Advanced programming - Assignment 2

Nicolas Dyhrman (LZG210)

September 2022

I do not know if there is too much missing for the cases to not be valid for re-submission, but would be nice to know, I know there are a lot missing and am waiting to see if it would be possible with the time estimates you would think for the missing parts.

1 Completeness

Not all functionality has been implemented, and some have been implemented incorrectly. *Execute* has not been implemented at all, and have not looked into how to implement it in a need of more time to do so.
Exec has not been tested very well. Although *print* does not save the results correctly to the monad output but is an error inside the monad that I was not capable of finding a solution for, and if a re-submission was to be assigned, a tip maybe as I spent a lot of time on it.
List comprehensions were a mistake by me, not understanding exactly what was meant by the functionality described. It therefore won't behave exactly like a list comprehension, and instead, only work if there is only one *CCFor*. A solution to the problem could be to first evaluate all of the lists (with eval (List xs)) and then use each to evaluate e_0 and call it again with a pattern match until we have resolved all of them for all the variables.
Without writing too much but the output problem I think is the same as I had in the warmup tasks.

2 Correctness

As said the print works incorrectly without me knowing why as well, and the output, therefore, becomes incorrect. List comprehension again cannot work for more *CCFor* than one and will NOT terminate before it has found a Boolean guard that fails looking from left to right, instead of checking *CCIf*s first. *Exec* seems to be able to evaluate anything where there is not an input environment. From evaluated environments, it seems to fail.

The return of the monad works by creating the base case of the monad, where we just want to insert '*a*' value into the value part of the monad (*a*, *output*) where output, in this case, will be *mempty*.
The binding (>>=) of the monad, first calls the already passed monad, with runComp to gain the values of the monad. We then save those and use those values to create the function application with the new value a and the environment. If the first call should instead fail with a Left we propagate the error, without calling the new application of a.

Since I developed List comprehensions wrongly I can not tell more than what I did under the Completeness section. But right now it works with the use of going through each step of the list given and then matching if it is, a *CCIf* or

CCFor. And then evaluate the statement if it is a *CCIf* evaluate e and check if it is True otherwise send back an empty Listval. And then recursively call for the next element in the list. If *CCFor* is met evaluate the list of expressions, and then evaluate e_0 with each value that it would evaluate with the binding. While calling the next element after recursively. Look further up for my idea of implementing the correct *List Comprehension*.

3 Efficiency

I have no reason to think that my code is not working have a good time and space calculation. I do not perform a lot of unwanted computations. I do sometimes, create some not needed lists, where I get back [Comp Value] and need to turn it into Comp Value and can therefore have some arrays that might be not needed, but nothing I would think would take a big toll on the program performance. The recursive calls also only are done on important and necessary calls, and therefore I think would have a fine efficiency.

4 Robustness

The robustness of the program is in a great state for all the fully developed parts. The error code given (either with string in operate or the run time error with the message) might not give a perfect understanding of what went wrong with the program, and how to fix it, but an abort should be thrown in almost all cases. In the less developed parts, I still cannot see where there should be a problem but I do not know how or where you would be throwing an error in them that the program would not handle. Most Errors are thrown using the abort function where ever possible.

5 Maintainability

5.1 Code

I have put more abstract auxiliary functions in to help with keeping the code cleaner, although it is not as bad as expected, in this assignment. *operate* uses a lot of the same types of code with *case x of* that we could have done more about. There is almost not used any form of Comp or runComp in the parts where it is not permitted. Only in Run as it gave me problems even worse than the ones I had since the monad does not save the output in itself.

5.2 Test suite

The test suite is very minimal only testing very softly on anything to do with eval, to make sure the functions work at least almost properly, while done more in hand when I developed each of the cases. But is certainly not enough to

create a good test suite for the entire interpreter. But does still go through most needed auxiliary functions, with just using eval.