

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

```

45
46 ***** We use the commenting convention in our code throughout the
47 course that code snippets within comments are demarcated with
48 backquotes, for instance, `x + 3` or `fun x -> x`. You can think
49 of this as corresponding to the fixed-width font in the textbook.
50 ****
51
52
53 .....
54 Exercise 2: So that you can see how the unit tests in labs work,
55 replace the `failwith` expression below with the integer `42`, so that
56 `exercise2` is a function that returns `42` (instead of failing). When
57 you submit, the Exercise 2 unit test should then pass.
58 .....*)
59
60 let exercise2 () = failwith "exercise2 not implemented" ;;
61
62 (* From here on, you'll want to test your lab solutions locally before
63 submitting them at the end of lab to Gradescope. A simple way to do that
64 is to cut and paste the exercises into an OCaml interpreter, such as
65 utop, which you run with the command
66
67 % utop
68
69 You can also use the more basic version, ocaml:
70
71 % ocaml
72
73 We call this kind of interaction a "read-eval-print loop" or
74 "REPL". Alternatively, you can feed the whole file to OCaml with the
75 command:
76
77 % ocaml < lab1.ml
78
79 to see what happens. We'll introduce other methods soon. *)
80
81 (======
82 Part 1: Concrete versus abstract syntax
83
84 We've distinguished concrete from abstract syntax. Abstract syntax
85 corresponds to the substantive tree structuring of expressions;
86 concrete syntax corresponds to the particulars of how those structures
87 are made manifest in the language's textual notation.
88
89 In the presence of multiple operators, issues of precedence and
90 associativity become important in constructing the abstract syntax

```

```

91   from the concrete syntax.

92
93   .....
94 Exercise 3: Consider the following abstract syntax tree:

95
96     ~-
97     |
98     |
99     -
100    ^
101    / \
102    /   \
103    5     3

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

109
110 How might this *abstract* syntax be expressed in the *concrete* syntax
111 of OCaml using the fewest parentheses? Replace the `failwith`
112 expression with the appropriate OCaml expression to assign the value
113 to the variable `exercise3` below.

114 .....*)

115
116
117 let exercise3 () : int = failwith "exercise3 not implemented" ;;
118
119 (* Hint: The OCaml concrete expression `~- 5 - 3` does *not*
120 correspond to the abstract syntax above.

121
122 .....
123 Exercise 4: Draw the tree that the concrete syntax `~- 5 - 3` does
124 correspond to. Check it with a member of the course staff.

125 .....*)

126
127
128 (*.....
129 Exercise 5: Associativity plays a role in cases when two operators
130 used in the concrete syntax have the same precedence. For instance,
131 the concrete expression `2 + 1 + 0` might have abstract syntax as
132 reflected in the following two parenthesizations:

133
134     2 + (1 + 0)
135
136 or

```

```

137
138     (2 + 1) + 0
139
140 As it turns out, both of these parenthesizations evaluate to the same
141 result ('3'). (That's because addition is an associative operation.)
142
143 Construct an expression that uses an arithmetic operator twice, but
144 evaluates to two different results dependent on the associativity of
145 the operator. Use this expression to determine the associativity of
146 the operator. Check your answer with a member of the course staff if
147 you'd like.
148 .....*)
```

149

```

150 (*=====
151 Part 2: Types and type inference
152
153 .....
154 Exercise 6: What are appropriate types to replace the ??? in the
155 expressions below? Test your solution by uncommenting the examples
156 (removing the `(*)` and `*)` at start and end) and verifying that no
157 typing error is generated.
158 .....*)
```

159

```

160 (* <-- After you've replaced the ???s, remove this start-comment line...
161
162 let exercise6a : ??? = 42 ;;
163
164 let exercise6b : ??? =
165   let greet y = "Hello " ^ y
166   in greet "World!";;
167
168 let exercise6c : ??? =
169   fun x -> x +. 11.1 ;;
170
171 let exercise6d : ??? =
172   fun x -> x < x + 1 ;;
173
174 let exercise6e : ??? =
175   fun x -> fun y -> x + int_of_float y ;;
176
177 ...and remove this whole end-comment line too. --> *)
```

178

```

179 (*=====
180 Part 3: First-order functional programming
181
182 For warmup, here are some "finger exercises" defining simple functions
```

```

183 before moving onto more complex problems.

184
185 .....
186 Exercise 7: Define a function `square` that squares its
187 argument. We've provided a bit of template code, supplying the first
188 line of the function definition but the body of the skeleton code just
189 causes a failure by forcing an error using the built-in `failwith`
190 function. Edit the code to implement `square` properly.

191
192 Test out your implementation of `square` by modifying the template
193 code below to define `exercise7` to be the `square` function applied
194 to the integer 5. You'll want to replace the `0` with the correct
195 function call.

196
197 Thorough testing is important in all your work, and we hope to impart
198 this view to you in CS51. Testing will help you find bugs, avoid
199 mistakes, and teach you the value of short, clear, testable
200 functions. In the file `lab1_tests.ml`, we've put some prewritten
201 tests for `square` using the testing method of Section 6.5 in the
202 book. Spend some time understanding how the testing function works and
203 why these tests are comprehensive. Then test your code by compiling
204 and running the test suite:

205
206     % ocamlbuild -use-ocamlfind lab1_tests.byte
207     % ./lab1_tests.byte
208
209 You should add some tests for other functions in the lab to get some
210 practice with automated unit testing.
211 .....
212
213 let square (x : int) : int =
214     failwith "square not implemented" ;;
215
216 let exercise7 = 0 ;;
217
218 (*.....
219 Exercise 8: Define a function `exclaim`, that, given a string,
220 "exclaims" it by capitalizing it and suffixing an exclamation mark.
221 The `String.capitalize_ascii` function may be helpful here. For
222 example, you should get the following behavior:
223
224     # exclaim "hello" ;;
225     - : string = "Hello!"
226     # exclaim "Ciao" ;;
227     - : string = "Ciao!"
228     # exclaim "what's up" ;;

```

```

229     - : string = "What's up!"
230     .....*)
231
232 let exclaim (text : string) : string =
233   failwith "exclaim not implemented";;
234
235 (*.....
236 Exercise 9: Define a function `needs_small_bills` that determines, given a
237 price, if one will need a bill smaller than a 20 to pay for the
238 item. For instance, a price of 100 can be paid for with 20s (and
239 larger denominations) alone, but a price of 105 will require a bill
240 smaller than a 20 (for the 5 left over after the 100 is paid). We will
241 assume (perhaps unrealistically) that all prices are given as integers
242 and (more realistically) that 50s, 100s, and larger denomination bills
243 are not available, only 1s, 5s, 10s, and 20s. In addition, you may
244 assume all prices given are non-negative.
245
246 # needs_small_bills 105;;
247 - : bool = true
248 # needs_small_bills 100;;
249 - : bool = false
250 # needs_small_bills 150;;
251 - : bool = true
252 .....*)
253
254 let needs_small_bills (price : int) : bool =
255   failwith "needs_small_bills not implemented" ;;
256
257 (*.....
258 Exercise 10:
259
260 The calculation of the date of Easter, a calculation so important to
261 early Christianity that it was referred to simply by the Latin
262 "computus" ("the computation"), has been the subject of innumerable
263 algorithms since the early history of the Christian church.
264
265 The algorithm to calculate the computus function is given in Problem
266 31 in the textbook, which you'll want to refer to.
267
268 Write two functions that, given a year, calculate the month
269 ('computus_month') and day ('computus_day') of Easter in that year via
270 the Computus function.
271
272 In 2018, Easter took place on April 1st. Your functions should reflect
273 that:
274
```

```

275      # computus_month 2018;;
276      - : int = 4
277      # computus_day 2018 ;;
278      - : int = 1
279      .....*)
280
281  let computus_month (year : int) : int =
282    failwith "computus_month not implemented" ;;
283  let computus_day (year : int) : int =
284    failwith "computus_day not implemented" ;;
285
286 (*=====
287 Part 4: Code review
288
289 A frustrum (see Figure 6.3 in the textbook) is a three-dimensional
290 solid formed by slicing off the top of a cone parallel to its
291 base. The formula for the volume of a frustrum in terms of its radii
292 and height is given in the textbook as well.
293
294 As an experienced programmer at Frustumco, Inc., you've been assigned
295 to mentor a beginning programmer. Your mentee has been given the task
296 of implementing a function `frustrum_volume` to calculate the volume
297 of a frustrum. Here is your mentee's stab at this task:
298
299 (* frustrum_volume -- calculate the frustrum *)
300 let frustrum_volume a b c =
301   let a =
302     let s a = a * a in
303     let h = b in 3.1416
304     *. h /. float_of_int 3*. (a *.
305     a +. c *. c+.a *. c) in a
306   ;;
307
308 As this neophyte programmer's mentor, you're asked to perform a code
309 review on this code. You test the code out on an example -- a frustrum
310 with radii 3 and 4 and height 4 -- and you get
311
312   # frustrum_volume 3. 4. 4. ;;
313   - : float = 154.98559999999977
314
315 which is (more or less) the right answer. Nonetheless, you have a
316 strong sense that the code can be considerably improved. *)
317
318 (*.....
319 Exercise 11: Go over the code with your lab partner, making whatever
320 modifications you think can improve the code, placing your revised

```

```

321 version just below. Once you've converged on a version of the code
322 that you think is best, call over a staff member and go over your
323 revised code together.
324 .....*)
325
326 (** Place your revised version here within this comment. **)
327
328 (* During the code review, your boss drops by and looks over your
329 proposed code. Your boss thinks that the function should be compatible
330 with the header line given at <https://url.cs51.io/frustrum>. You
331 agree.
332
333 .....
334 Exercise 12: Revise your code (if necessary) to make sure that it uses
335 the header line given at <https://url.cs51.io/frustrum>.
336 .....*)
337
338 (** Place your updated revised version below, *not* as a comment,
339 because we'll be unit testing it. (The two lines we provide are
340 just to allow the unit tests to have something to compile
341 against. You'll want to just delete them and start over.) **)
342 let frustum_volume _ _ =
343   failwith "frustum_volume not implemented" ;;
344 (*=====
345 Part 5: Utilizing recursion
346
347 .....
348 Exercise 13: The factorial function takes the product of an integer
349 and all the integers below it. It is generally notated as !. For
350 example,  $4! = 4 * 3 * 2 * 1$ . Write a function `factorial` that
351 calculates the factorial of its integer argument. Note: the factorial
352 function is generally only defined on non-negative integers (0, 1, 2,
353 3, ...). For the purpose of this exercise, you may assume all inputs
354 will be non-negative.
355
356 For example,
357
358 # factorial 4 ;;
359 - : int = 24
360 # factorial 0 ;;
361 - : int = 1
362 .....*)
363
364 let factorial (x : int) : int =
365   failwith "factorial not implemented" ;;
366

```

```
367 (*.....  
368 Exercise 14: Define a recursive function `sum_from_zero` that sums all  
369 the integers between 0 and its argument, inclusive.  
370  
371 # sum_from_zero 5 ;;  
372 - : int = 15  
373 # sum_from_zero 100 ;;  
374 - : int = 5050  
375 # sum_from_zero ~-3 ;;  
376 - : int = -6  
377  
378 (The sum from 0 to 100 was famously if apocryphally performed by  
379 the mathematician Carl Friedrich Gauss as a seven-year-old, *in his  
380 head!*)  
381 .....*)  
382  
383 let sum_from_zero (x : int) : int =  
384   failwith "sum_from_zero not implemented" ;;
```