

```
1  (*
2           CS51 Lab 1
3           Basic Functional Programming
4   *)
5  (*
6           SOLUTION
7   *)
8  (*=====
9  Readings:
10
11 This lab builds on material from Chapters 1-6 of the textbook
12 <http://book.cs51.io>, which should be read before the lab session.
13
14 Objective:
15
16 This lab is intended to get you up and running with the course's
17 assignment submission system, and thinking about core concepts
18 introduced in class, including:
19
20     * concrete versus abstract syntax
21     * atomic types
22     * first-order functional programming
23 =====*)*
24
25 (*=====
26 Part 0: Testing your Gradescope Interaction
27
28 Labs and problem sets in CS51 are submitted using the Gradescope
29 system. By now, you should be set up with Gradescope.
30
31 .....
32 Exercise 1: To make sure that the setup works, submit this file,
33 just as is, under the filename "lab1.ml", to the Lab 1 assignment on
34 the CS51 Gradescope web site.
35 .....
36
37 When you submit labs (including this one) Gradescope will check that
38 the submission compiles cleanly, and if so, will run a set of unit
39 tests on the submission. For this part 0 submission, the submission
40 should compile cleanly, but most of the unit tests will fail (as you
41 haven't done the exercises yet). But that's okay. We won't be checking
42 the correctness of your labs until the "virtual quiz" this
43 weekend. See the syllabus for more information about virtual quizzes,
44 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 In the presence of multiple operators, issues of precedence and
92 associativity become important in constructing the abstract syntax
93 from the concrete syntax.
94
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     / \

```

```

137      /   \
138      ~-   3
139      |
140      |
141      5
142
143      *)
144
145      (*.....
146 Exercise 5: Associativity plays a role in cases when two operators
147 used in the concrete syntax have the same precedence. For instance,
148 the concrete expression `2 + 1 + 0` might have abstract syntax as
149 reflected in the following two parenthesizations:
150
151      2 + (1 + 0)
152
153 or
154
155      (2 + 1) + 0
156
157 As it turns out, both of these parenthesizations evaluate to the same
158 result (`3`). (That's because addition is an associative operation.)
159
160 Construct an expression that uses an arithmetic operator twice, but
161 evaluates to two different results dependent on the associativity of
162 the operator. Use this expression to determine the associativity of
163 the operator. Check your answer with a member of the course staff if
164 you'd like.
165      .....*)
166
167      (* SOLUTION: We need to use an operator that, unlike `+` and `*`, is
168      not associative. Examples include `-` and `/`. For instance,
169
170      # (3 - 2) - 1 ;;
171      - : int = 0
172      # 3 - (2 - 1) ;;
173      - : int = 2
174      *)
175
176      (*=====
177 Part 2: Types and type inference
178
179      .....
180 Exercise 6: What are appropriate types to replace the ??? in the
181 expressions below? Test your solution by uncommenting the examples
182 (removing the `(*` and `*)` at start and end) and verifying that no

```

```

183   typing error is generated.
184   .....*)
185
186 let exercise6a : int = 42 ;;
187
188 let exercise6b : string =
189   let greet y = "Hello " ^ y
190   in greet "World!";;
191
192 (* If you were confused about what the `^^` operator does, you'd have
193   found it in the `Stdlib` module described in the OCaml
194   documentation online. *)
195
196 let exercise6c : float -> float =
197   fun x -> x +. 11.1 ;;
198
199 let exercise6d : int -> bool =
200   fun x -> x < x + 1 ;;
201
202 let exercise6e : int -> float -> int =
203   fun x -> fun y -> x + int_of_float y ;;
204
205 (* The reasoning for exercise6e goes something like this: The argument
206   of the function is `x`, which returns another function, with its
207   second argument `y`. The value of `y` is used as an argument of the
208   `int_of_float` function, which takes a float argument, so `y`
209   must be a float. The value of `x`, as argument of integer `+`,
210   must be an `int`. Thus the top-level function takes `x`, an `int`,
211   and `y`, a `float`, to the result of integer addition, an `int`,
212   that is, `int -> float -> int`. *)
213
214 (*=====
215 Part 3: First-order functional programming
216
217 For warmup, here are some "finger exercises" defining simple functions
218 before moving onto more complex problems.
219
220 .....
221 Exercise 7: Define a function `square` that squares its
222 argument. We've provided a bit of template code, supplying the first
223 line of the function definition but the body of the skeleton code just
224 causes a failure by forcing an error using the built-in `failwith`
225 function. Edit the code to implement `square` properly.
226
227 Test out your implementation of `square` by modifying the template
228 code below to define `exercise7` to be the `square` function applied

```

```

229 to the integer 5. You'll want to replace the `0` with the correct
230 function call.

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

239
240
241     % ocamlbuild -use-ocamlfind lab1_tests.byte
242     % ./lab1_tests.byte
243
244 You should add some tests for other functions in the lab to get some
245 practice with automated unit testing.
246 .....*)
247
248 let square (x : int) : int  =
249     x * x ;;
250
251 let exercise7 =
252     square 5 ;;
253
254 (*.....
255 Exercise 8: Define a function `exclaim` ,that, given a string,
256 "exclaims" it by capitalizing it and suffixing an exclamation mark.
257 The `String.capitalize_ascii` function may be helpful here. For
258 example, you should get the following behavior:
259
260     # exclaim "hello" ;;
261     - : string = "Hello!"
262     # exclaim "Ciao" ;;
263     - : string = "Ciao!"
264     # exclaim "what's up" ;;
265     - : string = "What's up!"
266 .....*)
267
268 let exclaim (text : string) : string =
269     (String.capitalize_ascii text) ^ "!" ;;
270
271 (*.....
272 Exercise 9: Define a function `needs_small_bills` that determines, given a
273 price, if one will need a bill smaller than a 20 to pay for the
274 item. For instance, a price of 100 can be paid for with 20s (and

```

```

275 larger denominations) alone, but a price of 105 will require a bill
276 smaller than a 20 (for the 5 left over after the 100 is paid). We will
277 assume (perhaps unrealistically) that all prices are given as integers
278 and (more realistically) that 50s, 100s, and larger denomination bills
279 are not available, only 1s, 5s, 10s, and 20s. In addition, you may
280 assume all prices given are non-negative.

281
282     # needs_small_bills 105 ;;
283     - : bool = true
284
285     # needs_small_bills 100 ;;
286     - : bool = false
287
288     # needs_small_bills 150 ;;
289     - : bool = true
290
291     .....*)
```

290 **let** needs_small_bills (price : int) : bool =
291 **let** cutoff_size = 20 **in**
292 (price mod cutoff_size) <> 0 ;;

293
294 (* Note the use of '<>' for inequality. You may have used '==' or '!='
295 for comparing integer values for equality or inequality. OCaml
296 distinguishes two kinds of equality comparisons: *structural* and
297 *physical* equality. More on the distinction later in the course
298 (Section 15.1.2 in the textbook for the interested), but for now,
299 it is sufficient to note that you should be using the *structural*
300 equality operators: '=' for equality and '<>' for inequality. These
301 differ from the operators in C and Python (which use '==' and
302 '!='); these are used as the physical equality operators in OCaml.

303
304 We also labeled 20 our 'cutoff_size'. It is generally good practice
305 to name special numbers used in code. If used many times, they
306 should be defined only in one location, so that changing them is
307 easy, but even if used only once, explicit naming provides for
308 better documentation. See the style guide section on "Constants and
309 magic numbers". *)

310 (*.....

312 Exercise 10:

313
314 The calculation of the date of Easter, a calculation so important to
315 early Christianity that it was referred to simply by the Latin
316 "computus" ("the computation"), has been the subject of innumerable
317 algorithms since the early history of the Christian church.

318
319 The algorithm to calculate the computus function is given in Problem
320 31 in the textbook, which you'll want to refer to.

```

321
322 Write two functions that, given a year, calculate the month
323 ('computus_month') and day ('computus_day') of Easter in that year via
324 the Computus function.
325
326 In 2018, Easter took place on April 1st. Your functions should reflect
327 that:
328
329     # computus_month 2018;;
330     - : int = 4
331     # computus_day 2018 ;;
332     - : int = 1
333     .....*)
334
335 (* SOLUTION: You might have implemented a `computus_month` function like
336 this:
337
338     let computus_month (year : int) : int =
339         let a = year mod 19 in
340         let b = year / 100 in
341         let c = year mod 100 in
342         let d = b / 4 in
343         let e = b mod 4 in
344         let f = (b + 8) / 25 in
345         let g = (b - f + 1) / 3 in
346         let h = (19 * a + b - d - g + 15) mod 30 in
347         let i = c / 4 in
348         let k = c mod 4 in
349         let l = (32 + 2 * e + 2 * i - h - k) mod 7 in
350         let m = (a + 11 * h + 22 * l) / 451 in
351         (h + l - 7 * m + 114) / 31 ;;
352
353 and then just made a copy and modified it to form the `computus_day`
354 function:
355
356     let computus_day (year : int) : int =
357         let a = year mod 19 in
358         let b = year / 100 in
359         let c = year mod 100 in
360         let d = b / 4 in
361         let e = b mod 4 in
362         let f = (b + 8) / 25 in
363         let g = (b - f + 1) / 3 in
364         let h = (19 * a + b - d - g + 15) mod 30 in
365         let i = c / 4 in
366         let k = c mod 4 in

```

```

367     let l = (32 + 2 * e + 2 * i - h - k) mod 7 in
368     let m = (a + 11 * h + 22 * l) / 451 in
369     (h + l - 7 * m + 114) mod 31 + 1;;
370
371 However, the first twelve equations are shared between the
372 computations for the month and the day, so it makes sense to split
373 this common part out as its own function (an application of the
374 edict of irredundancy): *)
375
376 let computus_common (year : int) : int =
377   let a = year mod 19 in
378   let b = year / 100 in
379   let c = year mod 100 in
380   let d = b / 4 in
381   let e = b mod 4 in
382   let f = (b + 8) / 25 in
383   let g = (b - f + 1) / 3 in
384   let h = (19 * a + b - d - g + 15) mod 30 in
385   let i = c / 4 in
386   let k = c mod 4 in
387   let l = (32 + 2 * e + 2 * i - h - k) mod 7 in
388   let m = (a + 11 * h + 22 * l) / 451 in
389   h + l - 7 * m + 114;;
390
391 (* The `computus_month` and `computus_day` functions can then be
392 implemented on the basis of that common calculation. *)
393
394 let computus_month (year : int) : int =
395   (computus_common year) / 31;;
396
397 let computus_day (year : int) =
398   (computus_common year) mod 31 + 1;;
399
400 (* Even more redundancy can be eliminated in the computation by taking
401 advantage of structured data, a topic that is coming up in the next
402 lab. You might note that many of the equations come in pairs -- 'b'
403 and 'c', 'd' and 'e', 'i' and 'k', and even the final two
404 calculations of the month and year -- with one of the equations a
405 division ('x / y'), and the other the remainder of that division
406 ('x mod y'). By defining a function to calculate both of these
407 values, and returning them as a pair, we can eliminate this
408 redundancy, and reduce the possibility that a change to one of the
409 pairs doesn't get reflected in the other. That's another
410 application of the edict of irredundancy.
411
412 let divmod (x : int) (y : int) : int * int =

```

```

413     x / y, x mod y;;
414
415 let computus_common (year : int) : int * int =
416     let a = year mod 19 in
417     let b, c = divmod year 100 in
418     let d, e = divmod b 4 in
419     let f = (b + 8) / 25 in
420     let g = (b - f + 1) / 3 in
421     let h = (19 * a + b - d - g + 15) mod 30 in
422     let i, k = divmod c 4 in
423     let l = (32 + 2 * e + 2 * i - h - k) mod 7 in
424     let m = (a + 11 * h + 22 * l) / 451 in
425     let month, previous_day = divmod (h + l - 7 * m + 114) 31
426     in month, previous_day + 1;;
427
428 let computus_month (year : int) : int =
429     fst (computus_common year);;
430
431 let computus_day (year : int) : int =
432     snd (computus_common year);;
433
434 Exercise to the reader: Why is the below implementation not ideal?
435
436 let computus_month (year : int) : int =
437     let january = year mod 19 in
438     let february = year / 100 in
439     let march = year mod 100 in
440     let april = february / 4 in
441     let may = february mod 4 in
442     let june = (february + 8) / 25 in
443     let july = (february - june + 1) / 3 in
444     let august = (january * 19 + february - april - july + 15) mod 30 in
445     let september = march / 4 in
446     let october = march mod 4 in
447     let november = (32 + 2 * may + 2 * september - august - october) mod 7 in
448     let december = (january + 11 * august + 22 * november) / 451 in
449     (august + november - 7 * december + 114) / 31;;
450 *)
451
452 (*=====
453 Part 4: Code review
454
455 A frustum (see Figure 6.3 in the textbook) is a three-dimensional
456 solid formed by slicing off the top of a cone parallel to its
457 base. The formula for the volume of a frustum in terms of its radii
458 and height is given in the textbook as well.

```

```

459
460 As an experienced programmer at Frustumco, Inc., you've been assigned
461 to mentor a beginning programmer. Your mentee has been given the task
462 of implementing a function `frustum_volume` to calculate the volume
463 of a frustum. Here is your mentee's stab at this task:
464
465 (* frustum_volume -- calculate the frustum *)
466 let frustum_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 frustum
476 with radii 3 and 4 and height 4 -- and you get
477
478 # frustum_volume 3. 4. 4. ;;
479 - : float = 154.98559999999977
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/frustum>. 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/frustum>.
502 .....*)
503
504 (** Place your updated revised version below, *not* as a comment,

```

```

505     because we'll be unit testing it. (The two lines we provide are
506     just to allow the unit tests to have something to compile
507     against. You'll want to just delete them and start over.) ***)
508 (* SOLUTION: There are, of course, lots of problems with the original
509     code; it's almost completely unreadable and obscure. Let's start by
510     at least adding white space -- line breaks and indentation -- to
511     make the structure of the function clear:
512
513     let frustum_volume a b c =
514         let a =
515             let s a = a * a in
516             let h = b in
517                 (3.1416 *. h /. float_of_int 3)
518                 *. (a *. a +. c *. c + .a *. c) in
519             a ;;
520
521     That's already much better. It makes more clear that the radii are
522     'a' and 'c' and the height is 'b'. (Maybe that's why the programmer has
523     the 'let h = b' renaming.) Better variable names are badly needed,
524     as well as a better order for arguments. (That'll allow for
525     dropping the 'let h = b', too.) We should also mark the intended
526     types for the arguments and for the return value:
527
528     let frustum_volume (radius1 : float)
529                     (radius2 : float)
530                     (height : float)
531                     : float =
532         let a =
533             let s a = a * a in
534                 (3.1416 *. height /. float_of_int 3)
535                 *. (radius1 *. radius1 +. radius2 *. radius2 +. radius1 *. radius2) in
536         a ;;
537
538     Now, what is this local function 's' that is being defined? It looks
539     like a squaring function, which might have been useful in calculating
540     the squares of the radii, but apparently that idea got dropped. We can
541     drop the definition as well. The code also defines the answer in a
542     local variable 'a' (for answer?), which it just returns. There's no
543     reason to name the return value in that way.
544
545     let frustum_volume (radius1 : float)
546                     (radius2 : float)
547                     (height : float)
548                     : float =
549             (3.1416 *. height /. float_of_int 3)
550             *. (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 frustum_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 (* frustum_volume radius1 radius2 height -- Returns the volume of a
575    conical frustum given the radii of the two faces ('radius1` and
576    `radius2`) and the perpendicular 'height` *)
577
578 let frustum_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

```

```

597 will be non-negative.

598 For example,
600
601 # factorial 4;;
602 - : int = 24
603 # factorial 0;;
604 - : int = 1
605 .....*)

606
607 let rec factorial (x : int) =
608   if x = 0 then 1
609   else x * factorial (x - 1);;

610 (* The above code is what we expected people to produce. However,
611    this code will run forever when the input is negative. Better
612    practice would be to raise an error, as below, when we encounter
613    an invalid input. You'll learn more about this issue in Lab 4.
614
615 let rec factorial (x : int) =
616   if x < 0 then raise (Invalid_argument "fact: arg must be non-negative")
617   else if x = 0 then 1
618   else x * factorial (x - 1);;
619
620 *)
621
622 (*.....
623 Exercise 14: Define a recursive function `sum_from_zero` that sums all
624 the integers between 0 and its argument, inclusive.
625
626 # sum_from_zero 5;;
627 - : int = 15
628 # sum_from_zero 100;;
629 - : int = 5050
630 # sum_from_zero ~-3;;
631 - : int = -6
632
633 (* The sum from 0 to 100 was famously if apocryphally performed by
634    the mathematician Carl Friedrich Gauss as a seven-year-old, *in his
635    head*!)
636
637 (* Here's an approach that works recursively. The recursive cases for
638    positive and negative numbers are handled separately, since in the
639    former case we want to count down (using the built-in `pred`
640    function) and for negatives we want to count up toward zero (using
641    `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 (* You may notice that there's a lot of similarity between the 'then'
649   and 'else' branches. We could factor out the similarities by
650   narrowing the scope of the conditional test inside, and use it just
651   for selecting whether to use the function 'succ' or 'pred'. The
652   result is this:
653
654   let rec sum_from_zero (x : int) : int =
655     if x = 0 then 0
656     else x + sum_from_zero (if x < 0 then succ x else pred x) ;;
657
658 or even this
659
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?) *)

```