CSE 130 - Programming Assignment #3

OCaml

100 points

(see submission instructions below)

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

(Programming Assignment #3 FAQ)

Note: See this for instructions on starting OCaml in the ACS lab machines. To download and install OCaml version 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. **Note:** While you can use windows to begin working with OCaml, the code you turn in must be that required for the ACS 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/NJ 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 to expose you to fold, fold, and more fold. And just when you think you've had enough, FOLD. The assignment is spread over two files misc.ml, test.ml, that you need to download. The first file contains several skeleton OCaml functions, with missing bodies, i.e. expressions, which currently contain the text `raise Failure "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 or while loops will receive **no credit**. Feel free to use any library functions that you want.

It is a good idea to start this assignment early as it is somewhat harder than the first assignment.

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 | grep "130>>" > 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 `raise ...`, 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";;
.
.
.
- : 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_pa3.zip` by going into the directory `solution` and executing the UNIX shell command:

```
zip <LastName>_<FirstName>_cse130_pa3.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_pa3` 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_pa3 <LastName>_<FirstName>_cse130_pa3.zip
```

The `validate_pa3` 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_pa3` 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` program

Once your zip file passes the validation check by `validate_pa3`, you will use the `turnin_pa3` program to submit this file for grading by going into the directory `solution/` and executing the UNIX shell command:

```
turnin_pa3 <LastName>_<FirstName>_cse130_pa3.zip
```

The `turnin_pa3` program will provide you with a confirmation of the submission process; make sure that the size of the file indicated by `turnin_pa3` matches the size of your tar file. (`turnin_pa3` is a thin wrapper script around the ACMS command `turnin` that repeats validation and ensures that the propper assignment name is passed). 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 #1: Warm-Up (`misc.ml`)

(a) 10 points

Fill in the skeleton given for `sqsum` , which uses `List.fold_left` to get an OCaml function `sqsum : int list -> int` that takes a list of integers `[x1;...;xn])` and returns the integer: `x1^2 + ... + xn^2` . Your task is to fill in the appropriate values for (1) the folding function `f` and (2) the base case `base`. Once you have implemented the function, you should get the following behavior at the OCaml prompt:

```
# sqsum [];;
- : int = 0
# sqsum [1;2;3;4] ;;
- : int = 30
# sqsum [-1;-2;-3;-4] ;;
- : int = 30
```

(b) 20 points

Fill in the skeleton given for `pipe`, which uses `List.fold_left` to get an OCaml function `pipe : ('a -> 'a) list -> ('a -> 'a)` . The function `pipe` takes a list of functions `[f1;...;fn])` and returns a function `f` such that for any `x`, the application `f x` returns the result `fn(...(f2(f1 x)))`. Again, your task is to fill in the appropriate values for (1) the folding function `f` and (2) the base case `base`. Once you have implemented the function, you should get the following behavior at the OCaml prompt:

```
# pipe [] 3;;
- : int = 3
# pipe [(fun x-> 2*x);(fun x -> x + 3)] 3 ;;
- : int = 9
# pipe [(fun x -> x + 3);(fun x-> 2*x)] 3;;
- : int = 12
```

(c) 15 points

Fill in the skeleton given for `sepConcat`, which uses `List.fold_left` to get an OCaml function `sepConcat : string -> string list -> -> string`. The function `sepConcat` is a curried function which takes as input a string `sep` to be used as a separator, and a list of strings `[s1;...;sn]`. If there are 0 strings in the list, then `sepConcat` should return `""`. If there is 1 string in the list, then `sepConcat` should return `s1`. Otherwise, `sepConcat` should return the concatination `s1 sep s2 sep s3 ... sep sn`. You should only modify the parts of the skeleton consisting of `failwith "to be implemented"`. You will need to define the function `f`, and give values for `base` and `l`. Once you have filled in these parts, you should get the following behavior at the OCaml prompt:

```
# sepConcat ", " ["foo";"bar";"baz"];;
- : string = "foo, bar, baz"
# sepConcat "---" [];;
- : string = ""
# sepConcat "" ["a";"b";"c";"d";"e"];;
- : string = "abcde"
# sepConcat "X" ["hello"];;
- : string = "hello"
```

(d) 5 points

Implement the curried OCaml function `stringOfList : ('a -> string) -> 'a list -> string`. The first input is a function `f: 'a -> string` which will be called by `stringOfList` to convert each element of the list to a string. The second input is a list `l: 'a list`, which we will think of as having the elemtns `l1, l2, ..., ln`. Your `stringOfList` function should return a string representation of the list `l` as a concatenation of the following: "[" (f l1) "; " (f l2) "; " (f l3) "; " ... "; " (f ln) "]". This function can be implemented on one line without using any recursion by calling `List.map` and `sepConcat` with appropriate inputes. Once you have completed this function, you should get the following behavior at the OCaml prompt:

```
# stringOfList string_of_int [1;2;3;4;5;6];;
- : string = "[1; 2; 3; 4; 5; 6]"
# stringOfList (fun x -> x) ["foo"];;
- : string = "[foo]"
# stringOfList (stringOfList string_of_int) [[1;2;3];[4;5];[6];[]];;
- : string = "[[1; 2; 3]; [4; 5]; [6]; []]"
```

Problem #2: Big Numbers (`misc.ml`)

As you may have noticed, the OCaml type `int` only contains values upto a certain size. For example, entering `9999999999` results in the message `int constant too large`. You will now implement functions to manipulate large numbers represented as lists of integers.

(a) 5 + 5 + 5 points

Write a curried function `clone : 'a -> int -> 'a list` which first takes as input `x` and then takes as input an integer `n`. The result is a list of length `n`, where each element is `x`. If `n` is 0 or negative, `clone` should return the empty list. Once you have implemented the function, you should get the following behavior at the OCaml prompt:

```
# clone 3 5;;
- : int list = [3; 3; 3; 3; 3]
# clone "foo" 2;;
- : string list = ["foo"; "foo"]
# clone clone (-3);;
- : ('_a -> int -> '_a list) list = []
```

Use the function `clone` to write a curried function `padZero : int list -> int list -> int list * int list` which takes two lists: `[x1,...,xn] [y1,...,ym]` and adds zeros in front to make the lists equal. Once you have implemented the function, you should get the following behavior at the OCaml prompt:

```
# padZero [9;9] [1;0;0;2];;
- : int list * int list = ([0;0;9;9],[1;0;0;2])
# padZero [1;0;0;2] [9;9];;
- : int list * int list = ([1;0;0;2],[0;0;9;9])
```

Now write a function `removeZero : int list -> int list`, that takes a list and removes a prefix of trailing zeros. Once you have implemented the function, you should get the following behavior at the OCaml prompt:

```
# removeZero [0;0;0;1;0;0;2];;
- : int list = [1;0;0;2]
# removeZero [9;9];;
- : int list = [9;9]
# removeZero [0;0;0;0];;
- : int list = []
```

(b) 15 points

Suppose we use the list `[d1;d2;d3;...;dn]`, where each `di` is in the range [0..9], to represent the (positive) integer `d1d2d3...dn`. For example, the list `[9;9;9;9;9;9;9;9;9;9]` represents the integer 9999999999. Fill out the implementation for `bigAdd : int list -> int list -> int list`, so that it takes two integer lists, where each integer is in the range [0..9] and returns the list corresponding to the addition of the two big integers. Again, you have to fill in the implementation to supply the appropriate values to `f, base, args`. Once you have implemented the function, you should get the following behavior at the OCaml prompt:

```
# bigAdd [9;9] [1;0;0;2];;
- : int list = [1;1;0;1]
# bigAdd [9;9;9;9] [9;9;9];;
```

```
- : int list = [1;0;9;9;8]
```

(c) 10 + 10 points

Next you will write functions to multiply two big integers. First write a function `mulByDigit : int -> int list -> int list` which takes an integer digit and a big integer, and returns the big integer list which is the result of multiplying the big integer with the digit. Once you have implemented the function, you should get the following behavior at the OCaml prompt:

```
# mulByDigit 9 [9;9;9;9];;
- : int list = [8;9;9;9;1]
```

Now, using the function `mulByDigit`, fill in the implementation of `bigMul : int list -> int list -> int list` . Again, you have to fill in implementations for `f` , `base` , `args` only. Once you are done, you should get the following behavior at the prompt:

```
# bigMul [9;9;9;9] [9;9;9;9];;
- : int list = [9;9;9;8;0;0;0;1]
# bigMul [9;9;9;9;9] [9;9;9;9;9];;
- : int list = [9;9;9;9;8;0;0;0;0;1]
```