. Then test your code by compiling
and running the test suite:

    % ocamlbuild -use-ocamlfind lab1_tests.byte
    % ./lab1_tests.byte

You should add some tests for other functions in the lab to get some
practice with automated unit testing.
....................................................................*)

let square (x : int) : int  =
  failwith "square not implemented" ;;

let exercise7 = 0 ;;

(*....................................................................
Exercise 8: Define a function `exclaim`,that, given a string,
"exclaims" it by capitalizing it and suffixing an exclamation mark.
The `String.capitalize_ascii` function may be helpful here. For
example, you should get the following behavior:

   # exclaim "hello" ;;
   - : string = "Hello!"
   # exclaim "Ciao" ;;
   - : string = "Ciao!"
   # exclaim "what's up" ;;
```

```ocaml
   - : string = "What's up!"
......................................................................*)

let exclaim (text : string) : string =
  failwith "exclaim not implemented";;

(*......................................................................
Exercise 9: Define a function `needs_small_bills` that determines, given a
price, if one will need a bill smaller than a 20 to pay for the
item. For instance, a price of 100 can be paid for with 20s (and
larger denominations) alone, but a price of 105 will require a bill
smaller than a 20 (for the 5 left over after the 100 is paid). We will
assume (perhaps unrealistically) that all prices are given as integers
and (more realistically) that 50s, 100s, and larger denomination bills
are not available, only 1s, 5s, 10s, and 20s. In addition, you may
assume all prices given are non-negative.

   # needs_small_bills 105 ;;
   - : bool = true
   # needs_small_bills 100 ;;
   - : bool = false
   # needs_small_bills 150 ;;
   - : bool = true
......................................................................*)

let needs_small_bills (price : int) : bool =
  failwith "needs_small_bills not implemented" ;;

(*......................................................................
Exercise 10:

The calculation of the date of Easter, a calculation so important to
early Christianity that it was referred to simply by the Latin
"computus" ("the computation"), has been the subject of innumerable
algorithms since the early history of the Christian church.

The algorithm to calculate the computus function is given in Problem
31 in the textbook, which you'll want to refer to.

Write two functions that, given a year, calculate the month
(`computus_month`) and day (`computus_day`) of Easter in that year via
the Computus function.

In 2018, Easter took place on April 1st. Your functions should reflect
that:

```

```
   # computus_month 2018;;
   - : int = 4
   # computus_day 2018 ;;
   - : int = 1
.......................................................................*)

let computus_month (year : int) : int =
  failwith "computus_month not implemented" ;;
let computus_day (year : int) : int =
  failwith "computus_day not implemented" ;;

(*======================================================================
Part 4: Code review

A frustrum (see Figure 6.3 in the textbook) is a three-dimensional
solid formed by slicing off the top of a cone parallel to its
base. The formula for the volume of a frustrum in terms of its radii
and height is given in the textbook as well.

As an experienced programmer at Frustromco, Inc., you've been assigned
to mentor a beginning programmer. Your mentee has been given the task
of implementing a function `frustrum_volume` to calculate the volume
of a frustrum. Here is your mentee's stab at this task:

(* frustrum_volume -- calculate the frustrum *)
let frustrum_volume a b c =
  let a =
  let s a = a * a in
  let h = b in 3.1416
  *. h /. float_of_int 3*. (a *.
  a +. c  *.  c+.a *. c) in a
;;

As this neophyte programmer's mentor, you're asked to perform a code
review on this code. You test the code out on an example -- a frustrum
with radii 3 and 4 and height 4 -- and you get

   # frustrum_volume 3. 4. 4. ;;
   - : float = 154.985599999999977

which is (more or less) the right answer. Nonetheless, you have a
strong sense that the code can be considerably improved. *)

(*.......................................................................
Exercise 11: Go over the code with your lab partner, making whatever
modifications you think can improve the code, placing your revised
```

```
version just below. Once you've converged on a version of the code
that you think is best, call over a staff member and go over your
revised code together.
.....................................................................*)

(*** Place your revised version here within this comment. ***)

(* During the code review, your boss drops by and looks over your
proposed code. Your boss thinks that the function should be compatible
with the header line given at <https://url.cs51.io/frustrum>. You
agree.

.......................................................................
Exercise 12: Revise your code (if necessary) to make sure that it uses
the header line given at <https://url.cs51.io/frustrum>.
.....................................................................*)

(*** Place your updated revised version below, *not* as a comment,
     because we'll be unit testing it. (The two lines we provide are
     just to allow the unit tests to have something to compile
     against. You'll want to just delete them and start over.) ***)
let frustrum_volume _ _ _ =
  failwith "frustrum_volume not implemented" ;;
(*======================================================================
Part 5: Utilizing recursion

.......................................................................
Exercise 13: The factorial function takes the product of an integer
and all the integers below it. It is generally notated as !. For
example, 4! = 4 * 3 * 2 * 1. Write a function `factorial` that
calculates the factorial of its integer argument. Note: the factorial
function is generally only defined on non-negative integers (0, 1, 2,
3, ...). For the purpose of this exercise, you may assume all inputs
will be non-negative.

For example,

   # factorial 4 ;;
   - : int = 24
   # factorial 0 ;;
   - : int = 1
.....................................................................*)

let factorial (x : int) : int =
  failwith "factorial not implementated" ;;

```

```
(*......................................................................
Exercise 14: Define a recursive function `sum_from_zero` that sums all
the integers between 0 and its argument, inclusive.

   # sum_from_zero 5 ;;
   - : int = 15
   # sum_from_zero 100 ;;
   - : int = 5050
   # sum_from_zero ~-3 ;;
   - : int = -6

(The sum from 0 to 100 was famously if apocryphally performed by
the mathematician Carl Friedrich Gauss as a seven-year-old, *in his
head*!)
......................................................................*)

let sum_from_zero (x : int) : int =
  failwith "sum_from_zero not implemented" ;;
```