UCLA Computer Science 131 (Spring 2015)
Midterm
100 minutes total, open book, open notes

Name: Alfred Lucero Student ID: 604251044

1	2	3	4	total
11	27	12	6	56

1. Consider the following code, which is simplified from a previous Homework 2 hint.

```
type nucleotide = A | C | G | T
type fragment = nucleotide list
type acceptor = fragment -> fragment option
type matcher = fragment -> acceptor ->
                 fragment option
type pattern = | Frag of fragment
               | List of pattern list
               | Or of pattern list

let match_empty frag accept = accept frag

let match_nothing frag accept = None

let match_nucleotide nt frag accept =
  match frag with
    | [] -> None
    | n::t -> if n == nt then accept t else None

let append_matchers matcher1 matcher2 frag accept
  = matcher1 frag
      (fun frag1 -> matcher2 frag1 accept)

let make_appended_matchers make_a_matcher ls =
  let rec mams = function
    | [] -> match_empty
    | h::t ->
       append_matchers (make_a_matcher h)
                       (mams t)
  in mams ls
```

```
let rec make_or_matcher make_a_matcher = function
  | [] -> match_nothing
  | h::t ->
      let h_matcher = make_a_matcher h
      and t_matcher =
            make_or_matcher make_a_matcher t
      in fun frag accept ->
           let ormatch = h_matcher frag accept
           in match ormatch with
                | None -> t_matcher frag accept
                | _ -> ormatch

let rec make_matcher = function
  | Frag frag -> make_appended_matchers
                   match_nucleotide frag
  | List pats ->
      make_appended_matchers make_matcher pats
  | Or pats -> make_or_matcher make_matcher pats
```

1a (15 minutes). When matching a concatenation of the patterns A and B, this implementation walks through the input string left to right, attempting to match A first, and then to match B on what remains when the A matcher is done. Modify the implementation so that it does it the other way: i.e., it (inefficiently) matches B first, and then A. If you consider a sequence of tokens abc (where a, b, and c are all token sequences, and abc represents their concatenation) your implementation should first verify that B matches b, and should then verify that A matches a, where c is the unmatched suffix. When breaking the input string into the three parts a, b, and c, your implementation should try the shortest a first, then the next shortest a, and so forth. Simplicity is more important than efficiency in your implementation.

1b (10 minutes). Give an example grammar that will parse input differently depending on whether you use the original or the modified version. Give an example input string and the two parses generated by the two versions of the code.

2. Consider the following EBNF grammar for a subset of OCaml type expressions as used in the homeworks.

```
1 typeexpr ::= ' ID
2   | ( typeexpr )
3   | typeexpr -> typeexpr
4   | typeexpr { * typeexpr }+
5   | typeconstr
6   | typeexpr typeconstr
7   | ( typeexpr { , typeexpr } ) typeconstr
8 typeconstr ::= ID
```

2a (8 minutes). Write a sample OCaml program that exemplifies each line of this grammar. Put a "1" next to the part of your program that exemplifies line 1 of the grammar, and so on for "2" through "8".

2b (5 minutes). Convert the grammar to the BNF format used in Homework 2 as straightforwardly as possible. Don't fix its ambiguities.

2c (5 minutes). Prove that the BNF-format is ambiguous.

2d (8 minutes). Rewrite the BNF grammar to make it unambiguous. Resolve ambiguities in the way that they are normally done in OCaml. If it doesn't matter whether an operator is left- or right-associative, make it left-associative. Keep your changes as simple as possible.

2e (5 minutes). Convert the unambiguous grammar to a railroad diagram or syntax diagram that's equivalent. Make the diagram as simple and clean as possible.

2f (10 minutes). If you gave your unambiguous BNF grammar to a working implementation of Homework 2, would it actually parse that part of OCaml without any problems? If so, give some OCaml code to invoke the implementation. If not, explain the problems you'd run into.

3. In Java, order of evaluation is defined to be left-to-right, whereas C and C++ compilers are allowed to evaluate expressions in any order, and even to interleave evaluation of subexpressions.

3a (4 minutes). Explain why Java's approach is better for portability. Give a brief example.

3b (4 minutes). Explain why C's approach is better for efficiency, again with a brief example.

3c (10 minutes). How does Java's order-of-evaluation rule interact with the Java Memory Model? Give brief examples of (i) a data-race-free program with competing side effects in subexpressions, and (ii) a buggy program with a data race due to competing side effects in subexpressions.

3d (6 minutes). Suppose C's memory model were the same as Java's. How would your answer to (c) differ?

4 (10 minutes). Would it possible, and would it make sense, to build a C and/or C++ compiler that generates Java byte codes? The idea is to run C and/or C++ programs atop the Java virtual machine. If this idea is impossible or impractical, explain why. If it would work, explain any difficulties or problems you'd see with it. Either way, give the most important advantages of using this approach over the traditional one used by GCC and other C and C++ compilers.