UCLA CS 131 Midterm, Spring 2009
100 minutes total, open book, open notes

1994

Name: _____ Student ID: _____

1	2	3	4	5	6	7	8	total
0	3	1	09	1	5	0	0	19

1 (6 minutes). Java has interfaces instead of multiple inheritance. If interfaces are such a good idea, why couldn't the designers of Java have dispensed with subclasses, and simply used interfaces for everything? In other words, what would be a serious problem for the subset of Java that omits the keyword "extends"? Give example code that would be hard to rewrite into the subset. Don't worry about backward-compatibility concerns; assume that Java had been designed this way from the beginning.

2 (9 minutes). Suppose I design a new language Noparen-Java. Noparen-Java is just like Java, except that there are no parentheses in it (other than in strings and comments). For example:

```
public class Factorial2 {
  public static long factorial long x {
    if x == 1 return 1;
    else return x * factorial x-1;
  }
}
```

One might object that without parenthesis, one cannot write statements like "return (a + b)*c;", but I plan to overcome that objection by having programmers break things into subexpressions, e.g., "int d = a + b; return d*c;". Other than this problem, and the usual backwards compatibility issues, what major problem do you see when implementing Noparen-Java? Give a specific example of the problem.

3. Consider the following code, adapted from the hint in Homework 2:

```
let append_matchers m1 m2 frag accept =
  m1 frag (fun frag1 -> m2 frag1 accept)
```

Suppose we change the calling conventions in the hint, so that a matcher takes its arguments in reverse order. That is, instead of being a curried function that takes a fragment and an acceptor, a matcher is a curried function that takes an acceptor and a fragment. To emphasize this difference, let's call the new-style matcher a "chermat" instead of a "matcher".

3a (10 minutes): Rewrite append_matchers to use the new calling convention rather than the old one. Call the new function "append_chermats". Simplify the resulting code as much as possible.

3b (5 minutes): What is the type of "append_chermats"?

4a (5 minutes). Suppose you have an OCaml function that solves Homework 1, and you give it an ambiguous grammar. What will happen during execution of the function? Briefly illustrate.

4b (5 minutes). Likewise for a matcher generated by Homework 2.

5 (15 minutes). For each of the following OCaml function definitions, give the type of the function, or if there is a type error, say precisely what the type error is. If there is no type error, say what the function does when you call it.

```
let rec a x = x
let rec b x = b
let rec c x = c x
let rec d x = x d
let rec e x y = y x
```

6a (12 minutes). Write an OCaml function "all_permutations X" that returns a list of all permutations of the list X. For example, (all_permutations [3;1;5]) might return [[3;1;5]; [3;5;1]; [1;3;5]; [1;5;3]; [5;3;1]; [5;1;3]]. It's OK to define some auxiliary functions in order to implement all_permutations. It's also OK if your implementation returns the permutations in some other order than this example.

6b (3 minutes). What is the type of all_permutations?

7a (10 minutes). Suppose we define a new language Golorp. Its syntax is the same as Prolog, but its semantics are backwards: within a clause, subgoals are attempted in reverse order, from right to left. Give an interesting example of a Prolog program (including the query) that will behave differently in a Golorp interpreter. Explain the difference in user-visible behavior.

7b (5 minutes). Give an example of a useful Prolog predicate that will have the same behavior in Prolog and Golorp.

8 (15 minutes). Suppose you are assigned the task of solving Homework 2 in Java. Java doesn't have higher-order functions or currying, so how would you reformulate the task of Homework 2 (writing a function that returns a matcher) while retaining the spirit of the problem? Explain how you'd formulate the notions of matcher and acceptor in Java, and illustrate your ideas with some Java code.

1. Java designers could not just dispense with subclasses for interfaces because it would undermine the concept of object oriented programming by taking away the flexibility of subclasses. Interfaces only allow for implementation of their methods and an object would have no flexibility beyond that.

2. Implementing no parentheses in Java would also introduce ambiguities in nesting (i.e. functions taking functions as arguments). There may be default parameters as well. (Also consider nested conditional statements.) This will cause parsing complications and will force Noparen-Java to implement rules to clear up these ambiguities. Also currying may be a solution.

example?

3. a) let append_chem m1 m2 tray accept =
m1 accept (fun a1 → m2 a1 tray)

b) ('a option → 'b) → ('a option → 'b) → 'b → 'a option → ('a option → 'b)

4. a.) If an ambiguous grammar is passed into the function, the random sentence generator will still work. However, because ambiguous grammars generate multiple parse trees for the same expression, the intended semantics of the random sentence may be incorrect. ✓

5 b.) If an ambiguous grammar is passed into the parse_prefix function for a given fragment, the function will still execute correctly. Only, it will generate the parser for the fragment which may be semantically incorrect.

5 let rec a x = x

Type error : a is not called recursively

let rec b x = b

a' → a' (returns a function that takes x)

let rec c x = c x

1 Type error : infinite recursion

let rec d x = x d

a' → a' → (a' → a') (returns x and a function that returns x)

let rec e x y = y x

Type error : e is not called recursively

6. a.) let all_permutations x =

match x with

| [] → [] ✓

| h::t →

let orig_length = List.length x in ✓

let rec get_nth_sublist list n =

2. match list with

| [] → ([], [])

| h::t →

if (List.length list > n) then

let (first, last) = get_nth_sublist t n

(h@first, last)

else

([], list)

in

```
let final_list = [] in
let rec powerset subset =
  match subset with
  | [] -> []
  | h :: t ->
      ( if List.length t = 0 then
          [[h]]
        else let n = 0 in
          let rec p1 build_list element n =
            if n != List.length build_list then
              let (first, rest) = get_nth_list h in
                [first @ h @ last] @ [p1 build_list element n+1]
            else
              build_list )
      in
      p1 build_list h 0
;;
```

(b) 'a list -> 'a list list

7. a.)

b.