```
(*
                              CS51 Lab 1
                        Basic Functional Programming
 *)
(*
                                SOLUTION
 *)
(*======================================================================
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.
```

```
45
46  Now let's get back to doing the remaining exercises so that more of
47  the unit tests pass.
48
49       ****************************************************************
50       We use the commenting convention in our code throughout the
51       course that code snippets within comments are demarcated with
52       backquotes, for instance, `x + 3` or `fun x -> x`. You can think
53       of this as corresponding to the fixed-width font in the textbook.
54       ****************************************************************
55
56  ................................................................
57  Exercise 2: So that you can see how the unit tests in labs work,
58  replace the `failwith` expression below with the integer `42`, so that
59  `exercise2` is a function that returns `42` (instead of failing). When
60  you submit, the Exercise 2 unit test should then pass.
61  ................................................................*)
62
63  let exercise2 () = 42 ;;
64
65  (* From here on, you'll want to test your lab solutions locally before
66  submitting them at the end of lab to Gradescope. A simple way to do that
67  is to cut and paste the exercises into an OCaml interpreter, such as
68  utop, which you run with the command
69
70      % utop
71
72  You can also use the more basic version, ocaml:
73
74      % ocaml
75
76  We call this kind of interaction a "read-eval-print loop" or
77  "REPL". Alternatively, you can feed the whole file to OCaml with the
78  command:
79
80      % ocaml < lab1.ml
81
82  to see what happens. We'll introduce other methods soon. *)
83
84  (*======================================================================
85  Part 1: Concrete versus abstract syntax
86
87  We've distinguished concrete from abstract syntax. Abstract syntax
88  corresponds to the substantive tree structuring of expressions;
89  concrete syntax corresponds to the particulars of how those structures
90  are made manifest in the language's textual notation.
```

```
91
92  In the presence of multiple operators, issues of precedence and
93  associativity become important in constructing the abstract syntax
94  from the concrete syntax.
95
96  ......................................................................
97  Exercise 3: Consider the following abstract syntax tree:
98
99        ~-
100        |
101        |
102        -
103        ^
104       / \
105      /   \
106     5     3
107
108  that is, the negation of the result of subtracting 3 from 5.  To
109  emphasize that the two operators are distinct, we've used the concrete
110  symbol `~-` (a tilde followed by a hyphen character, an alternative
111  spelling of the negation operation; see the Stdlib module) to notate
112  the negation.
113
114  How might this *abstract* syntax be expressed in the *concrete* syntax
115  of OCaml using the fewest parentheses? Replace the `failwith`
116  expression with the appropriate OCaml expression to assign the value
117  to the variable `exercise3` below.
118  ......................................................................*)
119
120  let exercise3 () : int = ~- (5 - 3) ;;
121
122  (* Hint: The OCaml concrete expression `~- 5 - 3` does *not*
123  correspond to the abstract syntax above.
124
125  ......................................................................
126  Exercise 4: Draw the tree that the concrete syntax `~- 5 - 3` does
127  correspond to. Check it with a member of the course staff.
128  ......................................................................*)
129
130  (* SOLUTION: The abstract syntax of the OCaml expression `~- 5 - 3`
131     has the subtraction as the primary operator, nesting the negation
132     of 5 and 3 as left and right children respectively, thus,
133
134        -
135        ^
136       / \
```

```
    /   \
   ~-    3
   |
   |
   5

 *)

(*.....................................................................
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.
.....................................................................*)

(* SOLUTION: We need to use an operator that, unlike `+` and `*`, is
   not associative. Examples include `-` and `/`. For instance,

   # (3 - 2) - 1 ;;
   - : int = 0
   # 3 - (2 - 1) ;;
   - : int = 2
 *)

(*=====================================================================
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.
.....................................................................*)

let exercise6a : int = 42 ;;

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

(* If you were confused about what the `^` operator does, you'd have
   found it in the `Stdlib` module described in the OCaml
   documentation online. *)

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

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

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

(* The reasoning for exercise6e goes something like this: The argument
   of the function is `x`, which returns another function, with its
   second argument `y`. The value of `y` is used as an argument of the
   `int_of_float` function, which takes a float argument, so `y`
   must be a float. The value of `x`, as argument of integer `+`,
   must be an `int`. Thus the top-level function takes `x`, an `int`,
   and `y`, a `float`, to the result of integer addition, an `int`,
   that is, `int -> float -> int`. *)

(*======================================================================
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  =
  x * x ;;

let exercise7 =
  square 5 ;;

(*.....................................................................
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" ;;
   - : string = "What's up!"
.....................................................................*)

let exclaim (text : string) : string =
  (String.capitalize_ascii text) ^ "!" ;;

(*.....................................................................
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 =
  let cutoff_size = 20 in
  (price mod cutoff_size) <> 0 ;;

(* Note the use of `<>` for inequality. You may have used `==` or `!=`
   for comparing integer values for equality or inequality. OCaml
   distinguishes two kinds of equality comparisons: *structural* and
   *physical* equality. More on the distinction later in the course
   (Section 15.1.2 in the textbook for the interested), but for now,
   it is sufficient to note that you should be using the *structural*
   equality operators: `=` for equality and `<>` for inequality. These
   differ from the operators in C and Python (which use `==` and
   `!=`); these are used as the physical equality operators in OCaml.

   We also labeled 20 our `cutoff_size`. It is generally good practice
   to name special numbers used in code. If used many times, they
   should be defined only in one location, so that changing them is
   easy, but even if used only once, explicit naming provides for
   better documentation. See the style guide section on "Constants and
   magic numbers". *)

(*.....................................................................
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
..................................................................*)

(* SOLUTION: You might have implemented a `computus_month` function like
   this:

    let computus_month (year : int) : int =
      let a = year mod 19 in
      let b = year / 100 in
      let c = year mod 100 in
      let d = b / 4 in
      let e = b mod 4 in
      let f = (b + 8) / 25 in
      let g = (b - f + 1) / 3 in
      let h = (19 * a + b - d - g + 15) mod 30 in
      let i = c / 4 in
      let k = c mod 4 in
      let l = (32 + 2 * e + 2 * i - h - k) mod 7 in
      let m = (a + 11 * h + 22 * l) / 451 in
      (h + l - 7 * m + 114) / 31 ;;

   and then just made a copy and modified it to form the `computus_day`
   function:

    let computus_day (year : int) : int =
      let a = year mod 19 in
      let b = year / 100 in
      let c = year mod 100 in
      let d = b / 4 in
      let e = b mod 4 in
      let f = (b + 8) / 25 in
      let g = (b - f + 1) / 3 in
      let h = (19 * a + b - d - g + 15) mod 30 in
      let i = c / 4 in
      let k = c mod 4 in
```

```
        let l = (32 + 2 * e + 2 * i - h - k) mod 7 in
        let m = (a + 11 * h + 22 * l) / 451 in
        (h + l - 7 * m + 114) mod 31 + 1 ;;

    However, the first twelve equations are shared between the
    computations for the month and the day, so it makes sense to split
    this common part out as its own function (an application of the
    edict of irredundancy): *)

let computus_common (year : int) : int =
  let a = year mod 19 in
  let b = year / 100 in
  let c = year mod 100 in
  let d = b / 4 in
  let e = b mod 4 in
  let f = (b + 8) / 25 in
  let g = (b - f + 1) / 3 in
  let h = (19 * a + b - d - g + 15) mod 30 in
  let i = c / 4 in
  let k = c mod 4 in
  let l = (32 + 2 * e + 2 * i - h - k) mod 7 in
  let m = (a + 11 * h + 22 * l) / 451 in
  h + l - 7 * m + 114 ;;

(* The `computus_month` and `computus_day` functions can then be
   implemented on the basis of that common calculation. *)

let computus_month (year : int) : int =
  (computus_common year) / 31 ;;

let computus_day (year : int) =
  (computus_common year) mod 31 + 1 ;;

  (* Even more redundancy can be eliminated in the computation by taking
   advantage of structured data, a topic that is coming up in the next
   lab. You might note that many of the equations come in pairs -- `b`
   and `c`, `d` and `e`, `i` and `k`, and even the final two
   calculations of the month and year -- with one of the equations a
   division (`x / y`), and the other the remainder of that division
   (`x mod y`). By defining a function to calculate both of these
   values, and returning them as a pair, we can eliminate this
   redundancy, and reduce the possibility that a change to one of the
   pairs doesn't get reflected in the other. That's another
   application of the edict of irredundancy.

    let divmod (x : int) (y : int) : int * int =
```

```
      x / y, x mod y ;;

    let computus_common (year : int) : int * int =
      let a = year mod 19 in
      let b, c = divmod year 100 in
      let d, e = divmod b 4 in
      let f = (b + 8) / 25 in
      let g = (b - f + 1) / 3 in
      let h = (19 * a + b - d - g + 15) mod 30 in
      let i, k = divmod c 4 in
      let l = (32 + 2 * e + 2 * i - h - k) mod 7 in
      let m = (a + 11 * h + 22 * l) / 451 in
      let month, previous_day = divmod (h + l - 7 * m + 114) 31
      in month, previous_day + 1 ;;

    let computus_month (year : int) : int =
      fst (computus_common year) ;;

    let computus_day (year : int) : int =
      snd (computus_common year) ;;

  Exercise to the reader: Why is the below implementation not ideal?

    let computus_month (year : int) : int =
      let january = year mod 19 in
      let february = year / 100 in
      let march = year mod 100 in
      let april = february / 4 in
      let may = february mod 4 in
      let june = (february + 8) / 25 in
      let july = (february - june + 1) / 3 in
      let august = (january * 19 + february - april - july + 15) mod 30 in
      let september = march / 4 in
      let october = march mod 4 in
      let november = (32 + 2 * may + 2 * september - august - october) mod 7 in
      let december = (january + 11 * august + 22 * november) / 451 in
      (august + november - 7 * december + 114) / 31 ;;
   *)

(*=====================================================================
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.
```

```
459
460 As an experienced programmer at Frustromco, Inc., you've been assigned
461 to mentor a beginning programmer. Your mentee has been given the task
462 of implementing a function `frustrum_volume` to calculate the volume
463 of a frustrum. Here is your mentee's stab at this task:
464
465 (* frustrum_volume -- calculate the frustrum *)
466 let frustrum_volume a b c =
467   let a =
468   let s a = a * a in
469   let h = b in 3.1416
470   *. h /. float_of_int 3*. (a *.
471   a +. c  *.  c+.a *. c) in a
472 ;;
473
474 As this neophyte programmer's mentor, you're asked to perform a code
475 review on this code. You test the code out on an example -- a frustrum
476 with radii 3 and 4 and height 4 -- and you get
477
478    # frustrum_volume 3. 4. 4. ;;
479    - : float = 154.985599999999977
480
481 which is (more or less) the right answer. Nonetheless, you have a
482 strong sense that the code can be considerably improved. *)
483
484 (*.....................................................................
485 Exercise 11: Go over the code with your lab partner, making whatever
486 modifications you think can improve the code, placing your revised
487 version just below. Once you've converged on a version of the code
488 that you think is best, call over a staff member and go over your
489 revised code together.
490 .....................................................................*)
491
492 (*** Place your revised version here within this comment. ***)
493
494 (* During the code review, your boss drops by and looks over your
495 proposed code. Your boss thinks that the function should be compatible
496 with the header line given at <https://url.cs51.io/frustrum>. You
497 agree.
498
499 .....................................................................
500 Exercise 12: Revise your code (if necessary) to make sure that it uses
501 the header line given at <https://url.cs51.io/frustrum>.
502 .....................................................................*)
503
504 (*** 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.) ***)
(* SOLUTION: There are, of course, lots of problems with the original
   code; it's almost completely unreadable and obscure. Let's start by
   at least adding white space -- line breaks and indentation -- to
   make the structure of the function clear:

   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 ;;

   That's already much better. It makes more clear that the radii are
   `a` and `c` and the height is `b`. (Maybe that's why the programmer has
   the `let h = b` renaming.) Better variable names are badly needed,
   as well as a better order for arguments. (That'll allow for
   dropping the `let h = b`, too.) We should also mark the intended
   types for the arguments and for the return value:

   let frustrum_volume (radius1 : float)
                       (radius2 : float)
                       (height : float)
                     : float =
     let a =
       let s a = a * a in
       (3.1416 *. height /. float_of_int 3)
       *. (radius1 *. radius1 +. radius2 *. radius2 +. radius1 *. radius2) in
     a ;;

   Now, what is this local function `s` that is being defined? It looks
   like a squaring function, which might have been useful in calculating
   the squares of the radii, but apparently that idea got dropped. We can
   drop the definition as well. The code also defines the answer in a
   local variable `a` (for answer?), which it just returns. There's no
   reason to name the return value in that way.

   let frustrum_volume (radius1 : float)
                       (radius2 : float)
                       (height : float)
                     : float =
     (3.1416 *. height /. float_of_int 3)
     *. (radius1 *. radius1 +. radius2 *. radius2 +. radius1 *. radius2) ;;
```

```
551
552     The squaring can be better implemented with the `**` exponentiation
553     operator, and we might as well order the three terms in the more
554     standard way (as shown in the equation in the textbook):
555
556     let frustrum_volume (radius1 : float)
557                         (radius2 : float)
558                         (height : float)
559                       : float =
560       (3.1416 *. height /. float_of_int 3)
561       *. (radius1 ** 2. +. radius1 *. radius2 +. radius2 ** 2.) ;;
562
563     Notice the "magic number" (see the style guide section on
564     "Constants and magic numbers") `3.1416`. That's presumably intended
565     to be pi. But we can make that intention clearer (and slightly more
566     accurate) by using a constant for pi. In fact, OCaml provides a
567     defined constant for pi in the Float library, `Float.pi`.
568
569     Finally, and importantly, we can update the documentation to make
570     this all clearer as well.
571
572     In the end, the code review process converges on the following: *)
573
574   (* frustrum_volume radius1 radius2 height -- Returns the volume of a
575      conical frustrum given the radii of the two faces (`radius1` and
576      `radius2`) and the perpendicular `height` *)
577
578   let frustrum_volume (radius1 : float)
579                       (radius2 : float)
580                       (height : float)
581                     : float =
582     (Float.pi *. height /. 3.)
583     *. (radius1 ** 2. +. radius1 *. radius2 +. radius2 ** 2.) ;;
584
585   (* Compare this with the original code above. Vast improvement, no? *)
586
587   (*============================================================================
588   Part 5: Utilizing recursion
589
590   ............................................................................
591   Exercise 13: The factorial function takes the product of an integer
592   and all the integers below it. It is generally notated as !. For
593   example, 4! = 4 * 3 * 2 * 1. Write a function `factorial` that
594   calculates the factorial of its integer argument. Note: the factorial
595   function is generally only defined on non-negative integers (0, 1, 2,
596   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 rec factorial (x : int) =
  if x = 0 then 1
  else x * factorial (x - 1) ;;

  (* The above code is what we expected people to produce. However,
     this code will run forever when the input is negative. Better
     practice would be to raise an error, as below, when we encounter
     an invalid input. You'll learn more about this issue in Lab 4.

     let rec factorial (x : int) =
        if x < 0 then raise (Invalid_argument "fact: arg must be non-negative")
        else if x = 0 then 1
        else x * factorial (x - 1) ;;
   *)

(*.....................................................................
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*!)
.....................................................................*)

(* Here's an approach that works recursively. The recursive cases for
   positive and negative numbers are handled separately, since in the
   former case we want to count down (using the built-in `pred`
   function) and for negatives we want to count up toward zero (using
   `succ`). *)
```

```
643
644 let rec sum_from_zero (x : int) : int =
645   if x = 0 then 0
646   else if x < 0 then x + sum_from_zero (succ x)
647   else x + sum_from_zero (pred x) ;;
648
649 (* You may notice that there's a lot of similarity between the `then`
650    and `else` branches. We could factor out the similarities by
651    narrowing the scope of the conditional test inside, and use it just
652    for selecting whether to use the function `succ` or `pred`. The
653    result is this:
654
655     let rec sum_from_zero (x : int) : int =
656       if x = 0 then 0
657       else x + sum_from_zero (if x < 0 then succ x else pred x) ;;
658
659    or even this
660
661     let rec sum_from_zero (x : int) : int =
662       if x = 0 then 0
663       else x + sum_from_zero ((if x < 0 then succ else pred) x) ;;
664
665    The latter, frankly, may be taking things too far. It's a bit too
666    "cute".
667
668    In this exercise, we were explicitly looking for this recursive
669    solution. However, there's a closed-form solution for the sum, the
670    one that Gauss himself used (see Figure 14.6 in Chapter 14 of the
671    textbook), which we can use to generate the following non-recursive
672    version.
673
674     let sum_from_zero (x : int) : int =
675       (x * (succ (abs x))) / 2 ;;
676
677    (Do you see why this also works for the negative cases?) *)
```