ECE/CPSC 3520
Spring 2019
Software Design Exercise #2
Grammar-Based Image Analysis in `ocaml`

Canvas submission only (rev. 2-22-2019)

Assigned 3/7/2019; Due 4/11/2019 11:59 PM

Contents

1 Preface

1.1 Objectives

The objective of SDE 2 is to implement the computations of selected SDE1 predicates as functions in `ocaml`. Recall in SDE 1 you worked with the picture description language (PDF), a grammar-base technique to extract object or region outlines from an image using geometric primitives derived from the image. In this assignment, we will specify 14 functions which **must** be developed as part of this assignment. I recommend you attempt development in the order in which the `ocaml` functions are introduced.

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

- Learn the paradigm of functional programming;
- Implement a functional (`ocaml`) version of an interesting algorithm;
- Deliver working 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 background provided in this document;
3. **Class discussions and examples**; and
4. The `ocaml` website and reference manual.

1.3 Standard Remark

Please note:

1. **This assignment (which counts as a quiz) tests *your* effort (not mine or those of your classmates or friends)**. 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. This is an individual effort.
3. It is never too early to get started on this effort. Perhaps you learned this from SDE 1.
4. We will continue to discuss this in class.

2 Preliminary Considerations

2.1 Data Structures and Representation in `ocaml`

We will not repeat the PDL background included in SDE1. Refer to the SDE 1 document for a review.

We will encode PDL contour descriptions using string lists in `ocaml`. For example, `"abcd"` is a string and `["a";"b";"c";"d"]` is an `ocaml` string list, as shown below:

```
# "abcd";;
- : string = "abcd"
# ["a";"b";"c";"d"];;
- : string list = ["a"; "b"; "c"; "d"]
```

3 Prototypes, Signatures and Examples of Functions to be Designed, Implemented and Tested

You will design, implement, test and deliver the functions described in the following sections. Also note:

1. **Pay special attention to the function naming and argument specifications**. Case matters. Since your submission will be graded by an `ocaml` script, if you deviate from this your tests will probably fail. **The graders will not fix or change anything in your submission.**

2. **All required multi-argument functions you will develop, implement and test have a tupled interface.**
3. Pay attention to the file naming conventions specified herein.
4. Pay special attention to the function signatures indicated, and check your submission against these signatures.
5. Your implementation of **all** `ocaml` functions is to be included in a single file named `which_shape.caml` in your archive.
6. You may design and use other functions to support any of these functions, but they are not (directly) graded. Of course, they must also be included in your single `ocaml` source file in your archive and are subject to the constraints of Section 6.

3.1 gen_string

This function, while required, is relatively easy and serves as one of several 'warmup' functions.

```
/**
Prototypes: gen_string(n,what)

Arguments: n is the ocaml string length
           what is the string contents (repeated n times)

Returned value: A string of n whats

Side Effects: none

Signature: val gen_string : int * 'a -> 'a list = <fun>
```

Sample Use.

```
# gen_string(6,"a");;
- : string list = ["a"; "a"; "a"; "a"; "a"; "a"]
# gen_string(5,"u");;
- : string list = ["u"; "u"; "u"; "u"; "u"]
# gen_string(5,3520);;
- : int list = [3520; 3520; 3520; 3520; 3520]
```

3.2 gen_square

```
/**
Prototype: gen_square(n)

Inputs: n is side length of the square

Returned Value: a string list representing a square of side length n
                starting with the lowest "u".

Side Effects: none

Signature: val gen_square : int -> string list = <fun>

Notes: Suggest use of gen_string
*/
```

Sample Use.

```
# gen_square(3);;
- : string list =
["u"; "u"; "u"; "r"; "r"; "r"; "d"; "d"; "d"; "l"; "l"; "l"]
# gen_square(5);;
- : string list =
["u"; "u"; "u"; "u"; "u"; "r"; "r"; "r"; "r"; "r"; "d"; "d"; "d"; "d"; "d";
 "l"; "l"; "l"; "l"; "l"]
# gen_square(1);;
- : string list = ["u"; "r"; "d"; "l"]
```

3.3 gen_rect

```
/**
Prototype: gen_rect(n,m)

Inputs: n is length of "u" and "d" sides; m is length of "r" and "l" sides

Returned Value: String list representing rectangle

Side Effects: none

Signature: val gen_rect : int * int -> string list = <fun>

Notes: Not much harder than square
*/
```

Sample Use.

```
# gen_rect(2,10);;
- : string list =
["u"; "u"; "r"; "r"; "r"; "r"; "r"; "r"; "r"; "r"; "r"; "r"; "d"; "d"; "l";
 "l"; "l"; "l"; "l"; "l"; "l"; "l"; "l"; "l"]
# gen_rect(1,2);;
- : string list = ["u"; "r"; "r"; "d"; "l"; "l"]
```

3.4 countups

```
/**
Prototype: countups (alist,0)

Inputs: a string list. The second argument is 0.
        (This is a hint for subsequent function development).

Returned Value: The number of "u" (ups) in the list.

Side Effects: none

Signature: val countups : string list * int -> int = <fun>

Notes: prelude to development of parsing functions.
*/
```

Sample Use.

```
# countups (["a"],0);;
- : int = 0

# countups (["u"],0);;
- : int = 1

# countups (["u";"a"],0);;
- : int = 1

# countups (["u";"a";"u"],0);;
- : int = 2
```

3.5 consec_counts

```
/**
```

```
Prototype: consec_counts(alist,count,what)

Inputs: a list of arbitrary type, 0, thing to be counted

Returned Value: returns tuple (rest,n) where n is number of consecutive
what in string and rest is the rest of the string

Side Effects: none

Signature: val consec_counts : 'a list * int * 'a -> 'a list * int = <fun>

Notes: counts CONSECUTIVE string of "x" in arbitrary string list
       starting at beginning of list. If this sounds LGN-ish, you're right.
       Look at the examples.
*/
```

Sample Use.

```
# consec_counts(["a";"a";"a"],0,"a");;
- : string list * int = ([], 3)
# consec_counts(["a";"a";"b"],0,"a");;
- : string list * int = (["b"], 2)
# consec_counts(["a";"b";"b"],0,"a");;
- : string list * int = (["b"; "b"], 1)
# consec_counts(["b";"b";"b"],0,"a");;
- : string list * int = (["b"; "b"; "b"], 0)
# consec_counts(["b";"b";"a"],0,"a");;
- : string list * int = (["b"; "b"; "a"], 0)
```

4 Prototypes, Signatures and Examples of Higher-Level functions to be Developed

Here, the real work starts. These functions have the higher-level functionality involved for recognition of the shape class. Using the same cyclic shift strategy of SDE1, we also develop this ability **regardless of where the PDL string description starts**. Recall in earlier parser development we assumed the list always started with the 'lowest' "u".

4.1 sq

```
/**
```

```
Prototype: sq(alist)

Inputs: list to be parsed.

Returned Value: Boolean. 1 if string is (u^n r^m d^j l^k); 0 otherwise. See examples.

Side Effects: none

Signature: val sq : 'a list -> bool = <fun>

Notes: This is the non-contextual parser, i.e., it does not really parse for a square,
       but rather, given dirs = it parses for (u^n r^m d^j l^k) and allows leftover
       elements. I suggest developing a pair of mutually recursive functions and maybe
       employing consec_counts.
*/
```

Sample Use.

```
# sq(["u";"r";"d"]);;
- : bool = false
# sq(["u";"r";"d";"l"]);;
- : bool = true
# sq(["u";"u";"r";"d";"d";"d";"l"]);;
- : bool = true
# sq(["u";"u";"d";"d";"d";"l"]);;
- : bool = false
```

4.2 sq_all

```
/**
Prototype: sq_all(alist)

Inputs: Same as sq

Returned Value: Same as sq, but entire string must be used (i.e., "consumed").

Side Effects: none

Signature: val sq_all : 'a list -> bool = <fun>

Notes: See remarks for sq. Requires [] left over in parse.
*/
```

Sample Use.

```
# sq(["u";"u";"r";"r";"d";"d";"l";"l";"d";"r"]);;
- : bool = true
# sq_all(["u";"u";"r";"r";"d";"d";"l";"l";"d";"r"]);;
- : bool = false
```

4.3 sqA

This is an evolution of your previous effort (sq_all), and probably the most complex. The 'A' suffix indicates attributes are used.

```
/**
Prototype: sqA(alist)

Inputs: List to be parsed as a square.

Returned Value: Boolean

Side Effects: none

Signature: val sqA : string list -> bool = <fun>

Notes: Must have equal length sides and nothing leftover.
       Must handle any side length (>0) square, so don't design by enumeration.
*/
```

Sample Use.

```
# sqA(["u";"u";"r";"r";"d";"d";"l";"l"]);;
- : bool = true
# sqA(["u";"u";"r";"r";"d";"d";"l";"l";"d"]);;
- : bool = false
# sqA(["u";"u";"r";"r";"d";"d";"l";"l";"l"]);;
- : bool = false
# sqA(["u";"u";"u";"r";"r";"r";"d";"d";"d";"l";"l";"l";"l"]);;
- : bool = false
# sqA(["u";"u";"u";"r";"r";"r";"d";"d";"d";"l";"l";"l"]);;
- : bool = true
# sqA(["u";"u";"u";"r";"r";"r";"d";"d";"x";"l";"l";"l"]);;
- : bool = false
```

4.4 eqtriA

```
/**
Prototype: eqtriA(alist)
```

```
Inputs: List to be parsed as equilateral triangle

Returned Value: Boolean

Side Effects: none

Signature: val eqtriA : string list -> bool = <fun>

Notes: Similar constraints to sqA, but for equilateral triangle.
*/
```

Sample Use.

```
# eqtriA(["u";"m30";"p240"]);;
- : bool = true
# eqtriA(["u";"m30";"p240";"p240"]);;
- : bool = false
# eqtriA(["u";"u";"m30";"m30";"p240";"p240"]);;
- : bool = true
# eqtriA(["u";"m30";"m30";"p240";"p240";"u"]);;
- : bool = false
```

The last example immediately above indicates we need to consider cyclic shifts for recognition with the description starting anywhere on the contour. This technique was employed in SDE 1, and introduces the need for the remaining functions.

4.5 one_shift

```
/**
Prototype: one_shift(alist)

Inputs: List to be shifted one element (to the left)

Returned Value: Shifted input list

Side Effects: none

Signature: val one_shift : 'a list -> 'a list = <fun>

Notes: one_shift assumes > 0 elements in list
*/
```

Sample Use.

```
# one_shift([1;2;3]);;
- : int list = [2; 3; 1]

# one_shift [1;2;3];;  (* one argument; 1-tuple *)
- : int list = [2; 3; 1]

# one_shift ["u";"r";"d";"l"];;
- : string list = ["r"; "d"; "l"; "u"]

# one_shift(one_shift(["u";"r";"d";"l"]));;
- : string list = ["d"; "l"; "u"; "r"]

# one_shift(one_shift(one_shift(["u";"r";"d";"l"])));; (* big hint *)
- : string list = ["l"; "u"; "r"; "d"]
```

4.6 all_shifts

```
/**
Prototype: all_shifts(alist)

Inputs: List to be shifted

Returned Value: All cyclic shifts of alist, not including (unshifted) alist.

Side Effects: none

Signature: val all_shifts : 'a list -> 'a list list = <fun>

Notes: none
*/
```

Sample Use.

```
# all_shifts(["u";"r";"d";"l"]);;
- : string list list =
[["r"; "d"; "l"; "u"]; ["d"; "l"; "u"; "r"]; ["l"; "u"; "r"; "d"]]

# all_shifts(["u";"m30";"p240"]);;
- : string list list = [["m30"; "p240"; "u"]; ["p240"; "u"; "m30"]]

# all_shifts(["u";"u";"r";"r";"d";"d";"l";"l"]);;
- : string list list =
[["u"; "r"; "r"; "d"; "d"; "l"; "l"; "u"];
```

```
 ["r"; "r"; "d"; "d"; "l"; "l"; "u"; "u"];
 ["r"; "d"; "d"; "l"; "l"; "u"; "u"; "r"];
 ["d"; "d"; "l"; "l"; "u"; "u"; "r"; "r"];
 ["d"; "l"; "l"; "u"; "u"; "r"; "r"; "d"];
 ["l"; "l"; "u"; "u"; "r"; "r"; "d"; "d"];
 ["l"; "u"; "u"; "r"; "r"; "d"; "d"; "l"]]
```

4.7 all_cases

```
/**
Prototype: all_cases(alist)

Inputs: List to be shifted

Returned Value: List of all shifts of alist, including no shifts.

Side Effects: none

Signature: val all_shifts : 'a list -> 'a list list = <fun>

Notes: Just all_shifts with alist on front of returned list.
       Brain dead if you have developed function all_shifts.
*/
```

Sample Use.

```
# all_cases(["u";"r";"d";"l"]);;
- : string list list =
[["u"; "r"; "d"; "l"]; ["r"; "d"; "l"; "u"]; ["d"; "l"; "u"; "r"];
 ["l"; "u"; "r"; "d"]]

# all_cases(["u";"m30";"p240"]);;
- : string list list =
[["u"; "m30"; "p240"]; ["m30"; "p240"; "u"]; ["p240"; "u"; "m30"]]

# all_cases(["u";"u";"r";"r";"d";"d";"l";"l"]);;
- : string list list =
[["u"; "u"; "r"; "r"; "d"; "d"; "l"; "l"];
 ["u"; "r"; "r"; "d"; "d"; "l"; "l"; "u"];
 ["r"; "r"; "d"; "d"; "l"; "l"; "u"; "u"];
 ["r"; "d"; "d"; "l"; "l"; "u"; "u"; "r"];
 ["d"; "d"; "l"; "l"; "u"; "u"; "r"; "r"];
 ["d"; "l"; "l"; "u"; "u"; "r"; "r"; "d"];
 ["l"; "l"; "u"; "u"; "r"; "r"; "d"; "d"];
 ["l"; "u"; "u"; "r"; "r"; "d"; "d"; "l"]]
```

4.8 try_all_sqA

Now that we can generate all cyclic shifts, we create parsing functions for squares and triangles where the description starts anywhere on the contour.

```
/**
Prototype: try_all_sqA(alist)

Inputs: String list representation of contour description to be parsed as (any size) square.
        Starts anywhere on valid contour.

Returned Value: Boolean. See examples.

Side Effects: none

Signature: val try_all_sqA : string list -> bool = <fun>

Notes: sqA constraints on contour.
*/
```

Sample Use.

```
# try_all_sqA(["u";"r";"d";"l"]);;
- : bool = true
# try_all_sqA(["r";"d";"l";"u"]);;
- : bool = true
# try_all_sqA(["d";"l";"u";"r"]);;
- : bool = true
# try_all_sqA(["d";"l";"u";"u"]);;
- : bool = false
# try_all_sqA(["d";"d";"l";"l";"u";"u";"r";"r"]);;
- : bool = true
```

4.9 try_all_eqtriA

```
/**
Prototype:  try_all_eqtriA(alist)

Inputs: list to be parsed as (any size) equilateral triangle
        (independent of contour PDL description start).

Returned Value: Boolean

Side Effects: none
```

```
Signature: val try_all_eqtriA : string list -> bool = <fun>

Notes: Uses all_cases and eqtriA constraints.
*/
```

Sample Use.

```
# try_all_eqtriA(["u"; "m30"; "p240"]);;
- : bool = true
# try_all_eqtriA(["m30"; "p240"; "u"]);;
- : bool = true
# try_all_eqtriA(["p240"; "u"; "m30"]);;
- : bool = true
# try_all_eqtriA(["p240"; "m30"; "u"]);;
- : bool = false
```

5 How We Will Grade Your Solution

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

```
#use "which_shape.caml";;          /* YOUR ocaml source -- all the required functions */
#use "grading_sde2.caml"          /* OUR testing of individual 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.

So that we may gain experience with functional programming, *only the applicative (functional) features of* `ocaml` *are to be used*[1]. **Please reread the previous sentence.** This rules out the use of `ocaml`'s imperative features.

[1]Anywhere in your ocaml source file, i.e., this includes any auxiliary or 'helper' functions you develop.

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 only functions you may use are those in the Pervasives module (excluding the imperative functions and constructs cited previously) and the 4 cited below:

```
List.hd
List.tl
List.nth
List.length
```

This, of course, means you may not use the Array Module or other functions, like List.map. Furthermore, you may not use the open construct. To use one of the 4 non-pervasives module allowed functions above, they must be invoked as List.`<allowed function name>`.
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.

7 Format of the Electronic Submission

The final **zipped** archive is to be named **`<yourname>`-sde2.zip**, where `<yourname>` is your (CU) assigned user name. You will upload this to the Canvas 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 `which_shape.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 and a log of 5 sample uses of the resulting main function (`cmeans`). Name this log file `<yourname>-sde2.log`.

The use of `ocaml` should not generate any errors or warnings. The grader will attempt to build your executable. Recall the grade is based upon a correctly working solution.

8 Final Remark: Deadlines Matter

This is the same remark included in the SDE 1 assignment. Since multiple submissions to Canvas are allowed[2], 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 to either me or the graders. There will be a penalty for this.**

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