COM SCI 131 Final exam

Sriram Balachandran

TOTAL POINTS

131 / 180

QUESTION 1

1 Interpreters **31 / 36**

QUESTION 2

Grammar 36 pts

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

2.2 Ocaml de_nearly_terminal **18 / 27**

QUESTION 3

3 Asnc I/O **22 / 36**

QUESTION 4

4 Replace programming language **33 / 36**

QUESTION 5

5 Scheme Continuation **18 / 36**

1 (36 points). Consider the following interpreters:

A. An interpreter for OCaml written in Prolog. 4 Hardest
B. An interpreter for Prolog written in OCaml. 3
C. An interpreter for Python written in Java. 2
D. An interpreter for Java written in Python. 1 Easiest

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.

D. I believe a Java interpreter written would be the easiest to implement. Python
employs similar ~~O~~object-oriented principles, albeit less purely so. For example,
I know of no equivalent in Python for a Java interface. Another complication
is that Python cannot offer Java's strict type-checking. However this should
be a huge issue since Java code is being converted to Python and not
vice versa. A definite correctness issue that will arise is handling Java's
multithreaded features. At best, Python can support ~~T~~ concurrency through
Asynchronous programming but this is not enough for the singlethreaded Python
to mimic Multithreaded Java.

C. Java is capable of offering all the functionality of Python with 2 exceptions. The
first is Python's dynamic type checking - although this could potentially be
mimicked through the use of Generics. ~~Java can also cannot offer Python's~~
Some type-related issues could be resolved by ~~the~~ using more of Python 3's
type-safety mechanisms. Java ~~can~~ also cannot offer the line-by-line interpretation
that Python does, ~~as~~ or it would do so much slower since Java needs to
be compiled to Byte-code first.

B. I think off the bat, unlike the previous examples, OCaml could definitely be used to write a Prolog interpreter.
There is no key loss of functionality switching from the logical paradigm
to functional pradigm (although the functional code would be bulkier).
The key issue that I believe would make this interpreter trickier than
the previous is Prolog's call by unification feature. This means one would
algorithmically need to figure out a way to separate a predicate into 2
functions: ~~to~~ 1 to solve the problem, and 2 to verify the solution. Another issue
is that OCaml can't offer the multiple-~~de~~backtracked predicate definition that Prolog offers
easily - however one could perhaps mimic this by passing in subsequent definitions
as acceptors to function definitions.

A. ~~Prolog cannot~~ Converting from functional to logical is definitely the most difficult. Functional languages evaluate expressions and return values; predicates simply verify some statement. Prolog cannot offer the same flexibility of currying that OCaml offers, making it difficult to write an interpreter for ~~Prolo~~ OCaml in Prolog.

1 Interpreters 31 / 36

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; T"+"; N Expr]; [N Term; T"-"; N Expr];
          [N Term]]
     | Term ->
         [~~[N Num];~~ [T"0"]; [T"1"]; [T"2"]; [T"3"]; [T"4"]; [T"5"]; [T"6"]; [T"7"]; [T"8"]; [T"9"]
          [N Lvalue];
          ~~[N Incrop; N Lvalue];~~ [T"++"; N Lvalue]; [T"--"; N Lvalue];
          ~~[N Lvalue; N Incrop];~~ [N Lvalue; T"++"]; [N Lvalue; T"--"];
          [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"]]~~)

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.)

Practically, de-nearly-terminal should be relatively straightforward to implement (one could do so in a manner similar to the last problem in hw1). However there are potential tricky grammars that would render this function quite impractical. Take this example grammar:

```
let example = (
    Term,
    function
      | TermTop -> [[N Term; T "+"]]
      | Term -> [[T "?"]; [N Num; T "."]]
      | Num -> [[T "6"]; [T "9"]; [T "4"]; [T "2"]; [T "0"]]
)
```

If one ~~made~~ tried to de_nearly-terminal the above grammar, it would first remove the Num Terminal and replace the corresponding rules in Term. However in doing so, Term has now become a nearly-terminal. This means the process has to ~~be~~ repeated until we can find no more ~~nonterminals~~ nearly-nonterminals. This backtracking logic/~~~~looping nature should be accounted for. Although not an issue per se, the above situation could also result in a case where we explicitly defined every possibility for a given grammar, which would not be very practical or readable.

A more prominent problem lies theoretically with the form of the grammar passed in. There is no way to remove unreachable nearly-~~non~~terminals nonterminals. Since the form of the grammar gives us a start symbol and a generator function, we can only explore the list of rules reachable by the start symbol. ~~There~~ One solution would be to return a new generator function containing only reachable nonterminals instead of modifying the generator in place. However, this would result in the removal of all unreachable nonterminals, not just nearly-terminal ones. Another approach is to simply ensure the grammar is formed without ~~unreachable~~ unreachable rules, or pass in a grammar in the form of the hw1 grammars.

2.2 Ocaml de_nearly_terminal 18 / 27

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.

in all analyses, I assume that the edge device has a multiple core ~~s~~ CPU.

A C++ server can offer highly performant server all in terms of speed since it can be compiled into pure machine code. However, since it is a statically compiled language, small changes to the source code can result in unnecessary waiting for recompilation. Depending on the type of operations being performed, C++ can also offer greater control over memory - on the flip side this means explicitly user-allocated memory must be explicitly deallocated by the programmer which can be tedious and error-inducing. C++ also offers some multithreading support, which can speed up more computationally heavy operations. Another disadvantage of C++ is that runtime error detection must be handled and accounted for by the programmer, which can be tedious also.

A Java server can offer the speed benefits associated with compilation with less overhead latency from compilation, this is accomplished by compiling Java programs to Java Byte code. This allows for quick source code updates more flexibly than C++, but not quite as well as Python. A con of this is that in order to run Java Byte Code, the edge device will need to have the Java Virtual Machine installed as well, which can be space-costly. Due to the very Object Oriented nature of Java, the Java program overall can be quite large spatially compared to C++/Python equivalents. Java also offers far more intuitive and safe multithreading mechanisms, compared to C++/Python, which allows it to multithread tasks in a more performant manner since multithreading was built into Java. One could also multithread the asynchronous server task itself, allowing one to process multiple requests concurrently. However, Java offers less explicit memory manipulation which could be detrimental depending on the task. Java will also handle runtime errors implicitly for us, allowing the programmer to worry less about small errors.

A Python server would be by far the easiest to code, and simple to understand. It's also an interpreted language, which means changes to the source code can be made extremely quickly with almost no latency. However, due to the interpreted nature of Python, it is slower than Java/C++, but can offer a much more lightweight program overall. There are also spatial costs associated with having to install the Python interpreter on the edge device. Python is single-threaded and cannot take advantage of the multiple cores. Python's dynamic type checking can result in a much more tedious development process, as many bugs could slip by undetected by the programmer until the error happens.

3 Asnc I/O 22 / 36

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.

I think it would be reasonable to replace Python with Dart. Dart shouldn't replace OCaml because OCaml and the second OCaml assignment in this class provides a valuable introduction and insight into the functional programming paradigm. I believe Assignment 2 ~~and OCaml~~ was crucial to show us as young programmers/engineers how different languages/paradigms can make a world of difference in ~~solving~~ making a problem easier to solve. A similar argument can be made for Prolog - Dart cannot replace the understanding gained from practical experience with a logical programming language. I think ~~Prolog~~ logical programming in particular represents the greatest departure from ~~on~~ the styles of thinking we as UCLA CS students ~~go~~ have grown accustomed to in our coursework, and Dart cannot mimic this experience. Java I believe represents one of the purest forms of an Object Oriented Programming language. Although Dart offers more OO structure than Python does, I don't believe it is enough to warrant replacing Java. Additionally Java is the only language covered in depth in this course that was designed with multithreading intended as an "out of the box" feature. As Dart is single-threaded, I don't believe it can teach us those same lessons learned from Java's multithreading. Additionally, Dart is not capable of replicating the same race-condition prevention mechanisms of Java, and it definitely cannot do so with the same ease-of-use that Java provides such as a simple "synchronized" keyword. ~~As~~ Scheme is tougher to justify, since OCaml covers the functional paradigm already. However, Scheme's exposure of continuations ~~is~~ is a powerful enough feature to warrant it staying in the curriculum. Although Dart similarly employs continuations in the form of callback functions, Scheme exposes continuations much more explicitly, with greater flexibility, forcing us as programmers to truly attempt to understand it.

Dart can replace Python in this course since it is designed with similar use-cases in the context of this course. The Python project can easily be switched to Dart with relatively minimal difficulty changes. Dart has better type-checking which contributes to making better programmers out of us, but also offers the dynamic type checking of Python with the "dynamic" type. Dart also has built in ~~&~~ Asynchronous support / Event Loop implementations, which would allow us to further explore Asynchronous programming in lecture, since it has become increasingly popular in industry. Python & Dart's garbage collectors are also similar, allowing us to use Dart's [illegible] Dart can also be interpreted like Python - Dart offers a [illegible]

4 Replace programming language 33 / 36

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.

Although it may lack in general uses outside of the forensics case I believe it would make sense. It would most likely be called ~~just~~ ~~after~~ with the execution of a function we are concerned with. It would then take a snapshot of the current stack frame just before returning from the function - something like this

(set! x (call/cp (myfunc())))

This call with current protination function would return the value of myfunc, but also store the stack frame just prior to returning. Then, to invoke a previous running of the function we would call the call with previous protination ~~calls~~. The Scheme interpreted would store protination in the form of a linkedlist, and key these linkedlist based on function headers (so each function has a "history"). If you wrap the call to call/cp inside a call/cc call, we can effectively jump back and then rewind the program using the values ~~returned~~ from the stack frames of the previous protinuation. By using continuations and this protination API, the forensics use case could be achieved, in a way similar to the ~~in addition to~~ trace primitive.

*call/pp (func-name)

5 Scheme Continuation 18 / 36