UCLA Computer Science 131 (Spring 2019) Midterm Solutions

1a
q : ('a -> 'a -> 'b) -> 'a -> 'b
The function q takes a function f and argument x, and calls f with two arguments (both x).
s : float -> float
The function s squares its input float argument.
p : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b
The function call (p f g) creates a curried function equivalent to the function composition ($f \circ g$).
c : float -> float
The function call (c x) returns $\sin^2(x) = (\sin(x))^2$.

1b
- (*.) is equivalent to (fun x y -> x *. y) in meaning but is shorter and does not use extra memory.
- (*.) would be parsed as the start of a comment in OCaml.
- *. would be parsed as an infix operator, as if we are evaluating the result of q times something nonexistent, resulting in a syntax error.

2a
(Using List.init)
```
let loltp l = match l with
  | [] -> []
  | hd :: _ ->
      let len = List.length hd in
      List.init len (fun i -> List.map (fun sl -> List.nth sl i) l)
```

(Using recursion and List.hd/tl)
```
let rec loltp l = match l with
  | [] -> []
  | hd :: _ -> match hd with
      | [] -> []
      | _ ->
          let col = List.map List.hd l in
          let rest = loltp (List.map List.tl l) in
          col :: rest
```

2b
'a list list -> 'a list list

2c
[["a"; "b"]; ["c"]], as my implementation expects all sublists to have the same length.

3a
We cannot reasonably solve this problem in general. A solution that works on 2×3:

```
let tottp ((a, b, c), (d, e, f)) =
  ((a, d), (b, e), (c, f))
```

This is due to the fact that OCaml's type system forces a function that takes in a tuple to only support tuples of a particular length. It is impossible to write a function that consumes both a pair and a triple, for instance. (Polymorphism represents an exception, for extremely simple functions like (fun x -> x), but here we need to actually read the tuple.)

3b
('a * 'b * 'c) * ('d * 'e * 'f) -> ('a * 'd) * ('b * 'e) * ('c * 'f)

3c
No such value is possible. Unlike in 2c, the structure of the input is statically checked to be that of a 2×3 tuple matrix. My implementation does not rely on any information other than the type to operate correctly.

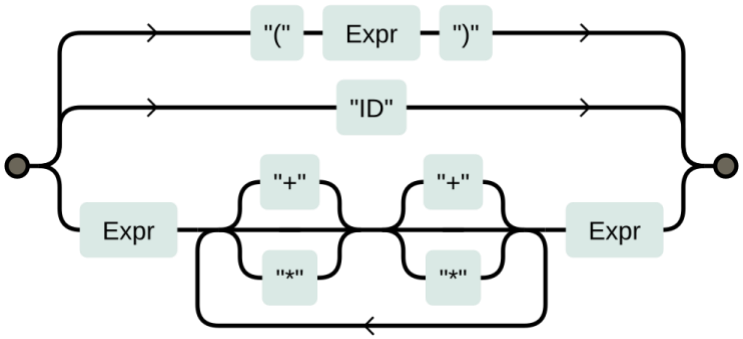
4a
```
(* determine if a rule is nullable given current set of nullables *)
let rec is_nullable nulls = function
  | [] -> true
  | hd :: tl -> match hd with
      | T _ -> false
      | N n -> if subset [n] nulls
            then is_nullable nulls tl
            else false

(* return new set of nullables given grammar and current set of
 * nullables *)
let rec new_nulls gram nulls = match gram with
  | [] -> nulls
  | (nt, rhs) :: tl ->
      if is_nullable nulls rhs
      then new_nulls tl (set_union [nt] nulls)
      else new_nulls tl nulls

(* if set of nullables no longer changes, the search is complete *)
let nullables G = computed_fixed_point equal_sets (new_nulls G) []
```

4b
Yes. Consider the input ["ID"]; the matcher would try to derive the following rule after realizing that "ID" is not "(":

Expr → *Expr Ops Expr*

but as the rule is left-recursive, the matcher would try to derive *Expr* again recursively, falling into infinite loop as a result.

4c
Something like the following suffices.



5
In Java, upcasting a variable to a supertype is allowed:

```
Integer a = new Integer(…);
Object  b = a;  // allowed, b == a (address comparison)
```

However, allowing that with List<Integer> and List<Object> could be problematic:

```
List<Integer> li = …
List<Object>  lo = li;
lo.add(new Thread());
Integer i = li.get();  // here, the first element of li is
                       // in fact a Thread
```

OCaml has no such problem, since all types are statically resolved (so 'a list would become the specialized type at compile time).

In Java, List<Integer> is a subtype of List<? super Integer>.

6
The function type *with _Noreturn* is a subtype of the function type *without*. Even with _Noreturn, a compiler can still save the call's return address, just like it does with ordinary functions. But with _Noreturn, the function has an *additional* operation: call w/o saving return address. Since a _Noreturn function has an operations set that is a strict superset of that of w/o _Noreturn, _Noreturn is a subtype.

7
(This response was graded 8/12: "Correct answer. Arguments somewhat developed/not fully correct.")
This idea is possible, but impractical. With multiple threads executing in parallel, for Java to be able to arbitrate which thread runs first, it would have to incur *significant* runtime overhead in synchronization, even if there is no contention. Additionally, the process of identifying sites of possible parallelism is very difficult at compile-time: it is hard to guarantee that the compiler finds all the places that only one thread operates on, as it cannot predict runtime behavior. Finally, with this proposal many parallel applications simply wouldn't work. If three threads are concurrently updating the same resource, Java would completely block the other two threads until the first finishes execution, defeating the purpose of parallelization. The crux here is the guarantee that "so long as the user can't tell the difference": the user sometimes wants to tell that other threads are doing work!

Even though it might be possible to implement the idea, the process would be difficult and the results defeat the point of parallelization.