ECE/CPSC 3520
SS II 2020
Software Design Exercise #1
Solving Problems Inspired by Google, Amazon, Microsoft, etc. Coding Interviews Using `ocaml`

Canvas Submission Only
Assigned 7/2/2020
Due 7/14/2020 11:59PM

Contents

1 Preface

The overall objective of SDE1 is to implement 4 sets of functions corresponding to 4 different list-based problems. Three of these problems were inspired by actual coding tests given during interviews for positions with companies such as Google, Amazon, Microsoft and Apple. This is to be done using a purely functional programming paradigm and `ocaml`. SDE2 will re-explore these problems using the declarative paradigm in Prolog. Note that a Canvas video lecture accompanies this assignment.
Notes:

1. Some of the problems which inspired this assignment refer to arrays or strings. **The common (and required) data structure in this assignment is the `ocaml` list.**

2. **The videos provided are only to characterize the problem.** The video solutions presented are imperative, i.e., neither functional (`ocaml` in SDE1) or declarative (Prolog in SDE2). Typically pointers and/or imperative (java, c) solutions are shown. Thus, the video solution probably has little to do with your functional or declarative (SDE2) solution.

The problems are:

first duplicate in a list problem :

`https://www.youtube.com/watch?v=XSdr_0-XVRQ`

first non-repeating in a list problem :

`https://youtu.be/5co5Gvp_-S0`

the sum of 2 problem :

`https://www.youtube.com/watch?v=BoHO04xVeU0`

`https://www.youtube.com/watch?v=sfuZzBLPcx4`

CYK parsing algorithm-inspired list decomposition problem :

Given n, return a list of tuples, each indicating how a string of length n could be formed from 2 strings. [(1,n-1); (2,n-2);...;(n-1,1)] is the returned `ocaml` list.

This should actually be some fun and really good experience with coding interview tests. Each of the problems is described separately, with examples.

2 The Problems and Required ocaml Functions

2.1 First Duplicate in a List (first_duplicate)

Pay special attention to the naming and argument(s) of each required ocaml function.
You can see the problem formulation at:
https://www.youtube.com/watch?v=XSdr_0-XVRQ

```
Prototype: first_duplicate alist
   returns -10000 if there are no duplicates in
            integer list alist
            otherwise first duplicate in the integer list

Signature: val firstduplicate : int list -> int = <fun>

Sample Use:

# first_duplicate [1;2;3;4;5;6;7;4;5;8;9];;
- : int = 4

# first_duplicate [1;2;3;4;5;6;7;4;5;2;9];;
- : int = 2

# first_duplicate [1;2;3;4;5;6;7;8;9;10];;
- : int = -10000
```

2.2 First Non-Repeating Element in a List (first_nonrepeating)

You can see the problem formulation at:
https://youtu.be/5co5Gvp_-S0

```
Prototype: first_nonrepeating alist
   "Find and return the first non-repeated element in the *entire* list"
   returns -10000 if no non-repeated element in integer alist
   otherwise first non-repeating element in alist

Signature: val first_nonrepeating : int list -> int = <fun>

Sample Use:

# first_nonrepeating [1;2;3;2;7;5;6;1;3];;
```

```
- : int = 7
# first_nonrepeating [1;2;9;3;2;7;5;6;1;3];;
- : int = 9
# first_nonrepeating [1;2;9;3;2;7;5;6;10;30];;
- : int = 1
# first_nonrepeating [1;2;9;3;2;7;5;6;1;10;30];;
- : int = 9
# first_nonrepeating [1;2;9;3;2;7;5;9;6;1;10;30];;
- : int = 3
# first_nonrepeating [1;2;3;2;7;5;6;1;3];;
- : int = 7
# first_nonrepeating [1;2;3;4;5;1;2;3;4;5];;
- : int = -10000
# first_nonrepeating [1;2;3;4;5;1;2;3;4;9];;
- : int = 5
# first_nonrepeating [1;1;1;2;2;2];;
- : int = -10000
```

2.3 The Sum of 2 Problem (`sumOfTwo`)

You can see the problem formulation at:

`https://www.youtube.com/watch?v=BoHO04xVeU0`

`https://www.youtube.com/watch?v=sfuZzBLPcx4`

Two arrays contain numbers able to produce a given sum. A sample case from the video:

`If a=[1;2;3], b=[10;20,30,40] and v=42, then sumOfTwo(a,b,v)=true since 40+2)=42=v.`

```
Prototype: sumOfTwo(a,b,v)

Signature: val sumOfTwo : int list * int list * int -> bool = <fun>

Samples:

# sumOfTwo([1;2;3],[10;20;30;40],42);;
- : bool = true
# sumOfTwo([1;2;3],[10;20;30;40],40);;
- : bool = false
# sumOfTwo([1;2;3],[10;20;30;40],41);;
- : bool = true
# sumOfTwo([1;2;3],[10;20;30;40],43);;
- : bool = true
```

```
# sumOfTwo([1;2;3],[10;20;30;40],44);;
- : bool = false
# sumOfTwo([1;2;3],[10;20;30;40],11);;
- : bool = true
# sumOfTwo([1;2;3],[10;20;30;40],15);;
- : bool = false
```

2.4 CYK Parsing Algorithm-Inspired Problem (`cyk_sublists`)

Please note we are NOT trying to implement the CYK algorithm in `ocaml`. However, in forming the CYK parse table (Book, Chapter 4, Section 4.4.6) it is necessary to consider how many ways a string (here a list) of length `n` may be comprised by concatenation of two non-empty sublists.
Given `n`, `cyk_sublists` returns a list of tuples. Each tuple is the length of the 2 component strings (and the two tuple elements sum to `n`).

```
Prototpe: cyk_sublists n

Signature: val cyk_sublists : int -> (int * int) list = <fun>

Samples:

# cyk_sublists 4;;
- : (int * int) list = [(1, 3); (2, 2); (3, 1)]
# cyk_sublists 3;;
- : (int * int) list = [(1, 2); (2, 1)]
# cyk_sublists 5;;
- : (int * int) list = [(1, 4); (2, 3); (3, 2); (4, 1)]

alternately (* note parens are superfluous *):

# cyk_sublists(6);;
- : (int * int) list = [(1, 5); (2, 4); (3, 3); (4, 2); (5, 1)]
```

3 Resources

As mentioned previously, it would be foolish to attempt this SDE without first carefully studying:

1. The text, especially the many examples in Chapter 11;

2. The ocaml course lectures (slides/videos);
3. The `ocaml` manual; and
4. The related SDE1 video posted on Canvas.

4 `ocaml` Functions and Constructs Not Allowed

> Of extreme significance is the restriction of the paradigm to pure functional programming (no side effects). **No `ocaml` imperative constructs are allowed.** Recursion must dominate the function design process.

So that we may gain experience with functional programming, *only the applicative (functional) features of* `ocaml` *are to be used.* **Please reread the previous sentence.** This rules out the use of `ocaml`'s imperative features. See Section 1.5 'Imperative Features' of the manual for examples of constructs not to be used. **To force you into a purely applicative style, `let` should be used only for function definition**. `let` cannot be used in a function body. Loops and local or global variables are prohibited.

The allowable functions in SDE1 are **only** those non-imperative functions in the Pervasives module and these 4 individual functions listed below:

```
List.hd
List.tl
List.nth
List.mem
```

This means you may not use the Array Module.

Finally, **the use of sequence (6.7.2 in the ocaml manual) is not allowed**. Do not design your functions using sequential expressions or begin/end constructs. Here is an example of a sequence in a function body:

```
let print_assignment = function(student,course,section) ->
print_string student; (* first you evaluate this*)
print_string " is assigned to "; (* then this *)
print_string course;  (* then this *)
print_string " section " ; (* then this *)
print_int section;  (* then this *)
print_string "\n;; (* then this and return unit*)
```

If you are in doubt, ask and I'll provide a 'private-letter ruling'.

The objective is to obtain proficiency in functional programming, not to try to find built-in `ocaml` functions or features which simplify or trivialize the effort.

5 How We Will Build and Evaluate Your ocaml Solution

An `ocaml` script with varying input files and test vectors are used to test the 4 required functions in Section 2. The grade is based upon a correctly working solution.

6 Format of the Electronic Submission

The final **zipped** archive is to be named **<yourname>-sde1_SSII2020.zip**, where `<yourname>` is your (CU) assigned user name. You will upload this archive to the Canvas assignment Content item prior to the deadline.

The minimal contents of this archive are as follows:

1. A `readme.txt` file listing the contents of the archive and a brief description of each file. Include 'the pledge' here. Here's the pledge:

   > **Pledge:**
   > On my honor I have neither given nor received aid on this exam.

   This means, among other things, that the code you submit is **your** code.

2. The `ocaml` source for your implementation in a single file named `sde1.caml`. Note this file must include all the required functions, as well as any additional functions you design and implement. We will supply the testing data.

3. An ASCII log file showing 2 sample uses of each of the functions required. Name this log file `sde1.log`.

The use of `ocaml` should not generate any errors or warnings. The grader will attempt to interpret your ocaml source and check for correct functionality. We will also look for offending `let` use and sequences. Recall the grade is based upon a correctly working solution.

7 Final Remark: Deadlines Matter

This remark is included in the course syllabus. Since multiple submissions to Canvas are allowed[1], if you have not completed all functions, you should submit a freestanding archive of your current success before the deadline. This will allow the possibility of partial credit. **Do not attach any late submissions to email and send them to either me or the graders.**

[1]But we will only download and grade the latest (or most recent) one, and it must be submitted by the deadline.