

(*)

CS51 Lab D
Improving Debugging Skills

*)

(* Objective: In this lab, you'll improve your debugging skills by applying fundamental debugging ideas to debugging an implementation of set operations (union, intersection, etc.).

```
*****
In this lab, sets of integers will be represented as `int
list`s whose elements are in sorted order with no
duplicates. All functions can assume this invariant and should
deliver results satisfying it as well.
*****
```

In the code that follows, 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 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.)

To get you started on debugging, we've placed a few unit tests for some of the functions in the file `labD_tests.ml`. Compile and run these tests to see how the functions are working so far.

```
% ocamlbuild -use-ocamlfind labD_tests.byte
% ./labD_tests.byte
```

What do you notice? Does this give you an idea on where to start debugging?

=====
Part 1: Some utilities for checking the sorting and no-duplicates conditions.

*)

```
(* is_sorted lst -- Returns 'true' if and only if 'lst' is a sorted list *)
```

```
let is_sorted (lst : 'a list) : bool =
  lst = List.sort Stdlib.compare lst ;;
```

```
(* dups_sorted lst -- Returns the number of duplicate elements in 'lst', a sorted list of integers *)
```

```
let rec dups_sorted (lst : 'a list) : int =
  match lst with
  | [] -> 0
  | [_] -> 0
  | first :: second :: rest ->
    if first = second then 1 else 0
    + dups_sorted rest
```

```
(* HINT: The 'dups_sorted' function already fails two of the tests in the testing file 'labD_tests.ml'. Examine the failing cases one at a time. What is it about the first case that causes it to fail? Can you find a simpler case that fails? What is the minimal example that fails? Does that help you repair the first buggy test case? Then turn your attention to the second case. Did you fix that bug too, or is it still around? If it's still around, you'll need to
```

```
fix that too. *)
```

```
(* is_set lst -- Returns 'true' if and only if lst represents a set,
   with no duplicates and elements in sorted order. *)
```

```
let is_set (lst : 'a list) : bool =
  is_sorted lst && dups_sorted lst = 0 ;;
```

```
(*=====
Part 2: Set operations -- member, union, and intersection
```

Below we provide code for computing membership, intersections, and unions of sets represented by lists with the stated invariant.

Check out the unit tests for these in 'labD_tests.ml'. Augment the tests until you're satisfied that you've fully tested these functions, making any needed changes as you go.

We'll test them further on larger examples in the next part, Part 3.

```
*)
```

```
(* member elt set -- Returns 'true' if and only if 'elt' is an element
   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 :: t11, hd2 :: t12 ->
    if hd1 < hd2 then
      hd1 :: union t11 s2
    else
      hd2 :: union t12 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 :: t11, hd2 :: t12 ->
    if hd1 = hd2 then hd1 :: intersection t11 t12
    else if hd1 > hd2 then intersection t11 s2
    else intersection s1 t12 ;;
```

```
(*=====
Part 3: Scaling up the testing
```

The file 'labD_examples' contains a couple of larger examples of sets represented as lists ('example1' and 'example2'). The 'labD_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?

More bugs to debug. Where do you think the problems lie? Remaining bugs in the functions above? In the examples? In the tests themselves?

You're on your own to figure out what's going on and correct the problems, wherever they might be.

*)