CS51 PROBLEM SET 1: CORE FUNCTIONAL PROGRAMMING

STUART M. SHIEBER

This problem set is an individual problem set, not a partner problem set. Please refer to the course collaboration policy for more information.

Introduction

The goal of this problem set is to introduce you to OCaml and the ideas underlying statically-typed functional languages.

We'll assume that you have successfully configured your development environment as specified by Problem Set 0. If not, you'll want to do that before getting started.

To download the problem set, accept the assignment here and follow any instructions. Now, you're ready to get started with the problem set. Open the directory `1`, and you will find a file (`ps1.ml`) that contains detailed instructions and scaffolding for all of the problems. In particular, it provides a stub for each function you are asked to implement. Typically, that stub merely raises an exception noting that the function has not yet been written – something like

```
let reversed = (fun _ -> failwith "reversed not implemented") ;;
```

You'll want to replace these stubs with a correct implementation of the function. Other parts of the problem set may have you perform other tasks to check your understanding of the material.

Here are some things to keep in mind as you complete the problem sets.

Style: Good style is an extremely important part of programming well. It is not only useful in making code more readable but also in helping identify syntactic and type errors in your functions. For this and all problem sets, your code should adhere to the CS51 Style Guide or CSCI E-51 Style Guide.

Compilation errors: In order to submit your work to the course grading server, *your solution must compile against our test suite.* The system will reject submissions that do not compile. If there are problems that you are unable to solve, you must still write a function that matches the expected type signature, or your code will not compile. When we provide stub code, that code will typically compile to begin with. If you are having difficulty getting your code to compile, please visit office hours or post on Piazza. *Emailing your homework to your TF or the Head TFs is* not *a valid substitute for submitting to the course grading server. Please start early.*

Testing: 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 functions. Problem 2b contains prewritten tests that use assert statements. Spend some time understanding how the `assert` statement works and why these tests are

Date: January 30, 2018.

comprehensive. *For each function in Problem 2, you must write tests that cover all code paths and corner cases.* To run your tests, run the `make` command in the directory with your `ps1.ml` and then run the compiled program by executing `./ps1_tests.byte`. If an assertion fails, an exception will be raised and printed. If the assertion passes, there will be no output.

You *must* write `assert` statements that thoroughly test the functionality of each of the remaining sections of Problem 2. You will write these tests in `ps1_test.ml`, which references your solution file. To get you started, example tests are provided for Problem 2b.

As this may be your first time writing tests, we require that you write the larger of: at least three tests per sub-problem, or as many tests as are required to ensure that your code is tested thoroughly.

Helper functions: Feel free to use helper functions to make your code cleaner, more modular, and easier to test.

1. Fun with types

1.1. **Determining types.** In `ps1.ml`, you will see several expressions labeled `prob0`, `prob1a`, `prob1b`, `prob1c` inside of a comment. For each one, determine its type, and enter it as a string value of the corresponding definition of `prob0_answer`, `prob1a_answer`, etc. replacing the "???". (We've done `prob)_answer` for you to get you started.)

1.2. **Type checking.** For problems 1.d–f, explain in a comment why each of the expressions doesn't type check, and provide a minimal fix by changing the type, modifying the expression, or both. More detail is provided in `ps1.ml`.

2. A series of tasks

In this section, you'll write a series of functions that perform simple manipulations over lists, strings, numbers, and booleans. See the comments in `ps1.ml` for the specifications. Please give the functions the names listed in the comments and enumerated in the list below, as they must be named specifically in order to compile against our automated unit tests.

You should not *use any library functions in Problem 2.* The best way to learn about the core language is to work directly with the core language features. Here are the type signatures of the functions you will be asked to implement.

```
(1)  reversed : int list -> bool
(2)  merge : int list -> int list -> int list
(3)  unzip : (int * int) list -> int list * int list
(4)  variance : float list -> float option
(5)  few_divisors : int -> int -> bool
(6)  concat_list : string -> string list -> string
(7)  to_run_length: char list -> (int * char) list
(8)  from_run_length: (int * char) list -> char list
```

Remember: your problem set is not complete if you have not written unit tests for your work.

3. Challenge problem: Permutations

On this and future problem sets, we often provide an additional problem or two for those who would like an extra challenge. These problems are for your edification only – we claim they'll improve your "karma" – and will not affect your grade. We encourage you to attempt the challenge problems only once you have done your best work on the rest of the problem set and successfully submitted a version.

Write a function to generate all of the permutations of a list of integers. It should have the following signature:

```
permutations : int list -> int list list
```

Please see `ps1.ml` for more details and hints. Do not worry about duplicates in the list. You may use the `List` library functions to solve this problem.

4. Troubleshooting

We provide a file `ps1.mli` that checks to make sure your functions in `ps1.ml` have the correct types. (We will discuss `.mli` files and the related concept of module signatures in some detail later in the course.) You *should not* modify `ps1.mli`. If you change the type of a function, you may see an error like:

```
Error: The implementation ps1.ml does not match the interface ps1.cmi:
       Values do not match:
         val reversed : int list -> int
       is not included in
         val reversed : int list -> bool
```

This error means that your function `reversed` has the wrong type. You should look back at your code and modify it so that it has the type specified in the problem set.

If you delete a function, you may see an error like:

```
Error: The implementation ps1.ml does not match the interface ps1.cmi:
       The value `reversed' is required but not provided
```

This error means that you are missing an implementation of the function `reversed`. If you are not able to get a particular function to compile, do not delete it. Instead, you can revert the function back to the stub implementation in the distribution code (you can look at previous commits on GitHub to see what the stub implementation was).

5. Submission

Before submitting, please estimate how much time you spent on the problem set by editing the line:

```
let minutes_spent_on_pset () : int = failwith "not provided" ;;
```

to replace the value of the function with an approximate estimate of how long (in minutes) the problem set took you to complete. For example, if you spent 6 hours on this assignment, you should change the line to:

```
let minutes_spent_on_pset () : int = 360
```

Make sure your code still compiles. Then, to submit the problem set, follow the instructions found in the CS51 Reference or CSCI E-51 Reference.