ECE/CPSC 3520
Spring 2018
Software Design Exercise #2

Canvas submission only

Assigned 3/13/2018; Due 4/10/2018 11:59 PM

Contents

1 Preface

1.1 Objectives

The objective of SDE 2 is to implement parts of the CYK parsing algorithm (CYK table formation) in `ocaml`. This effort is analogous to SDE1, albeit using a different paradigm and implementation language. Also, some of the parts are not contained in SDE1.

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 `ocaml` 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

[1] Miss these at your peril.

5. The `ocaml` reference manual (RTM).

You may use any linux version of `ocaml` $\geq$ 4.2.0

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`

In the list-based representation of a production, we assume only one upper or lowercase character used per terminal/nonterminal. The RHS of a production is one (a single) string. Recall that order matters in the RHS string. **Carefully study the types and structures in the examples below**.

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"]

(* B -> b *)
let prod3 = ["B"; "b"];;
val prod3 : string list = ["B"; "b"]
```

```
let prod4 = ["C"; "b"];;
val prod4 : string list = ["C"; "b"]

(* all productions (in CNF)
   here a global variable is used for illustration *)

let productions = [prod1;prod2;prod3;prod4];;
val productions : string list list =
  [["S"; "AB"]; ["A"; "a"]; ["B"; "b"]; ["C"; "b"]]

(* 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.

```
(* 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>]

----*)
```

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 or revise any of the arguments. Reread the preceding three sentences at least 3 times. You must learn to program to an API.
- 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 get_table_values_cell

```
(**
Prototype: get_table_values_cell([i;j],table)
Input(s): tuple of ([<column>;<row>], table)
Returned Value: cell with string values
Side Effects: none
Signature: val get_table_values_cell : int list * 'a list list -> 'a = <fun>
*)
```

Sample Use.

```
let sample_table4 = [[["11"];["21"];["31"];["41"]];
[["12"];["22"];["32"]];[["13"];["23"]];[["14"]]];;

# get_table_values_cell ([3;2],sample_table4);;
- : string list = ["32"]
# get_table_values_cell ([1;4],sample_table4);;
- : string list = ["14"]
```

3.2 cell_products

This function forms the 'outer product' of 2 cells. Either cell could be empty. It can be used for any location in the table other than the first row, and should return a single list of strings, unless it is empty.

```
(**
Prototype: cell_products [cell1;cell2]
Input(s): list containing 2 cells
Returned Value: resultant list of strings
Side Effects: none
Signature: val cell_products : string list list -> string list = <fun>
*)
```

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"]
```

3.3 form_row1_cell

```
(**
Prototype: form_row1_cell(element,productions)
Input(s): tuple of single terminal element, productions list
Returned Value: corresponding cell in first row of CYK table
Side Effects: none
Signature: val form_row1_cell : 'a * 'a list list -> 'a list = <fun>
Notes: Forms row 1 cells of CYK table as a special case.
*)
```

Sample Use.

```
reminder: val productions : string list list =
  [["S"; "AB"]; ["A"; "a"]; ["B"; "b"]; ["C"; "b"]]

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

3.4 equiv

This Boolean function tests for cell equivalence. Note that the order of nonterminals in a CYK parse table cell is arbitrary.

```
(**
Prototype: equiv(ca, cb)
Inputs: tuple of 2 cells
Returned Value: true or false
Side Effects: none
Signature: val equiv : 'a list * 'a list -> bool = <fun>
*)
```

Sample Use.

```
# equiv([1;5;3;0],[1;5;3]);;
- : bool = false
# equiv([1;5;3;0],[1;5;3;0]);;
- : bool = true
# equiv([1;5;3;0],[5;3;0;1]);;
- : bool = true
# equiv([1;6;3;0],[5;3;0;1]);;
- : bool = false
# equiv([],[]);;
- : bool = true
```

3.5 row_equivalent

```
(**
Prototype: row_equivalent(rowA,rowB)
Inputs: tuple of 2 rows
Returned Value: true or false
Side Effects: none
Signature: val row_equivalent : 'a list list * 'a list list -> bool = <fun>
```

Sample Use.

```
# let row1 = [["A"]; ["A"]; ["B"; "C"]; ["B"; "C"]];;

# let row1mod = [["A"]; ["A"]; ["C"; "B"]; ["C"; "B"]];;

# let row1mod2 = [["A"]; ["A"]; ["C"; "C"]; ["B"; "B"]];;

# let row1mod3 = [["A"]; ["B"; "C"]; ["A"]; ["B"; "C"]];;
```

```
# row_equivalent(row1,row1mod);;
- : bool = true
# row_equivalent(row1,row1mod2);;
- : bool = false
# row_equivalent(row1,row1mod3);;
- : bool = false
```

3.6 table_equivalent

```
(**
Prototype: table_equivalent(tableA,tableB)
Inputs: tuple of 2 tables
Returned Value: true or false
Side Effects: none
Signature: val table_equivalent :
           'a list list list * 'a list list list -> bool = <fun>
Notes:
*)
```

Sample Use.

```
val tablebook : 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"]]]
val tablebook2 : string list list list =
  [[["A"]; ["A"]; ["B"; "C"]; ["B"; "C"]];
   [["C"]; ["A"; "S"]; ["B"; "A"; "S"]]; [["C"; "A"]; ["A"; "S"; "C"]];
   [["S"; "B"; "C"; "A"]]]
val tablebook3 : string list list list =
  [[["A"]; ["A"]; ["B"; "C"]; ["B"; "C"]];
   [["C"]; ["A"; "G"]; ["B"; "A"; "S"]]; [["C"; "A"]; ["A"; "S"; "C"]];
   [["S"; "B"; "C"; "A"]]]

# table_equivalent(tablebook,tablebook);;
- : bool = true

# table_equivalent(tablebook,tablebook2);;
- : bool = true

# table_equivalent(tablebook,tablebook3);;
- : bool = false
```

3.7 valid_production

```
(**
Prototype: valid_production production
Inputs: a list
Returned Value: true or false
Side Effects: none
Signature: val valid_production : string list -> bool = <fun>
Notes: true if production is valid format and CNF
*)
```

Sample Use.

```
/* other productions defined previously */
# let prodb1 = ["s"; "AB"];;

# let prodb2 = ["S"; "A"];;

# let prodb3 = ["AB"; "CD"];;

# let prodb4 = ["s"; "cAB"];;

# valid_production prod1;;
- : bool = true
# valid_production prod2;;
- : bool = true
# valid_production prodb1;;
- : bool = false
# valid_production prodb2;;
- : bool = false
# valid_production prodb3;;
- : bool = false
# valid_production prodb4;;
- : bool = false
```

3.8 valid_production_list

For this function to return true, all productions in the production list must be valid CYK-format productions.

```
(**
Prototype: valid_production_list productionList
Inputs: list of productions
Returned Value: true or false
Side Effects: none
```

```
Signature: val valid_production_list : string list list -> bool = <fun>
Notes:
*)
```

Sample Use.

```
let prods1 = [prod1;prod2;prod3;prod4];;

let prods2 = [prod1];;

let prods3= [prod1;prod2;prod3;prodb4];;

let prods4 = [prod1;prod2;prodb3;prod4];;

let prods5 = [prod1;prodb2;prod3;prod4];;

(* samples:

# valid_production_list prods1;;
- : bool = true
# valid_production_list prods2;;
- : bool = true
# valid_production_list prods3;;
- : bool = false
# valid_production_list prods4;;
- : bool = false
# valid_production_list prods5;;
- : bool = false
```

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 Functions in the Pervasives Module and the Following Functions in Other Modules are Permitted in Your Solution

The allowable functions in SDE 2 are those non-imperative functions in the Pervasives module and these 9 individual functions listed below:

```
List.hd
List.tl
List.nth
List.length
List.append
String.length
String.get
String.concat
Char.code
```

No modules (other than Pervasives) may be opened. In other words, you cannot open any modules. There cannot be an `open List;;` or `open String;;`,

or any similar statement in your source file. Each of the above 9 functions must be used with the proper Module name. This also eliminates namespace ambiguity.
This constraint actually helps most of the class. Note you may not need all of these functions. Since no other functions are allowed, you can stop searching the ocaml libraries for something to trivialize the function development challenges in this assignment.

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

Use this as a checklist before submission.

0. All source must be ocaml.

1. No sequences.

2. No function definitions within functions (nested functions).

3. No use of `let`, except function naming. Note let used for illustration in this document. If you use let this way, you can quickly turn your 'perfect' solution into a 30/100. A 'perfect' solution is functionally correct AND meets the constraints in this document.

4. `List.nth` in ocaml is 0-based indexing. 1-based indexing is used in the book CYK tables and in this assignment.

5. All user-developed functions (to be tested) have tupled interface.

6. Implementation assumes productions are given in CNF. You will also test for this.

7. Allowable functions: Only those in the Pervasives module and the 9 individual functions listed in Section 5.2.

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[2], so if it looks like you will not succeed with all functions, 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:

[2]Recall they must be self-sufficient, i.e., only the latest submission is downloaded and graded.

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, tables, productions, etc.)**. 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.