UCLA Computer Science 131 (Winter 2020)
Final exam

Name:_____

Student ID:_____

180 points total. Open book, open notes, open computer. Answer all questions yourself, without assistance from other students or outsiders, and do not give any information about the contents of or possible answers to this exam to anybody other than the instructor or TAs.

Print this exam, write your answers on it, scan the completed exam, and upload your scans to CCLE Gradescope by 11:30 Wednesday (Los Angeles time). If you do not have easy access to a scanner, carefully photograph the sheets of paper with your cell phone and upload the photographs. Save your filled-out exam and do not give or show it to anybody other than an instructor or TA; we will send you instructions later about what to do with your filled-out exam.

If you lack access to a printer, 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. Please answer every question on a new sheet 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.

As previously announced, the exam is open book and open notes, but due to circumstances we are also making it 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.

This exam is designed to take three hours. Do not waste time trying to polish or embellish your answers. Excessively polished or long answers will be penalized.

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

Signature:_____ Date:_____

1 (36 points). Consider the following interpreters:

A. An interpreter for OCaml written in Prolog.
B. An interpreter for Prolog written in OCaml.
C. An interpreter for Python written in Java.
D. An interpreter for Java written in Python.

Compare and contrast the difficulty of implementing the four interpreters. What features will cause the most problem in implementing? Consider both correctness and efficiency issues. State any assumptions you're making.

When answering, assume just the subsets of the languages that were covered in class and homeworks; for example, do not consider Java features that were not covered, either when thinking about the Python interpreter or about the Java interpreter.

Also, assume expertise in all four languages, and that all four interpreters are written from scratch.

2. A nonterminal N is “nearly-terminal” if all rules with N as the left hand side contain only terminal symbols in the right hand side. For example, if no rules have N as the left hand side, or only one rule has N as the left hand side and that rule has an empty right hand side, then N is nearly-terminal.

2a (9 points). Convert the awkish_grammar of Homework 2 to an equivalent nearly-terminal grammar. Here is a copy of the grammar; convert it in place by modifying it:

```
  let awkish_grammar =
    (Expr,
     function
       | Expr ->
           [[N Term; N Binop; N Expr];
            [N Term]]
       | Term ->
           [[N Num];
            [N Lvalue];
            [N Incrop; N Lvalue];
            [N Lvalue; N Incrop];
            [T"("; N Expr; T")"]]
       | Lvalue ->
           [[T"$"; N Expr]]
       | Incrop ->
           [[T"++"];
            [T"--"]]
       | Binop ->
           [[T"+"];
            [T"-"]]
       | Num ->
           [[T"0"]; [T"1"]; [T"2"]; [T"3"]; [T"4"];
            [T"5"]; [T"6"]; [T"7"]; [T"8"]; [T"9"]])
```

2b (27 points). Suppose we want an OCaml function de_nearly_terminal that accepts a grammar in the style of Homework 2, and that returns an equivalent grammar that contains no nearly-terminal nonterminals. (Two grammars are “equivalent” if they correspond to the same language, i.e., the same sets of sequences of terminals.)

Describe the practical and/or theoretical problems that you’d run into when writing de_nearly_terminal. (Do not write an actual implementation of de_nearly_terminal.)

3 (36 minutes). Asynchronous I/O can be done in any imperative language. Suppose your application is I/O-bound and does a lot of asynchronous I/O, and that your programmers are equally comfortable and competent in C++, Java, and Python. List the important pros and cons of each of the three languages for the application. Assume all three languages have asyncio libraries of roughly equal capabilities. Assume the application is a server intended to be run in an edge device as part of a large Internet-of-things application.

4 (36 minutes). Would it be reasonable to replace one of the main programming languages of this course (OCaml, Java, Prolog, Python, Scheme) with Dart? If so, which language would you replace and why that one and not the others? If not, redo this question with some language other than Dart. Don’t worry that the textbook covers ML, Java and Prolog; assume that we can choose a new textbook that covers whatever languages we like. Assume that the goal of the course is to teach programming language fundamentals, not the trendy language of the month.

5 (36 minutes). In Scheme, a continuation is a compact data structure representing the future execution of your program. Continuations have certain uses as mentioned in class and in Dybvig. But suppose we want to look into the past instead of the future. That is, suppose we want a primitive that acts like call/cc but creates a data structure (let's call it a "protinuation") that represents the *past* execution of a program, rather than the *future* execution of the program as a continuation does. The intended application area is computer forensics, where analysts may want to write code that reviews program history to see what went wrong (e.g., after a criminal breaks into the application).

Would it make sense to add protinuation primitives to Scheme? If so, sketch out the Scheme API for them, discuss practicality and give an example of how the API might be used. If not, suggest an alternative primitive that would help attack the computer-forensics problem, and do a similar analysis for your API instead.