UCLA Computer Science 131 (winter 2019) midterm
100 minutes total, open book, open notes,
no computer or any other automatic device
Please be brief in answers; excessively long answers will be penalized.

Name: Saman Hashemipour　　Student ID: 904903562

```
-----+-----+-----+-----+-----+
1    |2    |3    |4    |5    |total
     |     |     |     |     |
     |     |     |     |     |
-----+-----+-----+-----+-----+
```

1. In C and C++, the type qualifier 'volatile' prohibits a compiler from optimizing away accesses to storage. For example, the declaration 'int volatile x;' means whenever the program evaluates x the implementation must actually load from x's location, and whenever the program assigns to x the implementation must actually store into x's location; a compiler is not allowed to optimize away loads and stores by caching x's value in registers. Similarly, given the declaration 'int volatile *p;', the compiler is not allowed to optimize away accesses to the integer *p (though it may optimize away accesses to the pointer p itself).

1a (10 minutes). Given the above, is (i) 'int volatile *' a subtype of 'int *', or (ii) 'int *' a subtype of 'int volatile *', or both (i) and (ii), or neither (i) nor (ii)? Justify your answer by appealing to general principles.

(i) int volatile * is a subtype of int * because of the fact that volatile is making a modification to the original int* definition. since int* is the original type in C++ it is clear to see that the type qualifier volatile makes a modification to the default type int*

1b (6 minutes). Give an example of what specifically could go wrong if a program violates the rule you specified in (a) and the compiler does not diagnose the violation; or if nothing could go wrong then explain why not.

If the compiler views int as a subtype of its volatile counterpart when the programmer uses int* (which is now assumed to be a volatile ~~typ~~ subtype) we will have all the operations of the supertype inherited by the subtype and therefore the call to int* will result in the same lack of storage optimization ~~or~~ as the int volatile* call.

2. "Reverse currying" a two-argument function F is like currying F, except that the curried version of F takes F's second argument instead of F's first argument. For example, if M is the reverse-curried version of the binary subtraction function, then ((M 3) 5) returns 5-3, or 2.

2a (8 minutes). Define a function (reverse_curry2 x) that accepts the curried version x of a two-argument function F, and returns the reverse-curried version of F. For example, ((reverse_curry2 (-)) 3 5) should yield 2, since (-) is the curried version of binary subtraction. Be as brief as you can.

let reverse_curry2 x -> fn a b = fn x b a;

2b (2 minutes). What is the type of reverse_curry2?

fn : 'a * 'a -> 'a

2c (2 minutes). Give the long version of your definition of reverse_curry2, that is, without using any syntactic sugar.

Reverse_curry2 assigns it

2d (6 minutes). What does the expression (reverse_curry2 reverse_curry2) return, exactly, and what is the type of this expression?

It returns the original curried expression. The type of this expression is fn: 'a -> 'a

2e (10 minutes). Suppose we want to generalize the notion of reverse_curry2 to n-argument functions. That is, we want to write a function reverse_curry such that (reverse_curry n x) accepts the curried version x of an n-argument function F, and returns a reverse-curried function that accepts F's last argument and returns a function that in turn accept's F's second-to-last argument, and so forth. Once we have reverse_curry, we can implement reverse_curry2 via 'let reverse_curry2 = reverse_curry 2'. Give an implementation of reverse_curry and its type, or explain why you can't.

```
let rec reverse_curry n x =
        if n = 0 then x
        else reverse_curry (n-1) x + x ;;
```

type: fn: int * 'a -> 'a

3. Consider the following EBNF grammar for a subset of OCaml type expressions:

```
typexpr ::=      ' ident
        |        _
        |        ( typexpr )
        |        [[?]lowercase-ident :] typexpr ->  typexpr
        |        typexpr { * typexpr }+
        |        lowercase-ident/
        |        typexpr lowercase-ident*
        |        ( typexpr { , typexpr } ) lowercase-ident○
        |        typexpr as ' ident
        |        < [..] >
        |        # lowercase-ident/
        |        typexpr # lowercase-ident*
        |        ( typexpr { , typexpr } ) # lowercase-ident○
```

This grammar uses one reserved word, 'as', and three kinds of identifiers: 'ident' (any identifier), 'lowercase-ident' (an identifier that begins with a lower-case letter or an underscore), and '_' (the identifier consisting of a single underscore). It also uses the notation '{ X }+' to stand for one or more instances of X, plain '{ X }' to stand for zero or more instances of X, and '[X]' to stand for zero or one instances of X.

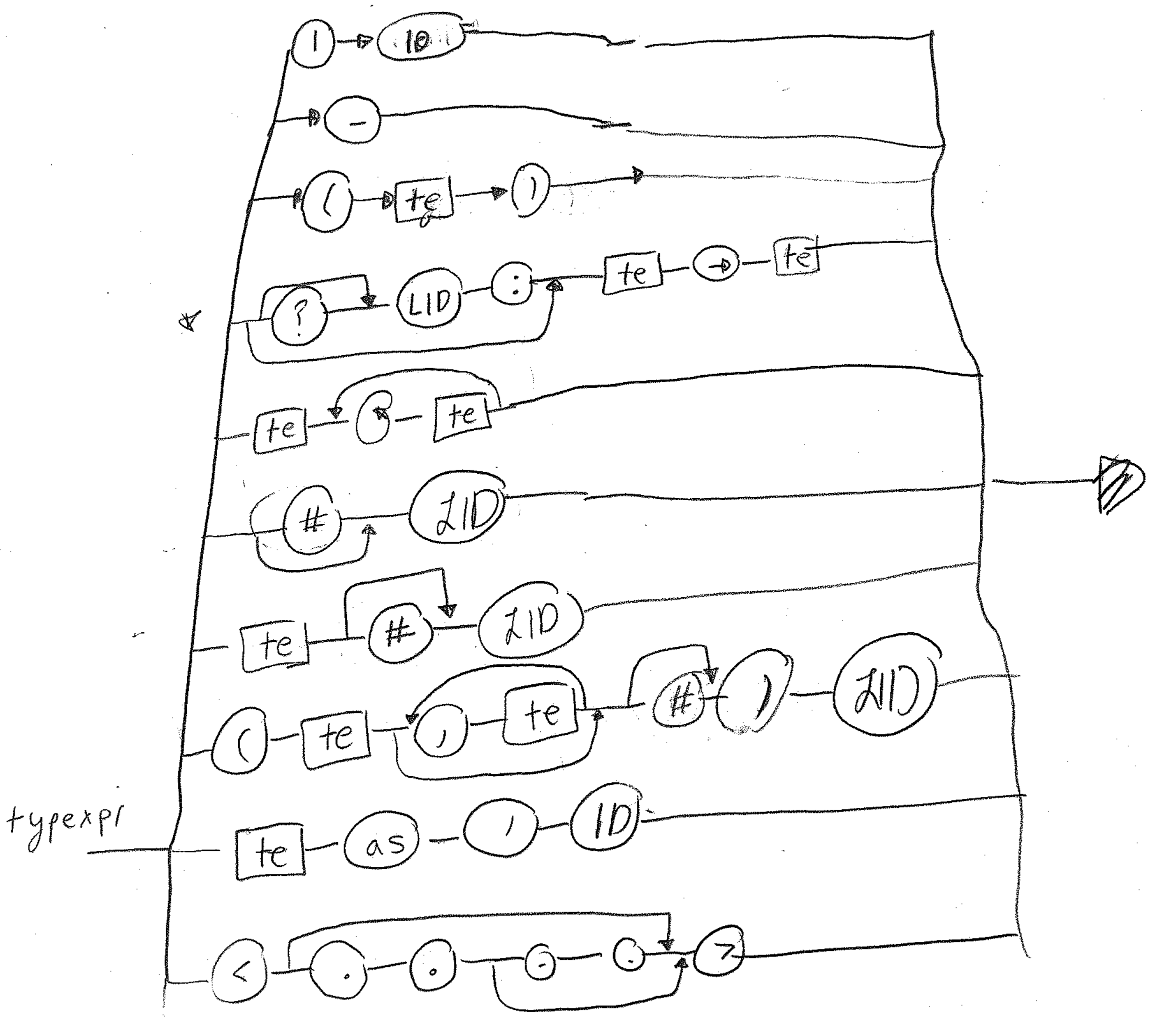
3a (3 minutes). List the nonterminals in this grammar.

typexpr

3b (5 minutes). Convert this grammar to Homework 1 form. Abbreviate 'typeexpr' as 'te', 'ident' as 'ID', and 'lowercase-ident' as 'LID'.

3c (3 minutes). Convert this grammar to Homework 2 form, with the same abbreviations.

3d (10 minutes). Convert this grammar to a syntax diagram with the same abbreviations. Your diagram should be as simple as possible.



4. Java has a class Integer that wraps an int value in an object. It is declared as follows:

```
public final class Integer extends Number implements Comparable<Integer>
```

and with the following public constructors, fields and methods:

```
  Integer(int value); // Create an Integer with the given value.
  static int MAX_VALUE; // maximum int value, 2**31 - 1
  static int MIN_VALUE; // minimum int value, -2**31.
  int intValue(); // Return this Integer's value as an int.
  long longValue(); // Return this Integer's value as a long.
  int hashCode(); // Return a hash code for this Integer.
  static int max(int a, int b); // Return the maximum of a and b.
  static int min(int a, int b); // Return the minimum of a and b.
  ... lots more methods like the above ...
```

The Comparable interface looks like this:

```
  public interface Comparable<T> { int compareTo(T o); }
```

4a (5 minutes). Why does this class have a constructor and instance methods at all? If most of its methods are static and take int arguments, there doesn't seem to be much use for Integer objects.

We need these methods so that we can implement the functionality of the Integer class when we decide to create an instance of it.
It extends the current use of ints to extend functionality.

4b (5 minutes). What is the point of implementing Comparable<Integer> as well as supplying an intValue() method? For example, why can't callers use (a.intValue() < b.intValue()) instead of the more-cumbersome (a.compareTo (b) < 0)?

We use compareTo to compare objects in java and while you can look at the int values we need the compare to in order to properly compare two different Integer Objects.

4c (5 minutes). Why does it make sense for the Integer class to override the hashCode method of its parent class?

Since the hash code is usually the location in memory (or some good random num) we don't want to confuse the Number hash code with our integer hashcode which may result in some unwanted behavior.

5. Consider the code used as a hint in Homework 2. It finds the first match for a pattern, where the match must start at the beginning of the input fragment. Suppose that instead we want the leftmost first match: that is, instead of requiring the match to start at the beginning of the input fragment, it may start later if there is no match at the beginning. If there are several matches, we want to choose the match starting leftmost (earliest) in the fragment; if there are several leftmost matches, we want the first match (in the sense of the hint for Homework 2).

5a (12 minutes). Write a curried function make_leftmost_matcher that accepts a pattern P, a fragment F, and an acceptor A. It acts like make_matcher except (1) its matchers find a leftmost match in F instead of requiring the match to start at F's beginning, and (2) instead of returning (Some U) its successful matchers return (Some (M, U)), where M starts at the position where the match was found, and U starts just after the end of the match that was found; here M and U are both suffixes of F. This way the caller can locate the match.

Your implementation can assume all the functions defined in the hint to Homework 2, as well as the Pervasives and List modules, but it should use no other modules. Also, it should avoid side effects.

let rec make_leftmost_matcher P F A =

5b (3 minutes). What is the type of make_leftmost_matcher?

[illegible] fn : str list * str * 'a → int

5c (5 minutes). Can make_leftmost_matcher go into an infinite loop when given a "bad" pattern and fragment, as your Homework 2 solutions can? If so, give an example; if not, explain why not.

No because this matcher doesn't deal with chaining the left most match. If there is a bad fragment then we can simply break out of the function once it has completed