CSE 130 - Programming Assignment #1

OCaml

90 points

(see submission instructions below)

(click your browser's refresh button to ensure that you have the most recent version)

(Programming Assignment #1 FAQ)

Note: See this for instructions on starting OCaml in the ACS lab machines. To download and install OCaml version 4.02 on your home machines see the instructions here. Remember that this is only to enable you to play with the assignment at home: the final version turned in must work on the ACS Linux machines. While you can use windows to begin working with OCaml, the code you turn in must be that required for the Linux environment.

Code Documentation and General Requirements

Code for all programming assignments should be **well documented**. A working program with no comments will **receive only partial credit**. Documentation entails writing a description of each function/method, class/structure, as well as comments throughout the code to explain the program logic. Comments in OCaml are enclosed within (* *), and may be nested. It is understood that some of the exercises in this programming assignment require extremely little code and will not require extensive comments.

While few programming assignments pretend to mimic the "real" world, they may, nevertheless, contain some of the ambiguity that exists outside the classroom. If, for example, an assignment is amenable to differing interpretations, such that more than one algorithm may implement a correct solution to the assignment, it is incumbent upon the programmer to document not only the functionality of the algorithm (and more broadly his/her interpretation of the program requirements), but to articulate **clearly** the reasoning behind a particular choice of solution.

Assignment Overview

The overall objective of this assignment is for you to gain some hands-on experience with OCaml. All the problems require relatively little code ranging from 2 to 15 lines. If any function requires more than that, you can be sure that you need to rethink your solution. The assignment is in the files misc.ml, and test.ml that you need to download. The file contains several skeleton OCaml functions, with missing bodies, i.e. expressions, which currently contain the text `failwith "to be written"` . Your task is to replace the text in those files with the the appropriate OCaml code for each of those expressions.

Note: All the solutions can be done using the purely functional fragment of OCaml, using constructs covered in class, and most require the use of *recursion*. Solutions using imperative features such as

references, while loops or library functions will receive **no credit**.

It is a good idea to start this assignment early; ML programming, while quite simple (when you know how), often seems somewhat foreign at first, particularly when it comes to recursion and list manipulation.

Assignment Testing and Evaluation

Your functions/programs **must** compile and/or run on a **Linux** ACS machine (e.g. `ieng6.ucsd.edu` , as this is where the verification of your solutions will occur. While you may develop your code on any system, ensure that your code runs as expected on an ACS machine prior to submission. You should test your code in the directories from which the zip files (see below) will be created, as this will approximate the environment used for grading the assignment.

Most of the points, except those for comments and style, will be **awarded automatically, by evaluating your functions against a given test suite**. The fourth file, `test.ml` contains a very small suite of tests which gives you a flavor of of these tests. At any stage, by typing at the UNIX shell :

```
ocaml test.ml > log
```

you will get a report on how your code stacks up against the simple tests.

The last line of the file `log` **must contain the word "Compiled" otherwise you get a zero for the whole assignment**. If for some problem, you cannot get the code to compile, leave it as is with the `failwith ...`, with your partial solution enclosed below as a comment. **There will be no exceptions to this rule**. The second last line of the log file will contain your overall score, and the other lines will give you a readout for each test. You are encouraged to try to understand the code in `test.ml`, and subsequently devise your own tests and add them to `test.ml`, but you will not be graded on this.

Alternately, inside the OCaml shell, type (user input is in red):

```
# #use "test.ml";;
.
.
.
val it = (...,...) : int * int
```

and it should return a pair of integers, reflecting your score and the max possible score on the sample tests. If instead an error message appears, your code will receive a zero.

Submission Instructions

1. Create the zip file for submission

Your solutions to this assignment will be stored in separate files under a directory called `solution/`, inside which you will place the files: `misc.ml`. There should be **no other files in the directory**.

After creating and populating the directory as described above, create a zip file called `<LastName>_<FirstName>_cse130_pa1.zip` by going into the directory `solution` and executing the UNIX shell command:

```
zip <LastName>_<FirstName>_cse130_pa1.zip *
```

You can refer to an example submission file to compare with yours. Make sure that your zipped file's structure is the same as the example.

2. Test the zip file to check for its validity

Once you've created the zip file with your solutions, you will use the `validate_pa1` program to see whether your zip file's structure is well-formed to be inspected by our grading system by executing the UNIX shell command:

```
validate_pa1 <LastName>_<FirstName>_cse130_pa1.zip
```

The `validate_pa1` program will output `OK` if your zip file is well-formed and your solution is compiled. Otherwise, it will output some error messages. Before going to step 3, make sure that your zip file passes `validate_pa1` program. **Otherwise you get a zero for the whole assignment.** If you have any trouble with this, refer to the instructions in step 1.

3. Submit the zip file via the `turnin_pa1` program

Once your zip file passes the validation check by `validate_pa1`, you will use the `turnin_pa1` program to submit this file for grading by going into the directory `solution/` and executing the UNIX shell command:

```
turnin_pa1 <LastName>_<FirstName>_cse130_pa1.zip
```

The `turnin_pa1` program will provide you with a confirmation of the submission process; make sure that the size of the file indicated by `turnin_pa1` matches the size of your tar file. Note that you may submit multiple times, but your latest submission overwrites previous submissions, and will be the ONLY one we grade. If you submit before the assignment deadline, and again afterwards, we will count it as if you only submitted after the deadline.

Problem #0: Pre-test and Surveys

Christine Alvarado, Mia Minnes and their colleagues in the Computer Science & Engineering (CSE) department are conducting a research study of students' performance and experiences in CSE courses in order to improve curriculum and pedagogies in CSE to ensure the academic success of a broader range of students in CSE and a better experience for all students in CSE courses at UCSD. The questions in this problem all relate to this study.

(a) 10 points

Fill out the following IRB form: IRB

If you do not agree to participate in the study, we ask that you fill out the IRB and choose the appropriate

option in question 1.

(b) 10 points

Complete the following survey: Survey

You will get credit as long as you complete the survey.

(c) 10 points

Complete the following pre-test: Pre-test

You will get credit as long as you complete the pre-test (the correctness of your answers does not affect your credit)

IMPORTANT: You have to complete the pre-test in one sitting. You cannot restaart it. The pre-test should not take more than 10 or 15 minutes.

Problem #1: Digital Roots and Additive Persistence `(misc.ml)`

(a) 10 points

Now write an OCaml function `sumList : int list -> int` that takes an integer list `l` and returns the sum of the elements of `l` . Once you have implemented the function, you should get the following behavior at the OCaml prompt:

```
# sumList [1;2;3;4];;
- : int = 10
# sumList [1;-2;3;5];;
- : int = 7
# sumList [1;3;5;7;9;11];;
- : int = 36
```

(b) 10 points

Write an OCaml function `digitsOfInt : int -> int list` that takes an integer `n` as an argument and if the integer is positive (i.e. I don't care what you return for the argument `0` or any negative number), returns the list of digits of `n` in the order in which they appear in `n` . Once you have implemented the function, you should get the following behavior at the OCaml prompt:

```
# digitsOfInt 3124;;
- : int list = [3;1;2;4]
# digitsOfInt 352663;;
- : int list = [3;5;2;6;6;3]
```

(c) 10+10 points

Consider the process of taking a number, adding its digits, then adding the digits of the number derived from it, etc., until the remaining number has only one digit. The number of additions required to obtain a single digit from a number n is called the *additive persistence* of n, and the digit obtained is called the

digital root of n. For example, the sequence obtained from the starting number 9876 is (9876, 30, 3), so 9876 has an additive persistence of 2 and a digital root of 3. Write two OCaml functions `additivePersistence : int -> int` and `digitalRoot : int -> int` that take positive integer arguments `n` and return respectively the additive persistence and the digital root of `n`. Once you have implemented the functions, you should get the following behavior at the OCaml prompt:

```
# additivePersistence 9876;;
- : int = 2
# digitalRoot 9876;;
- : int = 3
```

Problem #2: Palindromes (`misc.ml`)

(a) 15 points

Without using any built-in OCaml functions, write an OCaml function `listReverse : 'a list -> 'a list` that takes a list `l` as an argument and returns a list of the elements of `l` in the reversed order. Once you have implemented the function, you should get the following behavior at the OCaml prompt:

```
# listReverse [1;2;3;4];;
- : int list = [4;3;2;1]
# listReverse ["a";"b";"c";"d"];;
- : string list = ["d";"c";"b";"a"]
```

(b) 5 points

A *palindrome* is a word that reads the same from left-to-right and right-to-left. Write an OCaml function `palindrome : string -> bool` that takes a string `w` and returns true if the string is a palindrome and false otherwise. Your function should be case sensitie. You may want to use the OCaml function `explode`. (*Hint: You may call your `listReverse` function from your `palindrome` function.*) Once you have implemented the function, you should get the following behavior at the OCaml prompt:

```
# palindrome "malayalam";;
- : bool = true
# palindrome "Malayalam";;
- : bool = false
# palindrome "myxomatosis";;
- : bool = false
```
