Question 2: Interpreting ANTARESIA

# Implementation

Because I had issues with understanding how to work with monads I have chosen not to follow very much the skeleton imposed by the handed out code. I have chosen to use only one Monad and I have implemented the interpreter very much like I did with the MSM at the assignment.

I have created the Error to contain data *ErrorType* and the state of the program at the moment when error occurred. The *ErrorType* contains the different error that may occur during the execution of program. I have also added the state of the program at *Error* in case I wanted to also display the state of the program at the moment when the error occurred by I did not use it afterwards.

The state of the program is called *AntaState* which contains the program that is to be executed as list of *Decl*, the map *Env* which holds a mapping between the *Name* of the variables and the *Value* which was intended to hold all variables that appeared during the execution of the program and a list with all Results that happened during the evaluation of expressions along the execution of program. The list *res* was intended to be able to return the final results of the program but afterwards I have notices that however, my results are stored in *Env* so I have mostly used res as a list where I kept the history with results for all expressions interpreted and executed during the program and it has proven useful while I did the debugging of my interpreter since a *Name* can change its value in case an expression of type Name is interpreted.

I have created the monadic *newType* *AntaM* which takes the *AntaState* and returns either an error or the final state of the program. I have also created utility functions for setting and getting the state of *AntaM* during the execution of program to obtain or manipulate the state when interpreting the expression. There it is also a function *updateEnv* that I created with the purpose for changing only the *Env* of the *AntaState* but I no longer used it.

# Start execution of program

For starting the execution of a program I have created the function *runProg* as required which returns either error or the *Result* of the execution. This function in fact calls the function program which takes the program to execute and returns either error or final *AntaState*. The function *runProg* takes the result of execution and extracts the Result which is actually the *Env* of the *AntaState* or returns the error in case it happened.

I have made these last 2 functions to return *String* instead of *Error* because I have a function *showError* that returns a string with an error message corresponding to its *ErrorType*.

The function *program* is actually calling the function *interp* that starts the actual interpretation of program. The function *getDecl* is used to get the next *Decl* in program which is actually a list of *Decl*. The function *interp* takes the *Decl* returned by *getDecl* and calls the function *evalExpr* that is looking at the type of expression in *Decl* and is using the corresponding function for interpreting the expression. Each expression is removing the first *Decl*(the one it executed) for the *Program* of *AntaState* and in case the empty then it returns a boolean *False*, otherwise it returns *True*. The *interp* function checks the boolean result of each expression and it repeats the process if there are *Decl* left in the *Program*.

# Interpretation of expressions

I will describe shortly the implementation of the expressions that I have created and how I managed to solve the challenges that I met:

* Each expression is taking the parameters, is calculating the results and is adding the mapping between the *Name* and the type of *Value* that is resulting to both the map *Env* and the list Result of the *AntaState*. If the expression is for inserting constants it just adds the value without calculating anything.
* For the expression of type *Name Name* I have chosen to change the value of variable with name at the right of ‘= ‘sign and is assigned to the other variable. In case the name at the right of ‘=’ is not found in the map of Env then it is returned an error of type *VariableNotFound*. Since it was not described in the exam handout how to interpret these expressions, I hope this is what they were intended to do.
* For the mathematical expressions *Plus*, *Minus*, *Mult*, *Div* and *Modulus* the expressions are, at first checked if both parameters and then the result of the mathematical operation is returned. Otherwise, an error of type *InvalidArguments* is returned. For calculating the integer result of *Div* I have used *`div`* and for *Modulus* I have used `mod` operator and for both expression I return *DivideByZero* error.
* I have implemented the *range* function as described and it works properly. The result of expression is *List* of *IntConst* and I have checked the type *Args* to see how many values it has. I also checked if all parameters are integers and returned error of type *InvalidArguments.* Here it is where I noticed that my parser does not parse the negative integers and it was too late to change it so I can also test the program also works for negative step size.
* For the equal expression I have checked type of *Expr* and returned error of type *NotSameArguments* in case of different types. Because I did not have much time left and was not sure if it is needed I have not implemented that the expression to also be evaluated if arguments of *Equal* are *Names* in order take the values stored previously in map *Env* and compare them.
* Unfortunately I did have enough time to also implement the interpretation of list comprehensions and also it seems to be harder to implement. If thought and tried to make a call to a recursive function to look through multiple loops that could appear at the second argument of expressions but could not make it work in time. In case this expression appears in the program I have chosen to return an error of type *NotImplementedYet*.

# Tests

I have also done a few tests to prove that my interpreter works for what I have managed to implement. I have done them in same way as for the parser. I check if the result of running the program is *Error* and then mark test as failed. If the program returns *Result* then the test is validated. Unfortunately I did not manage to also make the tests to compare the result of program to the expected results. For the tests that I have expected to be fail (inserted wrong arguments to see if I get the expected *ErrorType*) I have negated the boolean returned by *checkResult* function so these tests to also be validated.

I have mostly done tests to check the basic functionality of all expressions for which I implemented the interpretation and also have some tests where I checked if Errors are returned for wrong arguments.

# Conclusion

I have chosen to make a simpler implementation of the Interpreter since I had no idea how to follow the skeleton proposed with 2 connected monads. And I also thought it is suited to the syntax of Antaresia that to have only one monad since it only contains basic expressions and not classes with objects and methods as it was for the Fast program that was at last attempt for the exam. Things have proven to work with few things that could be improved like make Equal to also work with Name.

The program returns some warnings at compilation but only because I did no longer pass the *AntaState* to the errors since I considered it is no longer needed. I could remove that field from *ErrorType* but I have chosen to let it like that.