ECE/CPSC 3520
SS II 2017
Software Design Exercise #2

Canvas submission only

Assigned 7/21/2017; Due 7/31/2017

Contents

1 Preface

1.1 A Little Perspective

- **The Good News:** We're using the grammars from SDE1.
- **The Bad News:** We're doing other things with them and using `ocaml`.

1.2 Objectives

Simply, the objective of SDE 2 is to implement `ocaml` string recognizers after identifying the $L(G_{Wi})$ corresponding to G_{Wi} from SDE1. Input strings are `ocaml` (not Prolog) formatted lists. **Note we are NOT using ocamllex and ocamlyacc** for this task. This document specifies a number of functions which must be developed as part of the effort.

Of extreme significance is the restriction of the paradigm to pure functional programming, i.e., **no imperative constructs are allowed.** This may make you unhappy and uncomfortable, but almost guarantees you will learn something new.

This effort is straightforward, and, as mentioned, 11 days 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`.

1.3 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` slide and corresponding video lectures;
3. The background provided in this document; and

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

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

1.4 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 via a SDE2 FAQ on Canvas.

2 Preliminary Considerations

2.1 G_{Wi} from SDE1

For simplicity, only the relevant details of the grammars from SDE1 are summarized here. Recall the 3 grammars specified in SDE1 (G_{w1}, G_{w2}, G_{w3}):

G_{w1}
Productions:

$$S \rightarrow D$$

$$S \rightarrow DS$$

$$D \rightarrow ac$$

G_{w2} (Case 2)
Productions:

$$S \rightarrow AC$$

$$A \rightarrow aA$$

$$A \rightarrow a$$

$$C \to cC$$

$$C \to c$$

G_{w3} (Case 3)
Productions:

$$S \to GH$$

$$G \to cG$$

$$G \to c$$

$$H \to aH$$

$$H \to a$$

2.2 Languages Corresponding to G_{Wi} from SDE1

Although it was not necessary to observe this to solve SDE1, each G_{Wi} has a corresponding language, $L(G_{Wi})$ which is expressible in a compact, closed form. This does not mean that the cardinality of $L(G_{Wi})$ is finite.

You should take some time and convince yourself (if you have not done this previously) that the following holds:

$$L(G_{W1}) = (ac)^k \text{ with } k > 0$$

$$L(G_{W2}) = a^k c^j \text{ with } k > 0 \text{ and } j > 0$$

$$L(G_{W3}) = c^k a^j \text{ with } k > 0 \text{ and } j > 0$$

2.3 `ocaml` String Representation

All 3 grammars have the same set of terminals, i.e., $V_T = \{a, c\}$.

In choosing or `ocaml` input string representation, we have a number of choices for the data structure. We restrict our data structure used to represent input to be a character string represented in `ocaml` as an `ocaml` list of characters. The `ocaml` manual indicates:

> char Characters
>
> Characters are ISO-8859-1 values. Character literals are written in single-quotes.

You should also be familiar with the representation of a list in `ocaml` and the constraint that all elements of a list must be of the same type. For example, string "abcd" would be represented as a 4-element `ocaml` list as `['a';'b';'c';'d']`. We can use the `ocaml` interpreter to check this:

```
$ ocaml
        OCaml version 4.02.3

# ['a';'b';'c';'d'];;
- : char list = ['a'; 'b'; 'c'; 'd']
```

3 Prototypes, Signatures and Examples of Functions to be Developed

3.1 Specific Objectives

In SDE2, you will use `ocaml` and $L(G_{Wi})$ to determine the status of an arbitrary input string, x. We are interested in developing functions[1] to detect 3 possible outcomes:

1. $x \in L(G_{W1})$
2. $x \in L(G_{W2})$
3. $x \in L(G_{W3})$

[1] Together with some relatively simple 'auxiliary' functions you might find useful.

3.2 Function Names and Signatures

To begin, here are the signatures of all the `ocaml` functions to be developed:

```
Elementary/Utility:
val tupfirst : 'a * 'b -> 'a = <fun>
val tupsecond : 'a * 'b -> 'b = <fun>
val next2 : 'a list -> 'a list = <fun>
val remainder2 : 'a list -> 'a list = <fun>

For Grammar Recognition:
val countAs : char list -> int * char list = <fun>
val countCs : char list -> int * char list = <fun>
val ack : char list -> bool = <fun>
val ajck : char list -> bool = <fun>
val ckaj : char list -> bool = <fun>
```

In what follows, `inlist` is the `ocaml` list holding the characters (see Section 2.3 and the Examples).

3.3 Easy Tuple and List Utility Functions

The 'elementary/utility' functions serve 2 purposes:

- They probably avoid anyone getting a 0. on this assignment; and
- They may be useful in the design and implementation of the 'grammar recognition' functions.

3.3.1 `tupfirst`

```
(**
Prototype: tupfirst (a,b)
Input(s): tuple (a,b)
Returned Value: a
Side Effects: none
Signature:
val tupfirst : 'a * 'b -> 'a = <fun>
Notes: Tupled interface.
*)
```

Sample Use.

```
# tupfirst (10,['a';'b';'c';'d']);;
- : int = 10
```

3.3.2 tupsecond

```
(**
Prototype: tupsecond (a,b)
Input(s): tuple (a,b)
Returned Value: b
Side Effects: none
Signature:
val tupsecond : 'a * 'b -> 'b = <fun>
Notes: Tupled interface.
*)
```

Sample Use.

```
# tupsecond (10,['a';'b';'c';'d']);;
- : char list = ['a'; 'b'; 'c'; 'd']
```

3.3.3 next2

```
(**
Prototype: next2 inlist
Input(s):  inlist
Returned Value: Given an inlist with length >=2,
returns a list of the first 2 elements, otherwise []
Side Effects: none
Signature:
val next2 : 'a list -> 'a list = <fun>
*)
```

Sample Use.

```
# next2 [1;2;3;4];;
- : int list = [1; 2]
# next2 [1;2;3];;
- : int list = [1; 2]
# next2 [1;2];;
- : int list = [1; 2]
# next2 [1];;
- : int list = []
```

3.3.4 remainder2

```
(**
Prototype:  remainder2 inlist
Input(s):  inlist
Returned Value: Given an inlist with length >=2,
returns a list with 1st 2 elements removed (remainder of list),
otherwise []
Side Effects: none
Signature:
val remainder2 : 'a list -> 'a list = <fun>
*)
```

Sample Use.

```
# remainder2 [1;2;3;4];;
- : int list = [3; 4]
# remainder2 [1;2;3];;
- : int list = [3]
# remainder2 [1;2];;
- : int list = []
# remainder2 [1];;
- : int list = []
```

3.4 countAs

```
(**
Prototype: countAs inlist
Input(s):  inlist
Returned Value: returns tuple consisting of number of *consecutive* 'a'
 elements from the beginning of the list and the remainder of the list
Side Effects: none
Signature:
val countAs : char list -> int * char list = <fun>
*)
```

Sample Use.

```
# countAs ['a';'a';'a';'c';'a';'b'];;
- : int * char list = (3, ['c'; 'a'; 'b'])
# countAs ['c';'a';'a';'c';'a';'b'];;
- : int * char list = (0, ['c'; 'a'; 'a'; 'c'; 'a'; 'b'])
# countAs ['a';'c';'a';'b'];;
- : int * char list = (1, ['c'; 'a'; 'b'])
# countAs ['a'];;
- : int * char list = (1, [])
```

3.5 countCs

This function is very similar to `countAs`, except it counts 'c's.

```
(**
Prototype:  countCs inlist
Input(s):  inlist
Returned Value: returns tuple consisting of number of *consecutive* 'c'
 elements from the beginning of the list as well as the remainder
Side Effects: none
Signature:
val countCs : char list -> int * char list = <fun>
*)
```

Sample Use.

```
# countCs ['c';'a';'b'];;
- : int * char list = (1, ['a'; 'b'])
# countCs ['c';'c';'c'];;
- : int * char list = (3, [])
# countCs ['c';'c';'a';'a'];;
- : int * char list = (2, ['a'; 'a'])
```

3.6 Recognition Function for $L(G(w1))$ (ack)

This function recognizes strings (in `ocaml` list form) $(ac)^k$ $k > 0$.

```
(**
Prototype:  ack input
Input(s):  inlist
Returned Value: true if input is in L(Gw1), otherwise false
Side Effects: none
Signature:
val ack : char list -> bool = <fun>
*)
```

Sample Use.

```
# ack ['a';'c'];;
- : bool = true
# ack ['a';'c';'a';'c';'a'];;
- : bool = false
# ack ['a';'c';'a';'c';'a';'c'];;
- : bool = true
```

```
# ack ['a';'c';'b';'c';'a';'c'];;
- : bool = false
# ack ['a';'c';'a';'c';'a';'a'];;
- : bool = false
# ack ['c';'a';'c';'a';'c'];;
- : bool = false
```

3.7 Recognition Function for $L(G(w2))$ (ajck)

This function recognizes strings (in `ocaml` list form) $a^j c^k$ $j > 0$ $k > 0$.

```
(**
Prototype:  ajck inlist
Input(s):  inlist
Returned Value: true if input is in L(Gw2), otherwise false
Side Effects: none
Signature:
val ajck : char list -> bool = <fun>
*)
```

Sample Use.

```
# ajck ['a';'c'];;
- : bool = true
# ajck ['a';'a';'a';'c';'c'];;
- : bool = true
# ajck ['a';'b';'a';'c';'c'];;
- : bool = false
# ajck ['a';'a';'a';'c';'c';'d'];;
- : bool = false
# ajck ['c';'a';'a';'a';'c';'c';'d'];;
- : bool = false
```

3.8 Recognition Function for $L(G(w3))$ (ckaj)

This function recognizes strings (in `ocaml` list form) $c^k a^j$ $j > 0$ $k > 0$.

```
(**
Prototype:  ckaj inlist
Input(s):  inlist
Returned Value: true if input is in L(Gw3), otherwise false
Side Effects: none
Signature:
val ckaj : char list -> bool = <fun>
*)
```

Sample Use.

```
# ckaj ['c';'a'];;
- : bool = true
# ckaj ['c';'a';'k'];;
- : bool = false
# ckaj ['c';'a';'c'];;
- : bool = false
# ckaj ['c';'c';'c';'a';'a';'a';'a'];;
- : bool = true
```

3.9 Important Notes

Notes:

- `let` should not appear in your `ocaml` source for anything except naming functions. See Section 5.
- **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.
- **Carefully note the argument interface 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 several of the signatures of the 'elementary' functions indicate polymorphic behavior.

4 How We Will Grade Your Solution

The script 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 "inputs.caml";;            (* OUR TEST inputs/cases*)
<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 prohibited.

5.2 Only Pervasives Module and 3 Functions from the List Module Are Allowed

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.

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 Apriori 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 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 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 `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.

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.