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: Jason Less Student ID: 404-640-158

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 *

This is because int *x (where x is some ptr) is dereferencing x and accessing the value stored at x's memory address. In this case, a compiler can choose to optimize away accesses to the int *x or not. In the volatile case (i.e. int volatile *x), the compiler has one of these choices removed, namely the choice to optimize it away.

base volatile
- optimize
- don't optimize

derived
- don't optimize

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 declaring an int volatile *x, as stated in the description: "the compiler may optimize away accesses to the ptr x itself." If the rule that accesses to the integer *x can't be optimized is violated, then this could perhaps result in a program that completely optimizes out the variable, resulting in a hard-to-debug error as all references to such an object have been optimized out.

let curry x y = x - y

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 op1 op2 = op2 - op1;;

let (-) = (fun x -> (fun y -> x - y))

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

val reverse_curry2 : ('a -> 'a -> 'a) -> 'a -> 'a -> 'a = <fun>

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

let reverse_curry2 x = (fun op1 -> (fun op2 -> op2 - op1));;

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

- Without passing in any operands, and just passing in the reverse_curry2 func as a param, this would result in partial application & resulting in returning another function

: (('a -> 'a -> 'a) -> 'a -> 'a -> 'a) -> ('a -> 'a -> 'a) -> 'a -> 'a -> 'a = <fun>

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 reverse_curry n x = reverse_curry (n-1) x;;

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
```

{X}+ => 1 or more
{X} => 0 or more
[X] => 0 or 1
EBNF syms
( )
[X] = ε | X
{X} = ε | X {X} | XX {X}

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, [[?] lowercase-ident :], {* typexpr}+, { , typexpr },

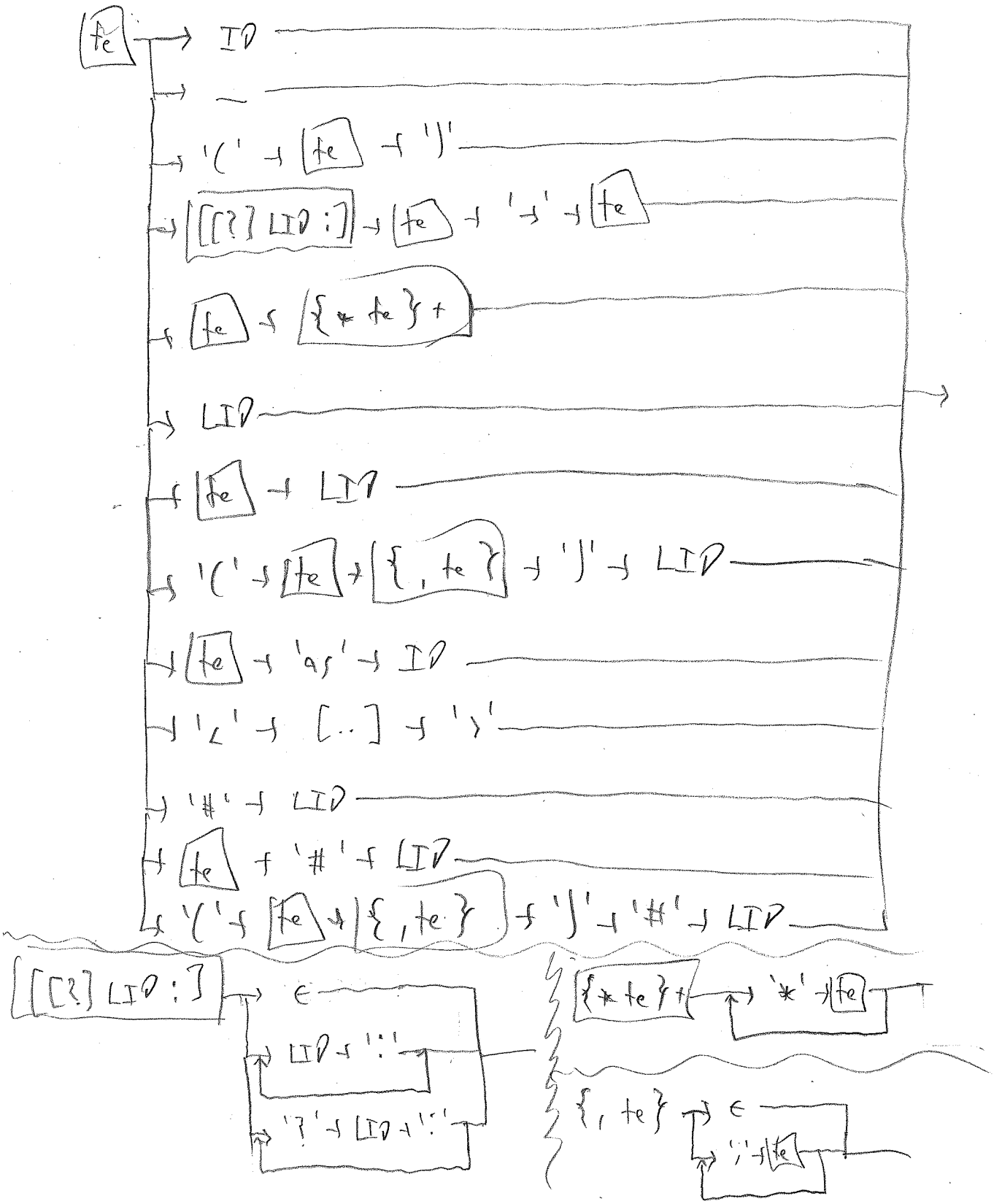
3b (5 minutes). Convert this grammar to Homework 1 form. Abbreviate 'typeexpr' as 'te', 'ident' as 'ID', and 'lowercase-ident' as 'LID'.

```
(te,
   [te, [ID]];
    te, [_];
    te, ['('; te; ')'];
    te, [[[?]LID :]; te; '->'; te];
    te, [te; {* te}+];
    te, [LID];
    te, [te; LID];
    te, ['('; te; {, te}; ')'; LID];
    te, [LID];
    te, ['<'; [..]; '>'];
    te, ['#'; LID];
    te, [te; '#'; LID];
    te, ['('; te; {, te}; ')'; '#'; LID]])
```

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

```
(te, function
  | te ->
     [[ID];
      [_];
      ['('; te; ')'];
      [[[?] LID :]; te; '->'; te];
      [te; {* te}+];
      [LID];
      [te; LID];
      ['('; te; {, te}; ')'; LID];
      [ID];
      ['<'; [..]; '>'];
      ['#'; LID];
      [te; '#'; LID];
      ['('; te; {, te}; ')'; '#'; LID]])
```

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.

- Without the constructor and instance methods, the Integer class would just be an abstract class
- The constructor and instance methods make it so that an instance of the Integer class can be created w/ its own memory and unique values
- The Integer object is useful for polymorphic functions that accept both int and long objects instead of having to reuse code to have separate functions or to have to do type casting

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)?

- This makes the Integer class more flexible and versatile
  - If users want to just know the int or long value, they can call the associated functions
  - Additionally, this makes the code flexible because the programmer writing the function doesn't have to know the types of a and b in the implementation, but can just call the generic compareTo func
- This in turn makes additional funcs later defined to be more generic instead of explicitly knowing if it is an int or long

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

- It makes sense only if the derived class implements the hashCode function differently. If it is the same, it would just be redundant
- As the Integer class has both ints and longs, it most likely will implement the hash function in a version that acts on the parity in the bit representation

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 F. 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 matcher_n n frag accept = match frag with
  | [] -> None
  | h::t -> if h=n then accept t else matcher_n n t accept

let append_matchers m1 m2 frag accept =
  m1 frag (fun f1 -> m2 f1 accept)

let make_appended_matchers make_a_matcher lr =
  let rec maker = function
    | [] -> match_empty
    | h::t -> append_matcher (make_a_matcher h) (maker t)

let rec make_leftmost_matcher = function
  | frag frag -> make_appended_matchers matcher_n frag
```

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

```
: 'a pattern -> 'a list -> ('a list -> 'a*'b option) -> 'a * 'b option = <fun>
```

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.

- It cannot go into an infinite loop in this case
  - This is because unlike in our hw2 solutions, it is trying to match the start of the frag. In this implementation, if it fails for the start of the frag, it will try the next possible match (i.e. makes work on the frag instead of repeatedly trying to match the same thing).
  - Because of this, the frag will eventually be exhausted having worked through the entire fragment