COM SCI 131 Final exam

Junhong Wang

TOTAL POINTS

116 / 180

QUESTION 1

1 Interpreters **24 / 36**

QUESTION 2

Grammar 36 pts

2.1 Convert to nearly-terminal **9 / 9**

2.2 Ocaml de_nearly_terminal **27 / 27**

QUESTION 3

3 Asnc I/O **14 / 36**

QUESTION 4

4 Replace programming language **24 / 36**

QUESTION 5

5 Scheme Continuation **18 / 36**

UCLA Computer Science 131 (Winter 2020)
Final exam

Name: Junhong Wang

Student ID: 504941113

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~~ Thursday (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: Junhong Wang Date: 03/18/2020

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.

Backtracking issue
assignment
lack of unification (variables as result)
passing function / importing C library /
Assume CPython
library issue,
Types Checking
list of same type
extra code to
pass type checking

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.

A: Assignment in OCaml causes a trouble in Prolog because in Prolog, once a variable is unified to a value, it cannot be changed. Backtracking in Prolog also cause a trouble. Consider pattern matching in OCaml and unification in Prolog. In OCaml, once a pattern is matched, we go inside and return whatever it returns (even it it returns false!) However, in Prolog, if an unification results in "No", it will backtrack and try different unification, which might end up return "Yes". Unnecessary backtracking also slows down the program.

B: In Prolog, it can return variables as a result of unification. For example, "length(X, 2)" in Proleg returns X = [_123, _124] where the underscored variables indicate "anything". We can't return variables (undermined values like _123, _124) in Ocaml.

C: In Python, function is first order citizen, meaning we can pass function as argument, which is not allowed in Java. Simulating currying in Java incurs extra overhead that slows down the program. Also some Python packages are written in C/C++, which can't be executed with JVM.

D: Assume we are using Cython (C implementation of Python) because interpreting Java with Jython (Java implementation of Python) causes a cyclic dependency. Then we will have a trouble when importing library written in Java because we are using Cython.

1 Interpreters **24 / 36**

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:

"grammar that contains no nearly-terminal nonterminals"

```
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"]])
```

; [N Incrop]

; [N Binop]

; [N Num]

2.1 Convert to nearly-terminal 9 / 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.)

Assumption: All we have access to is the argument of de_nearly_terminal (i.e. we don't know the list of possible nonterminals)

Here's the algorithm of de_nearly_terminal:

① For every nonterminal N, we check all its right hand side
If all tokens on the right hand sides are terminals, we
add a rule that points to itself just like in 2a. For example,
Binop → [[T"+"]; [T"-"]] becomes [Binop, [T"+"]; Binop, [T"-"]; Binop, [N Binop]]

② Use convert_grammar from HW2 to convert it back to the grammar style of HW2.
(i.e. Binop → [[T"+"]; [T"-"]; [N Binop]])

There is a problem when implementing this algorithm. Since we assume we don't have access to the list of nonterminals, we would need to start from the start symbol of the grammar, to find all the nonterminals. However, the problem is that we can only find reachable nonterminals. If there exists a nonterminal that is never used on the right hand side, and happened to be a nearly terminal, then this algorith does not work.

2.2 Ocaml de_nearly_terminal **27 / 27**

(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.

C++ : The benefit of using C++ is its speed. The comiler works in such a way that optimizes the code as much as possible. It also supports multithreading in case we also want to perform some CPU-bound operations. However, the problem with C++ is that, it's not portable. For example, order of execution may change depending on which compiler the server computer uses, which results in inconsistant behavior.

Java : Java is good for its portability. Java code is compiled into bytecode and executed by JVM (Java Virtual Machine). Any computer with JVM can run the program and we can expect it to all have the same behavior (e.g. order of execution is the same) In other words, the program won't be affected by which machine you are running. Just like C++, it also supports multithreading, so in case we want to do some CPU-bound tasks, we can do it without blocking the I/O. However, it is not as fast as C++ because we are not directly executing machine code. But JIT comilation allows us to execute frequently used bytecode into machine code to speed up the program, so the problem is not huge.

python : Python is good for its flexibility. Any feature can be quickly implemented thanks to all the Python packages available out there (Python community is huge). However, the problem with Python is that its relatively slower than these compiled languages like C++ and Java (Python is interpreted). Another con of Python is that multithreading is not well supported due to GIL (Global Interpreter Lock). Every time we want to update the in case we want to do CPU-bounded tasks reference count of an object, we need to put a lock on it to avoid race condition. But, we can work around this problem by using Python package like numpy because it's written in C, which supports multithreading.

3 Asnc I/O 14 / 36

[page

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.

Since the goal of the course is to learn programming language fundamentals, we must learn programming languages of different paradiams. We should also learn programming language features such as garbage collection algorithms, compilers, interpreters and types. Thus, these topics should be covered:

- imperative programming
- functional programming
- object-oriented programming
- logic programming
- garbage collection (mark-and-sweep)
- reference counting
- generational garbage collection
- statically typed
- dynamically typed
- type inference

Ocaml was covered to learn functional programmirs and type inference.
Java was covered to learn Object-Oriented Programming and garbage collection / generational garbage collection, statical typing, and JIT compilation.

Prolog was covered for logic programming.
Python was covered to learn reference countins, dynamically typing, and interpreter.
Scheme was covered to learn functional programming and cattinuation.

Dart, on the other hands, can cover topics such as object-oriented programming, JIT compilation, type inference, and generational garbage collection.

Thus, it looks like it's reasonable to replace Java with Dart because they can cover the same topics (since they cover same topics, we don't need to replace it also).

4 Replace programming language **24 / 36**

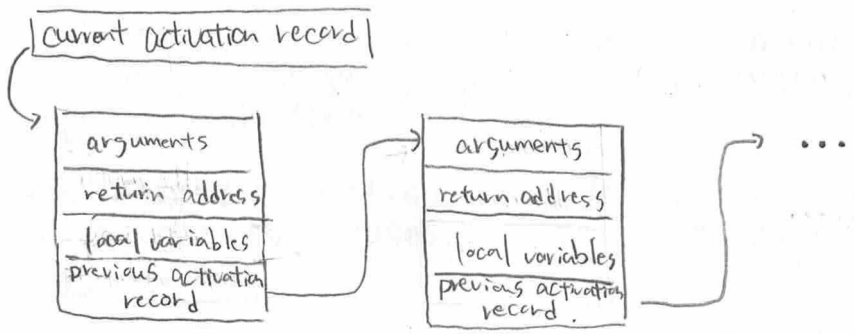
(36 minutes). In Scheme, a continuation is a compact data structure representing the future execution of your program. Continuations have certain uses as mentioned in class and in Dybvig. But suppose we want to look into the past instead of the future. That is, suppose we want a primitive that acts like 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.

From my interpretation of the problem, it sounds like we want a function that gives us history of the program execution, which is kind of like a call stack. Here's the Scheme API for this primitive (let's call it call/cp):

```
(call/cp) ; for example, consider (f (g (h (display (call/cp))))).
          ; Then this should output h ← g ← h.
```

In the API specification above, call/cp only shows us the functions that were executed, but we can implement call/cp such that it also returns more useful information such as variables, arguments, return address etc. call/cp can be implemented using the activation records (Here, we are assuming that we have access to current activation record in Scheme).



In terms of practicality, it may not be super useful because it can only show *past* that leads to *now*. Activation records that are already popped off the stack will not be shown with the implementation of call/cp discussed above.

5 Scheme Continuation 18 / 36