ECE/CPSC 3520
Fall 2016
Software Design Exercise #3

Blackboard submission only

Assigned 11-2-2016; Due 12-2-2016 10:00 PM

Contents

1 Preface

1.1 Objectives

The objective of SDE 3 is to implement an `ocaml` version of a fuzzy system for inference. This involves computations on set membership functions. Of extreme significance is the restriction of the paradigm to pure functional programming, i.e., **no imperative constructs are allowed.** This document specifies a number of functions which must be developed as part of the effort.

This effort is straightforward, and 4 weeks are allocated for completion. The motivation is to:

- Learn the paradigm of (pure) functional programming;
- Implement a (purely) functional version of an interesting technology;
- Deliver working functional programming-based software based upon specifications; and
- Learn `ocaml`.

1.2 Resources

As discussed in class, it would be foolish to attempt this SDE without carefully exploring:

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

2. The `ocaml` lectures;
3. The background provided in this document;
4. **Class discussions and examples, especially the 11/4/2016 class devoted to this SDE**;
5. `https://en.wikipedia.org/wiki/Membership_function_(mathematics)`; and
6. The `ocaml` reference manual.

1.3 Standard Remark

Please note:

1. **This assignment assesses *your* effort (not mine)**. I will not debug or design your code, nor will I install software for you. You (and only you) need to do these things.
2. It is never too early to get started on this effort.
3. We will continue to discuss this in class.

2 Preliminary Considerations – Set Representations

Fuzzy (and crisp) sets are useful data structures for a number of computational tasks. Fuzzy sets have gained some notoriety for the manipulation of imprecise information. I'll have more to say about this in class on 11/4/2016.

2.1 Representing Set Membership Functions, $\mu_i(x)$

We first consider the necessary, basic data structures used to represent a set, via a membership function, $\mu_i(x)$, in `ocaml`. One complication is the restriction of lists to be of the same type, therefore we employ a combination of tuples and lists. We show this structure by the following 'hello-world'-scale example[1].

[1]Typical domains are 100-1000 points. Later, I'll post a few more examples.

```
(* ------------sample data structures------------ *)

let s_list1 = [(1,0.0);(2,0.3);(3,0.9);(4,1.0);(5,0.8);(6,0.5);(7,0.0)];;

let s_list2 = [(1,0.0);(2,0.5);(3,0.7);(4,0.8);(5,0.8);(6,0.25);(7,0.3)];;

let mu1 = ((1,7), s_list1);;

let mu2 = ((1,7), s_list2);;
```

2.2 Corresponding `ocaml` Representation Signatures

```
val s_list1 : (int * float) list =
  [(1, 0.); (2, 0.3); (3, 0.9); (4, 1.); (5, 0.8); (6, 0.5); (7, 0.)]
val s_list2 : (int * float) list =
  [(1, 0.); (2, 0.5); (3, 0.7); (4, 0.8); (5, 0.8); (6, 0.25); (7, 0.3)]
val mu1 : (int * int) * (int * float) list =
  ((1, 7),
   [(1, 0.); (2, 0.3); (3, 0.9); (4, 1.); (5, 0.8); (6, 0.5); (7, 0.)])
val mu2 : (int * int) * (int * float) list =
  ((1, 7),
   [(1, 0.); (2, 0.5); (3, 0.7); (4, 0.8); (5, 0.8); (6, 0.25); (7, 0.3)])
```

Notes:

1. Carefully note the signatures from the above representations.

2. Assume the user is intelligent enough to restrict fuzzy set membership values, $\mu_i(x)$, to the interval `[0,1]`. Notice that set operations such as union and intersection require:

   - The domains for the fuzzy sets to be the same. Here it is `(1,7)`.
   - Corresponding elements in the singleton-based list representations for all integers in the domain of each set.

2.3 Specific Set Manipulations and Functionality to Implement

1. Set intersection, union and complement

2. Linguistic hedges

3. Selected error checking on inputs
4. Composite operations

3 Prototypes, Signatures and Examples of Functions to be Developed

Notes:

- The use of `let` in the following examples is only for illustration and naming of the input μ_i. `let` should not appear in your `ocaml` source for naming functions. See Section 6.
- **Carefully observe the function naming convention. Case matters. We will not rename any of the functions you submit. Reread the preceding three sentences at least 3 times.**
- You may (actually, 'must') develop additional functions to assist in the implementation of the required functions.
- **Note the argument interface on all multiple-argument functions you will design, implement and test is curried**.
- You should work through all the samples by hand to get a better idea of the computation prior to function design, implementation and testing.
- Note some of my signatures indicate polymorphic behavior on the fuzzy set domain values. This is OK, although we use `int` on our domain values.

3.1 `set_intersect`

Set intersection for sets 1 and 2 is defined as the point-by-point min of $\mu_1(x)$ and $\mu_2(x)$.

```
(**
Prototype: set_intersect mu1 mu2

Input(s): mu1 mu2
Returned Value: intersection of two input sets or exception
Side Effects: none
Signature:
val set_intersect :
  ('a * 'b) * ('c * 'd) list ->
  ('a * 'b) * ('c * 'd) list -> ('a * 'b) * ('c * 'd) list = <fun>
Notes:
*)
```

Sample Use.

```
# set_intersect mu1 mu2;;
- : (int * int) * (int * float) list =
((1, 7),
 [(1, 0.); (2, 0.3); (3, 0.7); (4, 0.8); (5, 0.8); (6, 0.25); (7, 0.)])
#
```

3.2 set_union

Set union is defined as the point-by-point max of $\mu_1(x)$ and $\mu_2(x)$.

```
(**
Prototype:  set_union mu1 mu2

Input(s): mu1 mu2
Returned Value: union of 2 input sets or exception
Side Effects: none
Signature:
val set_union :
  ('a * 'b) * ('c * 'd) list ->
  ('a * 'b) * ('c * 'd) list -> ('a * 'b) * ('c * 'd) list = <fun>

Notes:
*)
```

Sample Use.

```
# set_union mu1 mu2;;
- : (int * int) * (int * float) list =
((1, 7),
 [(1, 0.); (2, 0.5); (3, 0.9); (4, 1.); (5, 0.8); (6, 0.5); (7, 0.3)])
#
```

3.3 set_compl

This is set complement.

$$\mu_i^{complement}(x) = \mu_i(1 - x)$$

```
(**
Prototype: set_compl mu

Input(s): mu
Returned Value: set complement
Side Effects: none
Signature:
val set_compl : 'a * ('b * float) list -> 'a * ('b * float) list = <fun>

Notes:
*)
```

Sample Use.

```
# set_compl mu1;;
- : (int * int) * (int * float) list =
((1, 7),
 [(1, 1.); (2, 0.7); (3, 0.0999999999999999778); (4, 0.);
  (5, 0.199999999999999956); (6, 0.5); (7, 1.)])
#
```

3.4 set_somewhat

Here we introduce a so-called **linguistic hedge** on a fuzzy set. Here we show 'somewhat'.

$$\mu_i^{somewhat}(x) = \mu_i(x^{0.333})$$

```
(**
Prototype: set_somewhat mu

Input(s): mu
Returned Value: fuzzy set 'somewhat' of original
Side Effects: none
Signature:
val set_somewhat : 'a * ('b * float) list -> 'a * ('b * float) list = <fun>

Notes:
*)
```

Sample Use.

```
# set_somewhat mu1;;
- : (int * int) * (int * float) list =
((1, 7),
 [(1, 0.); (2, 0.669701663687948); (3, 0.965523293354208167); (4, 1.);
  (5, 0.928386818665011382); (6, 0.793883930931652437); (7, 0.)])
#
```

4 Extension to Input Error Detection and Raising `ocaml` Exceptions

I strongly recommend you attempt this part (it involves enhancements to your required functions in Section 3) ONLY after completing the design, implementation and testing of the functions in Section 3. The objective is to give you some experience with raising exceptions. Use the `failwith` construct to implement these exceptions.

4.1 Malformed μ_i

We are not attempting to detect all possible input errors. The examples shown below are sufficient. In the future. another class may have the opportunity to consider the general case. We are only interested in 3 problems cases wherein 2 input membership functions are input. They are:

1. Mismatched domains specified;
2. Unequal lists; and
3. Erroneous lists of singletons.

Each of these is shown in the examples below.

4.2 Bad Input Examples

Take a look at each of the ill-formed input examples below and see if you can determine the problem.

```
let s_list1b = [(1,0.0);(2,0.3);(3,0.9);(4,1.0);(5,0.8);(6,0.5);(7,0.0)];;

let s_list2b = [(1,0.0);(2,0.5);(3,0.7);(4,0.8);(5,0.8);(6,0.25);(7,0.3)];;

let mu1b = ((1,6), s_list1b);;  (* bad domain *)

let mu2b = ((1,7), s_list2b);;

let s_list1c = [(1,0.0);(2,0.3);(4,1.0);(5,0.8);(6,0.5);(7,0.0)];;

let s_list2c = [(1,0.0);(2,0.5);(3,0.7);(4,0.8);(5,0.8);(6,0.25);(7,0.3)];;

let mu1c = ((1,7), s_list1c);;  (* bad slist1 *)

let mu2c = ((1,7), s_list2c);;

let s_list1d = [(1,0.0);(2,0.3);(3,0.9);(4,1.0);(5,0.8);(6,0.5);(7,0.0)];;

let s_list2d = [(1,0.0);(2,0.5);(2,0.7);(4,0.8);(5,0.8);(6,0.25);(7,0.3)];;

let mu1d = ((1,7), s_list1d);;

let mu2d = ((1,7), s_list2d);;  (* bad slist2 -- length OK, but repeated/missing *)
```

4.3 Sample Response to Erroneous Input (Raising Exceptions)

```
# set_intersect mu1b mu2b;;
Exception: Failure "problem with domain or list length".

# set_intersect mu1c mu2c;;
Exception: Failure "problem with domain or list length".

# set_intersect mu1d mu2d;;
Exception: Failure "problem with singleton list element(s)".
```

The behavior with `set_union` should be analogous.

5 How We Will Grade Your Solution

The script below will be used with varying input files and parameters.

```
#use "sde3.caml";;               (* YOUR ocaml source -- all the required functions
                                   and any additional (supporting) functions you develop*)
#use "inputs.caml";;             (* OUR TEST inputs/cases/mu_i *)
<see above examples>             (* sample invocation of the functions *)
```

The grade is based upon a correctly working solution.

6 `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. To this end, we impose the following constraints on the solution.

6.1 No `let` for Local or Global Variables

So that you 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` **can only be used for function naming**. `let` or the keyword `in` cannot be used in a function body. Reread the following sentence. Loops and 'local' or 'global' variables or nested function definitions is prohibited.

6.2 Only Pervasives Module and 3 Functions from the List Module

The only module you may use (other than Pervasives) is the List module, and only the functions `List.hd`, `List.tl` and `List.length` from this module. This means no list iterators. Anything else inhibits further grading of your submission.

6.3 No Sequences

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*)
```

6.4 No (Nested) Functions

`ocaml` allows functions defined within functions definitions (another 'illegal' `let` use). SDE3 does not allow this. Here's an example of a nested function definition:

```
# let f a b =
    let x = a +. b in
    x +. x ** 2.;;
```

6.5 Appeals

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. I want you to come away from SDE3 with a perspective on (almost) pure functional programming (no side effects).

7 Format of the Electronic Submission

The final **zipped** archive is to be named **`<yourname>`-sde3.zip**, where `<yourname>` is your (CU) assigned user name. You will upload this to the Blackboard assignment 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 single `ocaml` source file for your function implementations. The file is to be named `sde3.caml`. Note this file must include all the functions defined in this document. It may also contain other 'helper' or auxiliary functions you developed.

3. A log of 2 sample uses of each of the required functions. Name this log file `sde3.log`.

The use of `ocaml` should not generate any errors or warnings. Recall the grade is based upon a correctly working solution with the restrictions posed herein.