UCLA Computer Science 131 final exam - Spring 2020

Name:_____

Student ID:_____

180 minutes total, 1 minute = 1 point.
Open book, open notes, open computer. Answer all questions yourself, without assistance from other students or outsiders.

The exam is not easy, and you are not expected to answer all the questions completely. In your answers, overall approach and intuition will count more than trivial detail. Budget your time while taking the exam. It may help to skip questions that are harder than their point count would suggest.

Print the exam, read the first page, then write your starting time on the first page. Then take at most 180 minutes to answer the questions and write your answers on the exam. (CAE students with x% extra time should add to the 180 minutes accordingly.) When you're done, write your finishing time on the first page, sign the first page, scan the completed exam, and upload your scans to Gradescope as quickly as you can. If you lack a scanner, carefully photograph the sheets of paper with your cell phone and upload the photographs.

Alternatively, you can use a notepad computer to write your answers into the PDF and upload the modified PDF. Or you can read the exam on your laptop's screen, write your answers on blank sheets of paper (preferably 8½″×11″) with one page per question, and upload the scanned sheets of paper; at the end of the exam, you should have scanned and uploaded as many photographs as there are questions (if you do not answer a question, scan a blank sheet of paper as the answer).

Do not give or show the exam or your answers to anybody other than this course's instructor or TA. Save your filled-out exam until at least June 22, in case there were glitches in the upload.

The exam is open book, open notes, and open computer. You can use your laptop to use a search engine for answers, and to run programs designed to help you answer questions. However, do not use your computer or any other method to communicate with other students or outsiders, or anything like that. Communicate only via CCLE and Gradescope to obtain your exam and upload your scanned results, or via Zoom or email with the instructor or TAs.

_____ Time (Los Angeles time) you started the exam

_____ Time that you ended the exam

IMPORTANT Before submitting the exam, certify that you have read and abided by the above rules by signing and dating here:

1a (6 minutes). Give an example OCaml program where the static chain grows in length at the same time that the dynamic chain shrinks in length, and state where this occurs and why. Or, if this can’t be done in OCaml, briefly state why and use Scheme instead of OCaml.

1b (2 minutes). Briefly say why the same thing cannot happen in C.

2. Dart has neither interfaces nor multiple inheritance, so one might wonder how it can rival the flexibility of languages like Java (which has interfaces) and C++ (which has multiple inheritance).

2a (6 minutes). Give an example of how you'd take something written with Java interfaces and express the same sort of code in Dart.

2b (6 minutes). Similarly for C++ multiple inheritance and Dart.

3. Consider the following grammar, written in a form of EBNF:

```
  Program  ::= ChumExpr        Expr'    ::= "I"
                                          | "K" | "k"
  ChumExpr ::= ChumExpr Expr              | "S" | "s"
             | epsilon                    | NonemptyJamExpr
                                          | "`" Expr1 Expr2
  Expr     ::= "i"                       | "*" TauExpr1 TauExpr2
             | Expr'                      | "(" ChumExpr ")"

  TauExpr  ::= "i"             NonemptyJamExpr
             | Expr'                      ::= JamExpr "0"
                                          | JamExpr "1"

                               JamExpr  ::= NonemptyJamExpr
                                          | epsilon
```

In this grammatical notation, an apostrophe (') acts like just another letter, terminals are represented by quoted strings, "epsilon" denotes the empty sequence of symbols, Program is the start symbol, and nonterminals are represented by capitalized identifiers (followed by a digit if there are two or more of the nonterminals in the same rule's right hand side). The "`" token here is a grave accent, not an apostrophe.

3a (6 minutes). Give an example sentence in this language that uses all the available terminal symbols, and give a parse tree for your sentence.

[continued on next page]

[continued from previous page]
3b (6 minutes). If you gave this grammar to a working solution for Homework 2, the resulting parser would likely loop forever. Fix the grammar so that this problem would not happen. Your change to the grammar should be as simple as possible and should not affect the language it accepts.

3c (6 minutes). Use EBNF to make the grammar simpler and easier to understand.

[continued on next page]

[continued from previous page]
3d (6 minutes). Draw a syntax diagram for the grammar, in the style of Webber. Keep the diagram as simple and uncluttered as possible.

3e (10 minutes). Is this grammar ambiguous? If so, give an example of an ambiguous sentence and explain why it's ambiguous; if not, explain why not. Your answer should take into account the fact that two distinct nonterminals (Expr and TauExpr) have identical right-hand sides, and should say whether and how this fact plays a role in your answer.

4a (12 minutes). Write a Prolog predicate adjdups(X,Y) that succeeds if Y is a copy of the list X with any adjacent duplicates removed. You can assume that X is bound to a list of known length. Adjacent elements should be unified as needed to make them duplicates. For example:

```
?- adjdups([10,10,10,10], R).
R = [10]
?- adjdups([1,3,5,5,-1,5,2], R).
R = [1,3,5,-1,5,2]
?- adjdups([X, f(Y), Z], R).
X = f(Y)
Z = f(Y)
R = [f(Y)]
?- adjdups([X, f(Y, a), f(b, Z)], R).
X = f(b,a)
Y = b
Z = a
R = [f(b,a)]
```

4b (6 minutes). What will your implementation do if a caller violates the rule about X being bound to a list of known length? For example, what will adjdups(L, [f(b,a)]) do and why?

5. Suppose you want to write a Scheme function ‘fnx’ that takes as its argument a list that represents a lambda expression with a single argument, and returns an equivalent lambda expression in which the argument is the symbol ‘x’. The lambda expression can contain subexpressions that are numbers, symbols other than ‘x’, function calls, and the special forms ‘lambda’ (with a single argument that is not ‘x’) and ‘quote’; you need not worry about other kinds of subexpressions. For example:

```
(fnx
 '(lambda (a) (list a '(a 27) ((lambda (a) (f a b -1)) (a a)))))
```

should yield the following list:

```
(lambda (x) (list x '(a 27) ((lambda (a) (f a b -1)) (x x))))
```

5a (6 minutes). As the example indicates, fnx should not replace instances of the argument that are quoted, or are an argument of a subsidiary ‘lambda’. Briefly explain why either of these replacements would cause the returned value to not be functionally equivalent to fnx’s argument, giving examples of trouble.

5b (24 minutes). Implement ‘fnx’.

6 (30 minutes). Scheme's continuations give you a way of representing the "future of your program". Python asyncio Futures are an alternative way of representing a programs's future, in that each Future object represents an eventual result of an asynchronous operation.

Compare and contrast these two ways of dealing with a program's planned execution. How are the methods similar, and how do they differ? Cover differences both in terms of implementation, and in terms of how programs can use continuations and/or Futures. For example, can either method be implemented in terms of the other, and if so how?

7 (16 minutes). Call by need (lazy evaluation) is in some sense an optimization of call by name, in that it avoids calling the function representing a function's argument until a call is absolutely needed. Suppose we want to do a similar optimization of call by reference, in that we want to avoid dereferencing the pointer representing a function's argument until the dereferencing is absolutely needed. We don't want this optimization to involve anything as tricky as call by name. But like call by need's optimization of call by name, we won't mind if your optimization has somewhat different semantics so long as most programs won't be affected by the change. What would such an optimization look like, and how would it work? Illustrate with an example.

8. Consider the following semantics for the small subset of OCaml discussed in class:

```
m(E1+E2, C, Vsum) :-
   m(E1, C, V1),
   m(E2, C, V2),
   Vsum is V1+V2.

m(K, C, K) :- integer(K).

m(Var, C, Val) :- member(Var=Val, C).

m(let(X,E1,E2), C, Val) :-
   m(E1, C, V1),
   m(E2, [X=V1 | C], Val).

m(fun(X,E), C, lambda(X,E)).
m(call(E1,E2), C, Result) :-
   m(E1, C, lambda(Xf, Ef)),
   m(E2, C, V2),
   m(Ef, [Xf = V2 | C], Result).
```

These semantics support only integers, but suppose we also wanted to support lists, by specifying the semantics of the following additional OCaml expressions:

‘[]’, represented by [] in Prolog.
‘E1::E2’, represented by '::'(E1, E2) in Prolog.
‘match E with |P1->E1 |P2->E2 | ... |Pn->En’, represented by
   match(E, [(P1->E1), (P2->E2), ..., (Pn->En)]) in Prolog.
   (Note that ‘->’ is a builtin binary operator in Prolog.)

and the following OCaml patterns:

‘[]’, represented by [] in Prolog
‘X::Y’ where X and Y are identifiers, represented by '::'(X,Y) in Prolog, where X and Y are like-named atoms.

For example, the OCaml expression:

```
match l with
  | h::t -> f (h::1::t)
  | [] -> 1::[]
```

is represented by the following Prolog term:

```
match(l, [('::'(h,t) -> call(f, '::'(h, '::'(1, t)))),
          ([] -> '::'(1,[]))])
```

[continued on next page]

[continued from previous page]
8a (16 minutes). Modify the semantics to support these additional OCaml expressions and these OCaml patterns.

8b (4 minutes). In your semantics, do the identifiers declared by ‘match’ patterns have static scoping or dynamic scoping? Briefly explain.

[page 12]
9 (12 minutes). GNU C has the notion of function attributes. For example:

```
  _Noreturn void exit (int status);
    // This function exits the program with the given status.
    // The ‘_Noreturn’ attribute means ‘exit’ never returns.
    // This lets the compiler avoid saving the return address
    // when calling ‘exit’ (calling code can just jump to ‘exit’).
    // Unoptimized calls to ‘exit’ still work (if a bit more slowly
    // than optimized calls).

  double muladd (double a, double b, double c) __attribute__ ((const));
    // This function returns a*b + c; the __attribute__ ((const))
    // means the function neither depends on nor affects observable state
    // so it always returns a value that depends only on its arguments.
    // This lets the compiler avoid calling ‘muladd’ multiple times
    // with the same arguments, as calling code can just reuse the
    // earlier result in that case.
```

C also has the notion of function pointers, such as this:

```
    void (*p) (int) = exit;
```

which declares ‘p’ to be a pointer to ‘exit’.

Suppose function attributes worked within pointer types, so that you could declare “pointer to _Noreturn function” and “pointer to const function”. What should the subtype relationship be? That is, should “pointer to _Noreturn function” be a subtype of ordinary pointer to function, or should it be the reverse, or should neither be a subtype of the other? And similarly for “pointer to const function” vs ordinary pointer to function. Briefly justify your answers.