```
(*
                              CS51 Lab D
                        Improving Debugging Skills
 *)


(*======================================================================
Readings:

    This lab has no prerequisite textbook reading.

Objective: In this lab, you'll improve your debugging skills by
applying fundamental debugging ideas to improving code.

    In the exercises that follow, some functions may have bugs, so that
    their behavior may not match the intended behavior described in the
    comments. Your job is to find and fix all of the bugs.
======================================================================*)


(*======================================================================
Part 0: Important aspects of debugging

You may not have thought explicitly about the debugging process, but
doing so can provide you with valuable skills in the process. Here are
some of the major aspects of the debugging process.

    Identification

        Read error messages in detail. They often provide not just the
        nature of the error, but an approximate location.

        Set up unit tests for individual functions. Unit tests can
        identify bugs in your code by finding cases that don't match
        the behavior you intended. Try to specify unit test cases that
        cover all of the important paths through the code. A good
        technique is to put the unit tests in a separate file that
        references the file with the functions to be tested. Then,
        whenever you make changes to the functions, you can rerun the
        test file to make sure that you haven't introduced bugs in
        previously working code.

    Localization

        When you first identify a bug, you may not know where in the
        code base the bug actually lives. You'll need to localize the
```

1

bug -- finding its location in the code base.

In tracking down problems in larger codebases, eliminate portions of the code to generate the minimal codebase that demonstrates the problem. Breaking the code into smaller parts can allow localization to one of the parts, as they can be unit-tested separately.

Simplification

When confronted with an error exhibited on a large instance, try to simplify it to find the minimal example that exhibits the problem.

Reproduction

Try alternate examples to see which ones exhibit the problem. The commonalities among the examples that exhibit the problem can give clues as to the problem.

Diagnosis

Verify that invariants that should hold in the code actually do, with assertions or other constructs. (The `Absbook.verify` function can be especially useful in verifying invariants of the arguments and return value.) Conduct experiments to test your theory of what has gone wrong.

Correction

Generate git commits to save a version of the code so that you can confidently make changes to the code while you are experimenting, knowing that you'll be able to return to earlier versions.

Maintenance

Code that was once working can become buggy as changes are made either to the code itself or to code that it uses. It's thus helpful to retest code when changes are made to it or its environment. Fortunately, unit test files are ideal for this process. Rerunning the unit tests liberally allows us to verify that working code hasn't regressed to a buggy state. (The process is referred to in the literature as "regression testing" for this reason. See <https://en.wikipedia.org/wiki/Regression_testing>.)

```
 91
 92    *)
 93
 94   (*===================================================================
 95   Part 1: Some finger exercises in debugging
 96
 97   In this part, we'll provide implementations of a few simple
 98   functions. These functions may or may not have bugs in them. You'll
 99   proceed through four steps:
100
101     1. Read the function definition, including the top-level comment, to
102        give you an idea of what the function is intended to do.
103
104     2. Write a full set of unit tests for each of the functions in the
105        file `part1_tests.ml`. The comments introducing each function may
106        give information about the intended behavior of each function.
107
108     3. Once you've written all of the unit tests, compile and run the
109        unit tests
110
111          % ocamlbuild -use-ocamlfind part1_tests.byte
112          % ./part1_tests.byte
113
114     4. For each of the functions, find a value for the function's
115        argument that expresses the bug, if there is one. (If you've built
116        your unit tests well, they should uncover such a value directly.)
117        Record the bug-inducing value by let-defining the corresponding
118        `-bug` value to be `Some v` where `v` is the bug-inducing value if
119        there is one, or `None` if the function is not buggy.
120
121     5. Revise each function to eliminate any bugs that you found. While
122        you're at it, you should probably deal with any warnings that
123        arise as well.
124
125   We've done the first of these exercises, the `abs` function, for you
126   to give you the idea.
127
128        1. ORIGINAL VERSION
129
130        (* abs x -- Returns the absolute value of the integer `x` *)
131        let abs x =
132          if x < ~-1 then ~- x else x
133
134        2. UNIT TESTS (These would be added to `part1_tests.ml`.)
135
136        unit_test (abs 0 = 0) "abs zero";
```

3

```
137     unit_test (abs 1 = 1) "abs one";
138     unit_test (abs max_int = max_int) "abs maxint";
139     unit_test (abs min_int = min_int) "abs minint";
140     unit_test (abs (-0) = 0) "abs zero";
141     unit_test (abs (-1) = 1) "abs neg one";
142     unit_test (abs (-max_int) = max_int) "abs neg maxint";
143     unit_test (abs (-min_int) = min_int) "abs neg minint";
144
145     3. COMPILE AND RUN
146
147     % ocamlbuild -use-ocamlfind part1_tests.byte
148     % ./part1_tests.byte
149     ...
150     abs zero passed
151     abs one passed
152     abs maxint passed
153     abs minint passed
154     abs zero passed
155     abs neg one FAILED        <-- This'll be helpful!
156     abs neg maxint passed
157     abs neg minint passed
158     ...
159     - : unit = ()
160
161     3. BUGGY VALUE
162
163     let abs_bug = Some (-1)
164
165     4. REVISED VERSION
166
167     (* abs x -- Returns the absolute value of the integer `x` *)
168     let abs x =
169        if x < 0 then ~- x else x
170  *)
171
172 (*................................................................*)
173 (* last_element lst -- Returns the last element of `lst` as an option;
174    `None` if there is no last element *)
175 let rec last_element lst =
176   match lst with
177   | [] -> None
178   | [x] -> Some x
179   | _ :: tail -> last_element (List.tl tail)
180
181 let last_element_bug = failwith "last_element_bug not defined"
182
```

4

```
183  (*....................................................................*)
184  (* sum_to_n n -- Returns the sum of integers from 1 to `n` *)
185  let rec sum_to_n n =
186    if n = 0 then n
187    else n + sum_to_n (n - 1)
188
189  let sum_to_n_bug = failwith "sum_to_n_bug not defined"
190
191  (*....................................................................*)
192  let describe_list lst =
193    match lst with
194    | [] -> "Empty list"
195    | _ :: _ -> "Multiple list"
196    | [x] -> "Singleton list"
197
198  let describe_list_bug = failwith "describe_list_bug not defined"
199
200  (*=====================================================================
201  Part 2: Debugging set operations
202
203  In this part, you will apply your debugging skills to debugging an
204  implementation of set operations (union, intersection, etc.).
205
206      **************************************************************
207      In this lab, sets of integers will be represented as `int list`s
208      whose elements are in *sorted order* with *no duplicates*. All
209      functions can assume this invariant and should deliver results
210      satisfying it as well.
211      **************************************************************
212
213  To get you started on debugging, we've placed a few unit tests for
214  some of the functions in the file `part2_tests.ml`. Compile and run
215  these tests to see how the functions are working so far.
216
217      % ocamlbuild -use-ocamlfind part2_tests.byte
218      % ./part2_tests.byte
219
220  What do you notice? Does this give you an idea on where to start
221  debugging?
222
223  ======================================================================
224  Part 2A: Some utilities for checking the sorting and no-duplicates
225  conditions.
226   *)
227
228  (* is_sorted lst -- Returns `true` if and only if `lst` is a sorted
```

```
229      list *)
230  let is_sorted (lst : 'a list) : bool =
231    lst = List.sort Stdlib.compare lst ;;
232
233  (* dups_sorted lst -- Returns the number of duplicate elements in
234     `lst`, a sorted list of integers. For example
235
236         # dups_sorted [1;2;5;5;5;5;5;5;6;7;7;9] ;;
237         - : int = 6
238   *)
239  let rec dups_sorted (lst : 'a list) : int =
240    match lst with
241    | [] -> 0
242    | [_] -> 0
243    | first :: second :: rest ->
244        if first = second then 1 else 0
245        + dups_sorted rest
246
247  (* HINT: The `dups_sorted` function already fails two of the tests in
248     the testing file `part2_tests.ml`. Examine the failing cases one at
249     a time. What is it about the first case that causes it to fail? Can
250     you find a simpler case that fails? What is the minimal example
251     that fails? Does that help you repair the first buggy test case?
252     Then turn your attention to the second case. Did you fix that bug
253     too, or is it still around? If it's still around, you'll need to
254     fix that too. *)
255
256  (* is_set lst -- Returns `true` if and only if lst represents a set,
257     with no duplicates and elements in sorted order. *)
258  let is_set (lst : 'a list) : bool =
259    is_sorted lst && dups_sorted lst = 0 ;;
260
261  (*======================================================================
262  Part 2B: Set operations -- member, union, and intersection
263
264  Below we provide code for computing membership, intersections, and
265  unions of sets represented by lists with the stated invariant.
266
267  Check out the unit tests for these in `part2_tests.ml`. Augment the
268  tests until you're satisfied that you've fully tested these functions,
269  making any needed changes as you go.
270
271  We'll test them further on larger examples in the next part, Part
272  2C. *)
273
274  (* member elt set -- Returns `true` if and only if `elt` is an element
```

```ocaml
     of `set` (represented as above). Search can stop early based on
     sortedness of `set`. *)
let rec member elt set =
  match set with
  | [] -> false
  | hd :: tl ->
      if elt = hd then true
      else if elt < hd then false
      else member elt tl ;;

(* union set1 set2 -- Returns a list representing the union of the
   sets `set1` and `set2` *)
let rec union s1 s2 =
  match s1, s2 with
  | [], [] -> []
  | _hd :: _tl, [] -> s1
  | [], _hd :: _tl -> s2
  | hd1 :: tl1, hd2 :: tl2 ->
      if hd1 < hd2 then
        hd1 :: union tl1 s2
      else
        hd2 :: union tl2 s1 ;;

(* HINT: This function has no tests in the testing file. Maybe you
should add some. *)

(* intersection set1 set2 -- Returns a list representing the
   intersection of the sets `set1` and `set2` *)
let rec intersection s1 s2 =
  match s1, s2 with
  | [], _ -> []
  | _, [] -> []
  | hd1 :: tl1, hd2 :: tl2 ->
      if hd1 = hd2 then hd1 :: intersection tl1 tl2
      else if hd1 > hd2 then intersection tl1 s2
      else intersection s1 tl2 ;;

(*======================================================================
Part 2C: Scaling up the testing

The file `labD_examples` contains a couple of larger examples of sets
represented as lists (`example1` and `example2`). The `part2_tests.ml`
file contains a few tests based on these larger examples, which are
commented out at the moment. Uncomment them now and rerun the unit
tests. What do you notice?
```

```
321   More bugs to debug. Where do you think the problems lie? Remaining
322   bugs in the functions above? In the examples? In the tests themselves?
323
324   You're on your own to figure out what's going on and correct the
325   problems, wherever they might be.
326    *)
```