ARIZONA STATE UNIVERSITY, COMPUTER SCIENCE

CSE 340 – FALL 2019

Homework 5

Due Wednesday November 27 by 11:59 PM

General Instructions

1. When you submit your solution to GradeScope, you should enter the answers to separate problems separately
2. Your answers should be typed.

[Hindley-Milner Type Inference]. For each of the following definitions, give the type of the function and its arguments. If there is a type mismatch, you should explain the reason for the mismatch. Note that for recursive functions, **`let rec`** is used. If there is no type mismatch, you are only asked to give the answer without explanation. I realize that you can get the answer by typing the code in an OCaml editor, but if you do so, you will guarantee that you will not do well on the final on the Hindley-Milner section.

For the problems below, you can use the following constraints that I did not cover in the notes:

Empty list

The empty list is written []. Its type is T List where T is not constrained

List of one or more elements

A list of one element is written [a]. The type constraint is type of [a] = T_a list

A list of two or more elements is written [a_1 ; a_2 ; … ; a_k]. The type constraints are $Ta_1 = Ta_2 = \ldots = Ta_k = T$ and the type of [a_1 ; a_2 ; … ; a_k] is T list

Appending to a list: The :: operator

If the expression h::L is appears in the program, then L must be a list of elements where all the elements have the same type as the type of h.

Option Types

An option type for type T is a type whose value can be either from T or are undefined (None). The type constraint for the expression None is T type where T is unconstrained. The type of the expression Some x is T_x option where Tx is the type of x.

For example,

```
let f a = if a then Some 1 else None
```

is a function that takes a bool a and returns int option.

Note that recusive functions are declared with `let rec`

Note variable and function names in OCaml start with lowercase.

Pattern matching

Pattern matching is a powerful OCaml expression that implements a "case-like" functionality. The syntax is

```
match e with
| p1 -> e1
| p2 -> e2
…
| pn -> en
```

The type constraints are

$Tp_1 = Tp_2 = \ldots = Tp_n =$
$Te_1 = Te_2 = \ldots = Te_n$ = Type of match e with | p1 -> e1 | p2 -> e2 … | pn -> en

Problem 1. let f x = []

Problem 2. let f x = [[x]]

Problem 3. let f x = [x ; x]

Problem 4. let f x l = x::l

Problem 5. let f g a b = if a (a g) then b else b + 1

Problem 6. let f a b i = if a.(i) b then b i else b (i+1)

Problem 7. let f a b c i = if a c then (if b then i else i+1) else c

Problem 8. let rec max l = match l with
```
    []      -> None                  (* max is not defined for empty list *)
  | h::l1   -> if h > max l1 then h else max l1
```

Problem 9. let rec f l g = match l with (* this is a map-reduce function *)
```
    []      -> 0
  | h::l1   ->  (g h) + (f l1 g)     (* apply g to the head of l and f to l1 (and g)
                                      * and add the results. The net effect is to
                                      * apply a to all the elements and to sum the results up
                                      *)
```
Problem 10. let rec mll l = match l with
```
    []      -> []
  | h::l1   -> [h]::(mll l1)
```

Problem 11. let f a b c = if a b c then c else a c b

Problem 12. let f a b c = if a (b c) then a else b

Problem 13. let f a b c = if a b then a c else b c

Problem 14. let f a b = if a (b (a b)) then 1 else 2

Problem 15. let f a b c = if a b c then c + 1 else a c b