Question 1: Parsing ANTARESIA

# Implementation

For the implementation of parser I used the monadic parser library *SimpleParse.hs* because I understood very well how it works in order to be able to solve the requirements of the question. While writing my solution I studied a lot of solutions offered for the previous exams by other students in order to manage to solve all the issues I had while writing the parser.

# Errors

I have changed the Error type that was offered in the handout and I have created a data type called ErrorType which contains two kinds of errors:

* Unspec for the situation when the code to be parsed has errors and the program cannot manage to parse it
* Ambiguous for the situation when the Parser reaches points where there are several way to parse a part of the text

# Functions to parse strings and parse files

I have 3 functions at the end of file that are involved in the process to start the parsing and offer the results and which are used to start the parsing at the command line:

* *parseString* which is the function that is called for parsing a string of characters. It returns either an error or the Program
* *checkError* function that is called by *parseString* to check if there is any kind of error during the parsing and also sorts the errors in the two types of errors that I have defined
* *parseFile* is the function which, as required, is used for parsing the program from a file. It calls the function *parseString* with the input the text read by IO from the file.

# How I managed to parse the grammar tree

I will describe shortly how I managed to deal with main challenges in parsing the code in respect for the grammar tree and the rules that are imposed:

* For dealing with whitespaces I have used the function *symbol* in order to remove all spaces before and after symbols like “,” ”\*” “{” and keywords in different expressions and other structures of the language. I have also removed whitespaces before and after *Name* and *Integer* by using function *token*
* In order to deal with the precedence in the type of operations that have to be parsed as *Expr* I have decided to split the expressions into 3 categories according to their precedence: *exp0* which deals with operators *in* and *not in* ; *exp1* that deals with operator *==*; *exp2* that deals with operators *+* and *–* ; *exp3* that deals with operators *//*, *\** and *%* and all the other types of expressions inside *exp4*. All the types of expression in each category is divided by or (*<|>*). These expressions are called from *exp0* to *exp4* in the reversed order of their precedence. By doing this splitting I have also solved the issues with left-associative and right-associative expressions.
* For dealing with right and left recursion I have used the schemas described in the book <<“Grammars and parsing with Haskell Using Parser Combinators”. I dealt with recursion in functions *eop0*, *eop1* for right-recusion on operators *in*, *not in* and *==*; I dealt with left-recursion in functions *eop2* and *eop3* for the other operators that require left-recursion. I have chosen to implement recursion by calling the functions that I had instead of using functions line *chainl* or *chainr* from the library because I did not understood how those work but I understood the basic algorithm behind dealing with left-associative and right-associative operations.
* For parsing the *Exprs* which are shown in the grammar tree as being formed of *Expr* *CommaExprs* I have simply used the function *symbol “,”* and it simply parses all the *Expr* that are separated by comma and forms a list *[Expr]* which is returned as type *Exprs*
* For parsing the *Name* I have 2 functions: *nameStart* that checks the first character of a string and if it is a letter and nameRest that checks if the other characters fulfill the requirements and the results are joined into a single *String* that is returned ad *Name* type.

Other than those described previously, I have done parsing of the grammar tree by simply chaining the functions properly much like it is described in <<“Grammars and parsing with Haskell Using Parser Combinators”, Peter Sestoft, Ken Friis Larsen, draft version 2, 2013-09-11>>.

# Testing

I have also done few simple tests. I have used HUnit library to check the results of parsing by using asserts. I assert error in case of that the result offered by the parser is an Error(Left Error) and the test is considered to be correct if the result is a Program type(Right Program). I have done the following tests:

* Test for the code given in the appendixes of the exam text and the parsing works propely. I have also checked in console by parsing the code from the file and the result is exactly the abstract syntax tree from the appendixes.
* Test for checking if the simplest program “a=1+2” is parsed correctly, which is in fact a Decl containing an adding operation. It also proved that it works with no error.
* A test for checking if an wrong code “a+\*2” is actually seen as error (*Unspec* error)
* A test which checks if *Exprs* that contains a series of *Expr* separated by comma is parsed correctly
* A test that takes a simple series of Expr separated by comma which is not inside of a ListComp is detected as an Unspec error
* A test to check if a for loop inside another for loop is parsed properly because I was little worried not to have an infinite loop in program by having a loop between *ListComp*, *ListFor*, *ListIter* and it has proved that the parsing is successful without getting any StackOverflow.
* One test with operations of different priority to show that it is respected
* One last test with a simple operation “a=1+2” with a lot of whitespace including tab and new line to prove that all whitespaces are ignored and the parsing is successful.
* I have done 2 last tests for comparison between right recursion and left recursion for the operators as required for the exam; mostly because making the difference between right recursion and left recursion caused me struggle a lot to solve. None of these tests shown error

Unfortunately my tests do not also compare the results for parsing with the some expected results, they only check if I get *Left* *Error* instead of *Right* *Program*.

# Conclusion

I have implemented all the functionality of the parser that was described in the exam text, I have dealt with the challenges of operator priorities and right or left associative in a quite simple way which has proven to work. I have also done tests to prove that the functionality is working properly and that the parsing ends into Error in case the rules of grammar are not respected. If I had time I would have done little more complex testing by also checking that the results of parsing are the same with expectations, not only check if it ends with *Error* or *Program*.