ECE/CPSC 3520
Fall 2017
Software Design Exercise #2

Canvas submission only

Assigned 10/9/2017; Due 11/1/2017 11:59 PM

Contents

1 Preface

1.1 Objectives

The objective of SDE 2 is to implement the CYK parsing algorithm (CYK table formation) in `ocaml`. This document specifies a number of functions which must be developed as part of the effort. **Of extreme significance** is the restriction of the implementation to pure functional programming, i.e., no imperative constructs are allowed and and some other features are excluded. This may make you unhappy and uncomfortable, but almost guarantees you will learn something new.

This assignment, given the objective and constraints, is challenging. Significant in-class discussion will accompany this document. The overall motivation is to:

- Learn the paradigm of (pure) functional programming;
- Implement a (purely) functional version of an interesting algorithm;
- 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` class lectures;
3. The background provided in this document;
4. **In-class discussions**, demonstrations and examples[1]; and
5. The `ocaml` reference manual (RTM).

You may use any linux version of `ocaml` $\geq$ 4.0.0

[1] Miss these at your peril.

1.3 Standard Remarks

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. As noted, we will continue to discuss this in class.

2 Data Structures and Representation in `ocaml`

2.1 Productions and Input String

```
(* all productions (in CNF)
   here let is used for illustration *)

(* productions *)

(* S -> AB *)
let prod1 = ["S"; "AB"];;  (* Note: NOT ["S";"A";"B"] *)
val prod1 : string list = ["S"; "AB"]

(* A -> a *)
let prod2 = ["A"; "a"];;
val prod2 : string list = ["A"; "a"]

(* string to parse *)
let astring = ["a"; "a"; "b"; "b"];;
val astring : string list = ["a"; "a"; "b"; "b"]
*)
```

2.2 CYK Table Structure

Here I show the structure of the table for an input string of length n.

```
(* pseudocode -----

let table = [<row_1>; <row_2>;<row_3>;...<row_n>]
```

```
where

|<row_1>| = n, |<row_2>| = n-1, ..., |<row_n>| = 1

and

<row_i> = [<cell_1>; <cell_2>; <cell_3>; ...<cell_j>]

where cell_i = [<nonterminal_symbols>]

----*)
```

2.3 An Example-Based Overview

The use of `let` in the following examples is only for illustration and naming of the inputs. `let` should not appear in your `ocaml` source for anything except naming functions. See Section 5.

Let's get right to work. Examine the session below. Note the functions shown are ones you will design, implement and test (i.e., don't look for them in your `ocaml` installation).

```
# bookprods;;
- : string list list =

[["S"; "AB"]; ["S"; "BB"]; ["A"; "CC"]; ["A"; "AB"]; ["A"; "a"]; ["B"; "BB"];
 ["B"; "CA"]; ["B"; "b"]; ["C"; "BA"]; ["C"; "AA"]; ["C"; "b"]]

# bookstring;;
- : string list = ["a"; "a"; "b"; "b"]

# form_row1;;
- : 'a list * 'a list list -> 'a list list = <fun>

# form_row1(bookstring,bookprods);;
- : string list list = [["A"]; ["A"]; ["B"; "C"]; ["B"; "C"]]

# form_cyk_table;;
- : string list list * string list list list -> string list list list = <fun>

# form_cyk_table(bookprods,[form_row1(bookstring,bookprods)]);;
- : string list list list =
[[["A"]; ["A"]; ["B"; "C"]; ["B"; "C"]];
 [["C"]; ["S"; "A"]; ["S"; "B"; "A"]]; [["C"; "A"]; ["C"; "S"; "A"]];
```

```
 [["C"; "B"; "S"; "A"]]]

(* look familiar? Look at page 145 of the text.
   Of course, lots of work is buried in this top level function.
   To see the structure of the resulting table, consider access *)

# let book_result= form_cyk_table(bookprods,[form_row1(bookstring,bookprods)]);;
val book_result : string list list list =
  [[["A"]; ["A"]; ["B"; "C"]; ["B"; "C"]];
   [["C"]; ["S"; "A"]; ["S"; "B"; "A"]]; [["C"; "A"]; ["C"; "S"; "A"]];
   [["C"; "B"; "S"; "A"]]]

#  get_table_values_cell ([1;1],book_result);;
- : string list = ["A"]
#  get_table_values_cell ([1;2],book_result);;
- : string list = ["C"]
#  get_table_values_cell ([1;3],book_result);;
- : string list = ["C"; "A"]
#  get_table_values_cell ([2;1],book_result);;
- : string list = ["A"]
#  get_table_values_cell ([2;2],book_result);;
- : string list = ["S"; "A"]
#  get_table_values_cell ([3;2],book_result);;
- : string list = ["S"; "B"; "A"]
#  get_table_values_cell ([1;4],book_result);;
- : string list = ["C"; "B"; "S"; "A"]
```

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

Notes:

- **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 some of the required functions.
- **Carefully note the argument interface (tupled) on all multiple-argument functions you will design, implement and test**. This may also be verified by the signatures.

- 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 function signatures indicate polymorphic behavior. This is OK.
- I recommend you attempt function development in the order they are listed.

3.1 add_to_cell

```
(**
Prototype: add_to_cell(symbol,cell)
Input(s): string symbol and possibly nonempty cell (string list)
Returned Value: symbol added to cell, if not already there
Side Effects: none
Signature: 'a * 'a list -> 'a list = <fun>
Notes: Used to add string to cell (list), w/o repeats
       May design and implement as polymorphic
*)
```

Sample Use.

```
# add_to_cell("A",[]);;
- : string list = ["A"]

# add_to_cell("B",["A"]);;
- : string list = ["B"; "A"]

# add_to_cell("A",["B";"A"]);;
- : string list = ["B"; "A"]

# add_to_cell("A",["C";"A";"B"]);;
- : string list = ["C"; "A"; "B"]
```

3.2 form_row1_cell

```
(**
Prototype: form_row1_cell(element,productions)
Input(s): string to parse, production list
Returned Value: cell (string list) containing LHS of production
                capable of producing  element
```

```
Side Effects: none
Signature: 'a * 'a list list -> 'a list = <fun>
*)
```

Sample Use.

```
# form_row1_cell ("a",productions);;
- : string list = ["A"]
# form_row1_cell ("b",productions);;
- : string list = ["B"; "C"]
```

3.3 form_row1

```
(**
Prototype: form_row1(string,productions)
Input(s): input string to be parsed, as a string list
          productions in CNF
Returned Value: first row of CYK table
Side Effects: none
Signature: 'a list * 'a list list -> 'a list list = <fun>
Notes: Forms row 1 of CYK table as a special case.
*)
```

Sample Use.

```
# form_row1(bookstring,bookprods);;
- : string list list = [["A"]; ["A"]; ["B"; "C"]; ["B"; "C"]]
```

3.4 decompositions

Decompositions indicate how to produce a string in 2 parts. They are the key to the efficiency of the CYK algorithm and the requirement that productions are in CNF.

```
(**
Prototype: decompositions (n)
Inputs: integer (string length)
Returned Value: list of decomposition lists, each of form [j;k]
Side Effects: none
Signature: int -> int list list = <fun>
Notes: Notation [j;k] in this function means a list of length j
       followed by a list of length k
        i.e., the 'j+k' decomposition.
        This is not cell j,k in the CYK table.
*)
```

Sample Use.

```
# decompositions(2);;
- : int list list = [[1; 1]]
# decompositions(3);;
- : int list list = [[1; 2]; [2; 1]]
# decompositions(4);;
- : int list list = [[1; 3]; [2; 2]; [3; 1]]
# decompositions(5);;
- : int list list = [[1; 4]; [2; 3]; [3; 2]; [4; 1]]
```

3.5 find_targets_in_all_decompositions

```
(**
Prototype: find_targets_in_all_decompositions(i,j)
Inputs: cell indices for cell t_ij in the CYK table
Returned Value: list of list of all cell pair indices to be tested
Side Effects: none
Signature: int * int -> int list list list = <fun>
Notes: Used to find cell pairs in binary decomposition of the string.
*)
```

Sample Use.

```
(* sample case: n=4

2nd row

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

3rd row

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

4th row

# find_targets_in_all_decompositions(1,4);;
```

```
- : int list list list =
[[[1; 1]; [2; 3]]; [[1; 2]; [3; 2]]; [[1; 3]; [4; 1]]]
*)
```

3.6 cell_products

```
(**
Prototype: cell_products([cell1;cell2])
Inputs: list of 2 cell lists (cell pairs)
Returned Value: 'product' of the 2 cells for CYK table
Side Effects: none
Signature: string list list -> string list = <fun>
Notes: See examples
*)
```

Sample Use.

```
# cell_products[["A";"B"];["D";"E";"F";"G"]];;
- : string list = ["AD"; "AE"; "AF"; "AG"; "BD"; "BE"; "BF"; "BG"]
# cell_products [["A";"B"];[]];;
- : string list = []
# cell_products[[];["D";"E";"F";"G"]];;
- : string list = []
# cell_products[["A";"B"];["D"]];;
- : string list = ["AD"; "BD"]
```

3.7 form_cyk_table

This is the top-level function. All of the preceding functions (and certainly others) may be useful for implementation of this function.

```
(**
Prototype: form_cyk_table(prods,ctable)
Inputs: productions, current table
Returned Value: the CYK parse table
Side Effects: none
Signature: string list list * string list list list -> string list list list = <fun>
Notes: This function is started with ctable = [row1].
*)
```

Sample Use.

```
# form_row1(bookstring,bookprods);;
- : string list list = [["A"]; ["A"]; ["B"; "C"]; ["B"; "C"]]

# form_cyk_table(bookprods, [form_row1(bookstring,bookprods)]);;
- : string list list list =
[[["A"]; ["A"]; ["B"; "C"]; ["B"; "C"]];
 [["C"]; ["S"; "A"]; ["S"; "B"; "A"]]; [["C"; "A"]; ["C"; "S"; "A"]];
 [["C"; "B"; "S"; "A"]]]

--- from quiz 1-------

# quizstring1;;
- : string list = ["d"; "c"]
# quizstring2;;
- : string list = ["c"; "d"]

# quizprods;;
- : string list list =
[["S"; "AB"]; ["S"; "d"]; ["A"; "d"]; ["A"; "a"]; ["B"; "b"]; ["B"; "c"];
 ["C"; "d"]]

# form_cyk_table(quizprods,[form_row1(quizstring1,quizprods)]);;
- : string list list list = [[["S"; "A"; "C"]; ["B"]]; [["S"]]]

# form_cyk_table(quizprods,[form_row1(quizstring2,quizprods)]);;
- : string list list list = [[["B"]; ["S"; "A"; "C"]]; [[]]]
```

4 How We Will Grade Your Solution

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

```
#use "sde2.caml";;           (* YOUR ocaml source -- all the required functions
                                and any additional (supporting) functions you develop*)
#use "gradeit.caml";;         (* OUR TEST inputs and scripts*)
<testing>                    (* sample invocation of the required functions *)
```

The grade is based primarily upon a correctly working solution.

5 `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.

5.1 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 strictly prohibited.

5.2 Only Pervasives Module and the List and String Modules Are Allowed

The only modules you may use (other than Pervasives) are the List and String modules. Anything else inhibits further grading of your submission.

5.3 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*)
```

5.4 No (Nested) Functions

`ocaml` allows 'functions defined within functions' definitions (another 'illegal' `let` use for SDE2). Here's an example of a nested function definition:

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

5.5 Summary of the Constraints

```
0. All source must be ocaml.
1. No sequences.
2. No function definitions within functions.
3. No let use, except function naming. If you use let in 'illegal' ways,
   you can quickly turn your 'perfect' solution into a 20/100 solution.
   A 'perfect' solution is functionally
   correct AND meets the constraints in this document.
4  nth uses 0-based indexing. 1-based indices are used in book CYK tables.
5. All user-developed functions (to be tested) have tupled interface
6. Implementation assumes productions are given in CNF;
   a later class can implement this test.
7. Allowable modules are (Pervasives) String and List (i.e., no Array module).
```

5.6 Apriori Appeals

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

The objective of this SDE is to obtain proficiency in functional programming, not to try to find built-in `ocaml` functions or features which simplify or trivialize this SDE. I want you to come away from SDE 2 with a perspective on (almost) pure functional programming (no side effects).

6 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. Multiple submissions are allowed, so if it looks like you will not succeed with function `form_cyk_table`, submit the other functions when they are finished and before 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 all your function implementations. The file is to be named `sde2.caml`. Note this file must include all the functions defined in this document. It may also contain other 'helper' or auxiliary functions you developed. Do not include test data (strings or productions). We will provide these.

3. A log of 2 sample uses of each of the required functions. Name this log file `sde2.log`. Use something other than my examples.

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.