Advance Programming

Reexamination

Mariuta Nicolae

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*.

When I did the Interpreter I have noticed that my parser does not also work with parsing negative integers and those are actually needed at interpreter. I did not have enough time to fix this but I think it could be easily done as for the parsing of names, to split in 2 parts: one function to check if there is sign ‘–‘ before the integer and the other function to parse the digits in integer.

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.

Question 3: Ant Colony

Structure of the program

I have implemented the Ant Colony by creating two loops: *ant_loop* that handles messages received by one ant process and *colony_loop* that handles the colony process. Inside the *ant_loop* I kept the current coordinates of the and as a tuple *{X,Y}* , one Boolean that says if pen is up or down, the value of the current angle, the whole picture that was drawn by the ant while having the pen up and the process id of the colony process to which it belongs. Inside the colony_loop process I have chosen to keep a list with coordinates of the living ants, a counter for the number of ants that died and nother list containing the pictures of the ants that died.

Implementation for the interaction with ants processes

I have made that inside of the functions *forward*, *left*, *right* and *setpen* to send asynchronous messages to the process as described in the exam, even if it has proven to have some race condition problems as stated. I could also implement by making *rpc* calls to the ant process and also return *{ok}* in case of wrong values passed to the ant but I have chosen to follow the requirements of the exam. Since it was not specified I have chosen to set the value false for the boolean pen that is set to the ant. So the pen has to be set up in order to have the ant draw something when moved.

In case it happens that an ant to die because of wrong values passed to functions *forward*, *left*, *right* or *setpen* I have chosen to simply send an asynchronous message to the colony that holds the ant and simply no longer call the loop of that function when the handling of the message ends. Like this, the process exits the loop and stops. I know it was better to use a separate process to handle the ant and colony processes that die or get blasted but I did not have much time to make such a complex implementations. And however it has proven that even this simple way to solve this point is effective and correct. If I use the function is_process_alive(Pid) before and after the ant process has to stop, I get true before and false after I stop the process by doing as I described.

In case of using the clone function for an ant process it starts a recursive function called *clones* that sends a message to the colony process with information needed by it to create a new ant process: coordinates and angle so the clones have these values similar to these. For this purpose I have also created a function *new_ant_clone(C,{X,Y},Angle)* to create a new ant process with these values set.

Implementation for the interaction with colony processes

For the interaction with the colony I have chosen to only keep a list with process ids of the ants that are alive and in case that one ant has to die, the colony concatenates its picture to the list *Graveyard* which is also returned in case of using the function that requests the picture drawn by the dead ants. I also have a counter *NrDeadAnts* which is incremented each time a message *{From, {ant_died, DeadPicture}}* reaches the colony process.

For the implementation of *blast* function I have chosen that the colony process that has to be stopped is at first calling a recursive function *blastAllAnts* which takes the list of ids that the colony has and asks each ant process to stop in the same manner as in case of ants that die because of errors. After this function ends, then the current colony process is also stopped by simply not calling the *colony_loop* function again.

Since I have the process Ids, the number of dead ants and graveyard already memorized by the colony process then the functions requesting these where easy to implement by just returning the information that is asked.

For the implementation of function *picture* I have chosen to also call a recursive function that requests the *Picture* from each ant with process id stored in *LivingAnts*.

Testing

I have done several tests to check the functions are working properly by using Unit testing. In order to avoid the issues caused by race conditions when calling the asynchronous function one after another I have used the function timer:sleep(1) which makes the function to sleep for 1 millisecond which I have noticed it is enough for that the previous operation is finished by the ant process.

- The first two tests are just checking if the functions for starting a new colony process and a new ant process are working properly
- The next 3 tests are checking that by spawning multiple ants, their corresponding colony process is returning the correct number of ants. In 2 of these tests I have made that some of the ants to die by sending wrong parameters to each of the functions *forward*, *left*, *right* and *setpen* and also checked if the number of dead ants is returned correctly
- I have done one test where I have moved an ant and also rotated it in order to check if the final position is the one that was expected.
- One test that checks the functionality of cloning that starts two processes assigned to same colony and makes multiple spawns for them. At the end it is checked if the total number of ants in the colony is the one that was expected.

- Two tests which prove that colony and processes are stopped properly when killing or blasting them by checking the result offered by *is_process_alive()* function before and after they are killed.
- 2 tests which check if there is data saved into Picture of an ant if the pen is set up or down. I wanted to make sure that the function setpen works properly.

Inside of the file *svg.erl* I have also created a function that takes the code returned by the *pictureToSvg* function and writes it into a file so it was easier for me to visualize the evolution of ants.

Conclusion

I have chosen to solve the challenges of this question in a simple and quite effective manner. Perhaps it is not entirely correct but the functionality of the program is working very well even like this. I have done tests to prove that functions are working properly and that the framework is quite robust in case of that some processes have to stop. Perhaps more error handling for wrong parameters or unexpected failure of a process could have been a good improvement to my implementation. If I had time I would have created another loop process that is controlling the servers for ants and colony.

Appendix A: Antaresia Parser

Appendix A.1 Antaresia parser code

```
{-# OPTIONS_GHC -Wall #-}
module AntaParser
       ( Error
       , parseString
       , parseFile
       )
       where

import SimpleParse
import Data.Char
import Control.Applicative
import AntaAST

--Error type
type Error = ErrorType

data ErrorType = Unspec String | Ambiguous String
      deriving (Show)

entry :: Parser Program
entry = prog <* spaces

--keywords
keywords :: [String]
keywords = ["True", "False", "range", "for", "if", "in", "not"]

-- parser for Name
name :: Parser Name
name = token $ do
                        name_start <- satisfy nameStart
                        name_rest <- munch nameRest
                        if (name_start : name_rest) `elem` keywords
                              then reject
                        else return (name_start : name_rest)
-- names must start with a letter
nameStart :: Char -> Bool
nameStart c = isAlpha c
-- names can include letters, digits and underscores
nameRest :: Char -> Bool
nameRest c = isAlpha c || c == '_' || isDigit c

--parse for integers. It is returned as Expr
intConst :: Parser Integer
intConst = token $
      do
```

```
            integer <- munch1 isDigit
            return $ read integer


--parsing of main program structures
prog :: Parser Program
prog = decls

decls :: Parser Decls
decls = many decl

decl :: Parser Decl
decl = do
            decl_name <- name
            _ <- symbol "="
            decl_expr <- expr
            return (decl_name, decl_expr)


args3 :: Parser Args3
args3 =
      do
            e1 <- expr
            return $ A1 e1
      <|>
      do
            e1 <- expr
            _ <- symbol ","
            e2 <- expr
            return $ A2 e1 e2
      <|>
      do
            e1 <- expr
            _  <- symbol ","
            e2 <- expr
            _  <- symbol ","
            e3<- expr
            return $ A3 e1 e2 e3


exprs :: Parser Exprs
exprs =
      do
            es <- commaExprs
            return es

commaExprs :: Parser [Expr]
commaExprs =
      do
            y <- (expr `sepBy` (symbol ","))
            return y

--parsing of expression
expr :: Parser Expr
expr =
      do
            e <- exp0
```

```
            return e

exp0 :: Parser Expr
exp0 =
      do
            e1 <- exp1
            eop0 e1



--parsing expr operations
eop0 :: Expr -> Parser Expr
eop0 e =
      do
            _ <- symbol "in"
            e2 <- exp1
            er <- eop0 e2
            return $ In e er
      <|>
      do
            _ <- symbol "not"
            _ <- symbol "in"
            e2 <- exp1
            er <- eop0 e2
            return $  NotIn e er
      <|>
            return e


exp1 :: Parser Expr
exp1 =
      do
            e1 <- exp2
            eop1 e1

--right recursion for oprator ==
eop1 :: Expr -> Parser Expr
eop1 e =
      do
            _ <- symbol "=="
            e2 <- exp2
            er <- eop1 e2
            return $ Equal e er
      <|>
            return e

exp2 :: Parser Expr
exp2 =
      do
            e3 <- exp3
            eop2 e3

--parsing expressions Plus and Minus
eop2 :: Expr -> Parser Expr
eop2 e =
      do
            _ <- symbol "+"
```

```
            e3 <- exp3
            eop2 $ Plus e e3
      <|>
      do
            _ <- symbol "-"
            e3 <- exp3
            eop2 $ Minus e e3
      <|>
            return e

exp3 :: Parser Expr
exp3 =
      do
            e4 <- exp4
            eop3 e4

eop3 :: Expr -> Parser Expr
eop3 e =
      do
            _ <- symbol "*"
            e4 <- exp4
            eop3 $ Mult e e4
      <|>
      do
            _ <- symbol "//"
            e4 <- exp4
            eop3 $ Div e e4
      <|>
      do
            _ <- symbol "%"
            e4 <- exp4
            eop3 $ Modulus e e4
      <|>
            return e


exp4 :: Parser Expr
exp4 =
      do
            c1 <- intConst
            return $ IntConst c1
      <|>
      do
            _ <- symbol "True"
            return TrueConst
      <|>
      do
            _ <- symbol "False"
            return FalseConst
      <|>
      do
            n1 <- name
            return $ Name n1
      <|>
      do
            _ <- symbol "range"
            _ <- symbol "("
```

```
            a3 <- args3
            _ <- symbol ")"
            return $ Range a3
      <|>
      do
            _ <- symbol "["
            l1 <- listComp
            _ <- symbol "]"
            return $ ListComp l1
      <|>
      do
            _ <- symbol "("
            e1 <- expr
            _ <- symbol ")"
            return e1




--parsing of the lists data structures
listComp :: Parser ListComp
listComp =
      do
            l1 <-  exprs
            return $ Simple l1
      <|>
      do
            l1 <- expr
            l2 <- listFor
            return $ ListFor l1 l2
--parsing  for
listFor :: Parser ListFor
listFor     =
      do
            _ <- symbol "for"
            n1 <- name
            _ <- symbol "in"
            e1 <- expr
            l1 <- option listIter
            return (n1,e1, l1)

--parsing inter
listIter :: Parser ListIter
listIter =
      do
            l1 <-  pListIter
            return l1

pListIter :: Parser ListIter
pListIter =
      do
            l1 <- listFor
            return $ ListForIter l1
      <|>
```

```
    do
        _ <- symbol "if"
        e1 <- expr
        l1 <-  option listIter
        return $ ListIf e1 $ l1


--utilty functions for starting the parsing
parseString :: String -> Either Error Program
parseString s = checkError (parseEof entry s)

checkError :: [(Program, String)] -> Either Error Program
checkError ((p, _):[]) = Right p
checkError ((_, _):_) = Left (Ambiguous "Could you specify that?")
checkError [] = Left (Unspec "Fatal parsing error")

parseFile :: FilePath -> IO (Either Error Program)
parseFile filename = parseString <$> readFile filename
```

Appendix A.1 Antaresia parser test

```
module Main where
import Test.HUnit
import Data.Either as Either
import AntaParser
import AntaAST


--files with code for resting
test1 = "S = [x*x for x in range(10)] M = [x for x in S if x % 2 == 0]
noprimes = [j for i in range(2, 8) for j in range(i*2, 50, i)] primes = [x
for x in range(2, 50) if x not in noprimes]"
test2 = "a=1+2"
test3 = "a+*2"
test4 = "a = 1 n = [True,False,2+3]"
test5 = "True,False,2+3"
test6 = "M = [x for x in S if [j for i in range(2, 8) for j in range(i*2, 50,
i)]]"
test7 = "a=1+2//5 in True"
test8 = "\n a= \t 1 \t +  \n        2  "
test9 = "a=[1in2not in3in4]"
test10 = "a=[1*2//3%5]"
--trees that are expected for each file1
resultFile1 = [("a",Plus (IntConst 1) (IntConst 2))]


--function that checks if the parsing results into Error or Program
checkResult :: Either Error Program -> Bool
checkResult (Right _) = True
checkResult _ = False

--test asserts
```

```
testParse1 = TestCase $ assertBool "Parsing code in appendix" $ checkResult
(AntaParser.parseString  test1)
testParse2 = TestCase $ assertBool "Parsing simple add" $ checkResult
(AntaParser.parseString  test2)

testParse3 = TestCase $ assertBool "Parsing wrong expression" $ not $
checkResult (AntaParser.parseString  test3)
testParse4 = TestCase $ assertBool "Comma expressions test" $ checkResult
(AntaParser.parseString  test4)
testParse5 = TestCase $ assertBool "Wrong Comma expressions test" $ not $
checkResult (AntaParser.parseString  test5)
testParse6 = TestCase $ assertBool "For inside for" $ checkResult
(AntaParser.parseString  test6)
testParse7 = TestCase $ assertBool "Chain of expression with different
prioprities operands" $ checkResult (AntaParser.parseString  test7)
testParse8 = TestCase $ assertBool "Lost of whitespaces" $ checkResult
(AntaParser.parseString  test8)
testParse9 = TestCase $ assertBool "Right recursion" $ checkResult
(AntaParser.parseString  test9)
testParse10 = TestCase $ assertBool "Left recursion" $ checkResult
(AntaParser.parseString  test10)
--list of tests
tests = TestList [TestLabel "AntaParser testSuite" $ TestList
[testParse1,testParse2, testParse3, testParse4, testParse5, testParse6,
testParse7, testParse8, testParse9, testParse10]]

--main
main = do
      runTestTT tests
```

Appendix B: Antaresia Interpreter

Appendix B.1 Antaresia Interpreter code

```
{-# OPTIONS_GHC -Wall #-}
module AntaInterpreter
       ( runProg
       , Error (..)
       )
       where

import AntaAST

-- You might need the following imports
import Control.Applicative
import Control.Monad
import qualified Data.Map as Map


-- ^ Any runtime error.  You may add more constructors to this type
-- (or remove the existing ones) if you want.  Just make sure it is
-- still an instance of 'Show' and 'Eq'.
data ErrorType = DivideByZero
               | VariableNotFound
               | WrongInput
               | InvalidArguments
               | InvalidExpression
               | NotSameArguments
               | NotImplementedYet
               | Unspec String
              deriving (Show, Read, Eq)
data Error = Error { errorType :: ErrorType,
                                  state :: AntaState}
           deriving (Show, Eq)

-- | A mapping from names to values
type Env = Map.Map Name Value
--state of AntaM program
data AntaState = AntaState
             { prog  :: Program
             , env  :: Env
                   , res :: Result
                   }
           deriving (Show, Eq)

 -- | `initial` constructs the initial state of an Antaresa program
-- given program.
initial :: Program -> AntaState
initial p = AntaState { prog= p
```

```
                    , env = Map.fromList([])
                    , res = []
                      }

 -- | The basic monad in which execution of a Antaresia program takes
 --place.  Maintains the global state and whether or not an error has
 --occurred.
newtype AntaM a = AntaM ( AntaState -> Either String (a,AntaState))

instance Functor AntaM where
  fmap = liftM

instance Applicative AntaM where
  pure = return
  (<*>) = ap

instance Monad AntaM where
    (AntaM p) >>= f = AntaM(\x -> case p x of
                                    Right m -> let Right(a,x') = p x;
                                                   (AntaM q) = f a
                                               in  q x'
                                    Left m -> Left m
                                )
      -- return :: a -> AntaM a
    return a = AntaM(\x -> Right(a,x))


--change state of AntaM
modify :: (AntaState -> AntaState) -> AntaM ()
modify f = AntaM (\s -> Right ((), f s))

 -- | `get` returns the current state of the running AntaM.
getAntaState :: AntaM AntaState
getAntaState = AntaM $ \s -> Right(s,s)

-- | set a new state for the running AntaM.
setAntaState :: AntaState -> AntaM ()
setAntaState m = AntaM $ \s -> Right ((), m)

--change Env in in program
updateEnv :: Name -> Value -> AntaM ()
updateEnv name val =
                    do
                        s <- getAntaState
                        setAntaState  s{ env = Map.insert name val (env s)}

--function that returns the next  Decl in program
--fail the program if there is no Decl left
getDecl :: AntaM Decl
getDecl = do
  stat <- getAntaState
  if length (prog stat) == 0
     then fail "Error no expression left"
     else return $ (prog stat) !! 0
```

```haskell
-- | This function runs the AntaM
interp :: AntaM()
interp = run
    where run = do (n,e) <- getDecl
                   cont <- evalExpr n e
                   when cont run


--function to calculate range expression
range :: Integer -> Integer -> Integer -> [Integer]
range start stop step = [start,(start+step)..stop]
      --  return [start,b..stop]

-- | This function interprets the given expression. It returns True
-- if AntaM is supposed to continue it's execution after this instruction
evalExpr :: Name -> Expr -> AntaM Bool
evalExpr n expr = do
    thisState <- getAntaState
    case expr of
        IntConst a -> do
           intConst n a thisState
        TrueConst -> do
           trueConst n thisState
        FalseConst -> do
           falseConst n thisState
        Name a -> do
            nameExp n a thisState
        Range a -> do
            rangeExpr n a thisState
        Plus e1 e2 -> do
            plusExpr n e1 e2 thisState
        Minus e1 e2 -> do
            minusExpr n e1 e2 thisState
        Mult e1 e2 -> do
            multExpr n e1 e2 thisState
        Div e1 e2 -> do
            divExpr n e1 e2 thisState
        Modulus e1 e2 -> do
            modulusExpr n e1 e2 thisState
        Equal e1 e2 -> do
            equalExpr n e1 e2 thisState
        ListComp l -> do
            listComp n l thisState
        _ ->fail $ showError Error{errorType = InvalidExpression}


--function that inserts into AntaState the value and name of new IntConst
--at the end checks if there is any Decl left into AntaM
intConst :: Name -> Integer -> AntaState -> AntaM Bool
intConst n a x = do
    setAntaState x{prog = tail (prog x), env = Map.insert n (IntVal a) (env
x), res = (n,IntVal a):(res x)}
    newState <- getAntaState
    if length (prog newState) == 0
        then return False
        else return True
```

```
--function that inserts into AntaState the value and name of new TrueConst
--at the end checks if there is any Decl left into AntaM
trueConst :: Name -> AntaState -> AntaM Bool
trueConst n x = do
    setAntaState x{prog = tail (prog x), env = Map.insert n (TrueVal) (env
x), res = (n,TrueVal):(res x)}
    newState <- getAntaState
    if length (prog newState) == 0
        then return False
        else return True

--function that inserts into AntaState the value and name of new FalseConst
--at the end checks if there is any Decl left into AntaM
falseConst :: Name -> AntaState -> AntaM Bool
falseConst n x = do
    setAntaState x{prog = tail (prog x), env = Map.insert n (FalseVal) (env
x), res = (n,FalseVal):(res x)}
    newState <- getAntaState
    if length (prog newState) == 0
        then return False
        else return True

--function that attributes to variable n the value of variable a
--at the end checks if there is any Decl left into AntaM
--returns error if varibale n is not in the program
nameExp :: Name -> Name -> AntaState ->  AntaM Bool
nameExp n a x = do
    case Map.lookup a (env x) of
                    Just v -> setAntaState x{prog = tail (prog x), env =
Map.insert n v (env x), res = (n,v):(res x)}
                    Nothing -> fail $ showError Error{errorType =
VariableNotFound}
    newState <- getAntaState
    if length (prog newState) == 0
        then return False
        else return True

--function to evaluate the range expressions according to the number of
arguments
--at the end checks if there is any Decl left into AntaM
rangeExpr :: Name -> Args3 -> AntaState -> AntaM Bool
rangeExpr n a x = do
    case a of
        A1 (IntConst e1 ) -> setAntaState x{prog = tail (prog x), env =
Map.insert n ( List $ fmap IntVal $ range 0 e1 1) (env x), res = (n,(List $
fmap IntVal $ range 0 e1 1 )):(res x)}
        A2 (IntConst e1 ) (IntConst e2 ) -> setAntaState x{prog = tail (prog
x), env = Map.insert n  (List $ fmap IntVal $ range e1 e2 1) (env x), res =
(n, (List $ fmap IntVal $ range e1 e2 1 )):(res x)}
        A3 (IntConst e1 ) (IntConst e2 ) (IntConst e3 )->  setAntaState
x{prog = tail (prog x), env = Map.insert n (List $ fmap IntVal $ range e1 e2
e3) (env x), res = (n, List $ fmap IntVal $ range e1 e2 e3 ):(res x)}
        _ -> fail $ showError Error{errorType = WrongInput}
    newState <- getAntaState
    if length (prog newState) == 0
        then return False
```

```
        else return True

--function to evaluate the Plus operation expressions
--at the end checks if there is any Decl left into AntaM
--returns error in case any of the arguments is not integer
plusExpr :: Name -> Expr -> Expr -> AntaState -> AntaM Bool
plusExpr n e1 e2 x = do
       case e1 of
          IntConst v1 ->   do
                           case e2 of
                               IntConst v2 -> setAntaState x{prog = tail
(prog x), env = Map.insert n (IntVal (v1+v2)) (env x), res = (n,IntVal
(v1+v2)):(res x)}
                               _ -> fail $ showError Error{errorType =
InvalidArguments}
          _ -> fail $ showError Error{errorType = InvalidArguments}
       newState <- getAntaState
       if length (prog newState) == 0
       then return False
       else return True


--function to evaluate the Minus operation expressions
--at the end checks if there is any Decl left into AntaM
--returns error in case any of the arguments is not integer
minusExpr :: Name -> Expr -> Expr -> AntaState -> AntaM Bool
minusExpr n e1 e2 x = do
       case e1 of
          IntConst v1 ->   do
                           case e2 of
                               IntConst v2 -> setAntaState x{prog = tail
(prog x), env = Map.insert n (IntVal (v1-v2)) (env x), res = (n,IntVal (v1-
v2)):(res x)}
                               _ -> fail $ showError Error{errorType =
InvalidArguments}
          _ -> fail $ showError Error{errorType = InvalidArguments}
       newState <- getAntaState
       if length (prog newState) == 0
       then return False
       else return True


--function to evaluate the Multiplication operation expressions
--at the end checks if there is any Decl left into AntaM
--returns error in case any of the arguments is not integer
multExpr :: Name -> Expr -> Expr -> AntaState -> AntaM Bool
multExpr n e1 e2 x = do
       case e1 of
          IntConst v1 ->   do
                           case e2 of
                               IntConst v2 -> setAntaState x{prog = tail
(prog x), env = Map.insert n (IntVal (v1*v2)) (env x), res = (n,IntVal
(v1*v2)):(res x)}
                               _ -> fail $ showError Error{errorType =
InvalidArguments}
          _ -> fail $ showError Error{errorType = InvalidArguments}
       newState <- getAntaState
```

```
        if length (prog newState) == 0
        then return False
        else return True

--function to evaluate the Divide operation expressions
--at the end checks if there is any Decl left into AntaM
--returns error in case any of the arguments is not integer or the second
operator is 0
divExpr :: Name -> Expr -> Expr -> AntaState -> AntaM Bool
divExpr n e1 e2 x = do
        case e1 of
           IntConst v1 ->   do
                             case e2 of
                                 IntConst v2 ->
                                       if v2 /= 0
                                       then setAntaState x{prog = tail (prog
x), env = Map.insert n (IntVal (v1 `div` v2)) (env x), res = (n,IntVal (v1
`div` v2)):(res x)}
                                       else fail $ showError Error{errorType =
DivideByZero}
                                 _ -> fail $ showError Error{errorType =
InvalidArguments}
           _ -> fail $ showError Error{errorType = InvalidArguments}
        newState <- getAntaState
        if length (prog newState) == 0
        then return False
        else return True


--function to evaluate the Modulus operation expressions
--at the end checks if there is any Decl left into AntaM
--returns error in case any of the arguments is not integer or the second
operator is not >0
modulusExpr :: Name -> Expr -> Expr -> AntaState -> AntaM Bool
modulusExpr n e1 e2 x = do
        case e1 of
           IntConst v1 ->   do
                             case e2 of
                                 IntConst v2 ->
                                       if v2 >0
                                       then setAntaState x{prog = tail (prog
x), env = Map.insert n (IntVal (v1 `mod` v2)) (env x), res = (n,IntVal (v1
`mod` v2)):(res x)}
                                       else fail $ showError Error{errorType =
DivideByZero}
                                 _ -> fail $ showError Error{errorType =
InvalidArguments}
           _ -> fail $ showError Error{errorType = InvalidArguments}
        newState <- getAntaState
        if length (prog newState) == 0
        then return False
        else return True



--function to evaluate the Equals operation expressions
--at the end checks if there is any Decl left into AntaM
```

```
--returns error in case arguments are not of the same type
equalExpr :: Name -> Expr -> Expr -> AntaState -> AntaM Bool
equalExpr n (IntConst e1) (IntConst e2) x = do
       if e1 == e2
        then setAntaState x{prog = tail (prog x), env = Map.insert n
(TrueVal) (env x), res = (n,TrueVal):(res x)}
        else setAntaState x{prog = tail (prog x), env = Map.insert n
(FalseVal) (env x), res = (n,FalseVal):(res x)}
       newState <- getAntaState
       if length (prog newState) == 0
        then return False
        else return True
equalExpr n (TrueConst) (TrueConst) x = do
       setAntaState x{prog = tail (prog x), env = Map.insert n (TrueVal) (env
x), res = (n,TrueVal):(res x)}
       newState <- getAntaState
       if length (prog newState) == 0
        then return False
        else return True
equalExpr n (FalseConst) (FalseConst) x = do
       setAntaState x{prog = tail (prog x), env = Map.insert n (TrueVal) (env
x), res = (n,TrueVal):(res x)}
       newState <- getAntaState
       if length (prog newState) == 0
        then return False
        else return True
equalExpr n (FalseConst) (TrueConst) x = do
       setAntaState x{prog = tail (prog x), env = Map.insert n (FalseVal)
(env x), res = (n,FalseVal):(res x)}
       newState <- getAntaState
       if length (prog newState) == 0
        then return False
        else return True
equalExpr n (TrueConst) (FalseConst) x = do
       setAntaState x{prog = tail (prog x), env = Map.insert n (FalseVal)
(env x), res = (n,FalseVal):(res x)}
       newState <- getAntaState
       if length (prog newState) == 0
        then return False
        else return True
equalExpr n (ListComp e1) (ListComp e2) x = do
       if e1 == e2
        then setAntaState x{prog = tail (prog x), env = Map.insert n
(TrueVal) (env x), res = (n,TrueVal):(res x)}
        else setAntaState x{prog = tail (prog x), env = Map.insert n
(FalseVal) (env x), res = (n,FalseVal):(res x)}
       newState <- getAntaState
       if length (prog newState) == 0
        then return False
        else return True
equalExpr _ _ _ _ = fail $ showError Error{errorType = NotSameArguments}

--function to evaluate the ListCom operation expressions
--returns error because I did not manage to implement these
listComp :: Name -> ListComp -> AntaState -> AntaM Bool
listComp _ _ _ = fail $ showError Error{errorType = NotImplementedYet}
```

```
--show corresponding message error in case of failure
showError :: Error -> String
showError x = case (errorType x) of
    DivideByZero -> "Cannot divide by zero"
    VariableNotFound -> "Variable was not found"
    WrongInput -> "Value give is of wrong type"
    InvalidArguments -> "Operator arguments must be integers"
    InvalidExpression -> "Expression is invalid"
    NotSameArguments -> "Arguments of equals must be same type"
    NotImplementedYet -> "List comprehesions not implemented yet! They are
about to come in next patch."
    Unspec a -> "Unspec String" ++ show a

-- | Run the given program on the AntaM
program :: Program -> Either String AntaState
program p =let (AntaM f) = interp
               in fmap snd $ f $ initial p



--starts the program function and checks the result to extract error or final
Result
runProg :: Program -> Either String Result
runProg p = case program p of
    Left e -> Left e
    Right n -> Right (Map.toList(env n))
```

Appendix B.2 Antaresia Interpreter test

```
module Main where
import Test.HUnit
import Data.Either as Either
import AntaInterpreter
import AntaAST



--different tests
test1 = [("a",(IntConst 1))]
test2 = [("a",(TrueConst))]
test3 = [("a",(TrueConst)),("b",(Name "a"))]
test4 = [("a",(TrueConst)),("b",(Name "c"))]
test5 = [("a",Range (A3 (IntConst 1) (IntConst 12) (IntConst 3)))]
test6 = [("a",Plus (IntConst 1) (IntConst 3))]
test7 = [("a",Minus (IntConst 1) (IntConst 3))]
test8 = [("a",Mult (IntConst 3) (IntConst 5))]
test9 = [("a",Div (IntConst 6) (IntConst 5))]
test10 = [("a",Div (IntConst 6) (IntConst 0))]
test11 =[("a",Modulus (IntConst 15) (IntConst 2))]
```

```
test12 =[("a",Equal (IntConst 15) (IntConst 12))]
test13 =[("a",Equal (IntConst 15) (TrueConst))]
test14 =[("a",ListComp (Simple [IntConst 1,IntConst 2,IntConst 3]))]



--function that checks if the parsing results into Error or Program
checkResult :: Either String Result -> Bool
checkResult (Right _) = True
checkResult _ = False


--test asserts
testParse1 = TestCase $ assertBool "Test IntConst" $ checkResult
(AntaInterpreter.runProg  test1)
testParse2 = TestCase $ assertBool "Test bool expression" $ checkResult
(AntaInterpreter.runProg test2)

testParse3 = TestCase $ assertBool "Test expression Name Name" $ checkResult
(AntaInterpreter.runProg  test3)
testParse4 = TestCase $ assertBool "Test expression Name Name where argument
is invalid" $ not $ checkResult (AntaInterpreter.runProg  test4)
testParse5 = TestCase $ assertBool "Test range with 3 args" $ checkResult
(AntaInterpreter.runProg  test5)
testParse6 = TestCase $ assertBool "Test plus expression" $ checkResult
(AntaInterpreter.runProg  test6)
testParse7 = TestCase $ assertBool "Test minus epression" $ checkResult
(AntaInterpreter.runProg  test7)
testParse8 = TestCase $ assertBool "Test multiply expression" $ checkResult
(AntaInterpreter.runProg  test8)
testParse9 = TestCase $ assertBool "Test div expression" $ checkResult
(AntaInterpreter.runProg  test9)
testParse10 = TestCase $ assertBool "Test div by 0" $ not $ checkResult
(AntaInterpreter.runProg  test10)
testParse11 = TestCase $ assertBool "Test modulus" $ checkResult
(AntaInterpreter.runProg test11)
testParse12 = TestCase $ assertBool "Left recursion" $ checkResult
(AntaInterpreter.runProg test12)
testParse13 = TestCase $ assertBool "Test equals between variables of the
same type" $ checkResult (AntaInterpreter.runProg test13)
testParse14 = TestCase $ assertBool "Test equals between variables of
different types" $ not $ checkResult (AntaInterpreter.runProg test14)

--list of tests
tests = TestList [TestLabel "AntaParser testSuite" $ TestList
[testParse1,testParse2, testParse3, testParse4, testParse5, testParse6,
testParse7, testParse8, testParse9, testParse10, testParse11, testParse12,
testParse13, testParse14]]

--main
main = do
      runTestTT tests
```

Appendix C: Ant Colony

Appendix C.1 Anttalk code

```
-module(anttalk).

%% Exports
-export([ %% Ant API
          forward/2,
          left/2,
          right/2,
          setpen/2,
              clone/2,
          position/1,

          %% Colony API
              start/0,
          blast/1,
          new_ant/1,
          picture/1,
          ants/1,
          graveyard/1]).

%%%====================================================================
%%% API
%%%====================================================================

%% Ant API
forward(A,N) ->
      async(A, {moveForward, N}).

left(A,D) ->
      async(A, {rotateLeft, D}).

right(A,D) ->
      async(A, {rotateRight, D}).

setpen(A,P) ->
      async(A, {set_pen, P}).



position(A) ->
      rpc(A, get_position).



clone(A, N) ->
      clones(A,N,[]).
```

```
%% Colony API
start() ->
      {ok, spawn(fun() -> colony_loop([], 0, []) end)}.

blast(C) ->
      async(C, blast_colony).

new_ant(C) ->
      Pid = spawn(fun() -> ant_loop({0,0}, false, 0, [], C) end),
      async(C, {newAnt,Pid}),
      {ok,Pid}.


picture(C) ->
      rpc(C,get_picture).

ants(C) ->
      rpc(C,all_ants).

graveyard(C) ->
      rpc(C,get_graveyard).

%%%=====================================================================
%%% Internal functions
%%%=====================================================================

%%% Helping functions

%%recursively make all clones that are needed
clones(_,0,AL)  -> {ok,AL};
clones(A,N,AL) when n>0 -> {ok,L}  = rpc(A, clone_ant),
                                     clones(A,N-1,[L|AL]).

%%recursivley kill each ant process in the colony

blastAllAnts(LivingAnts) -> blastAllAntsOneByOne(LivingAnts).


blastAllAntsOneByOne([]) -> stop;
blastAllAntsOneByOne([OneAnt|LivingAnts]) ->
                                     async(OneAnt, stopAnt),
                                     blastAllAntsOneByOne(LivingAnts).

concatenateAllLivingAnts(AntIds) ->
concatenateAllLivingAntsOneByOne(AntIds,[]).

concatenateAllLivingAntsOneByOne([],Picture) -> Picture;
concatenateAllLivingAntsOneByOne([Ant|AntIds],Picture) ->

      {ok,NextPic} = rpc(Ant,take_picture),

      NewPicture = NextPic++Picture,

      concatenateAllLivingAntsOneByOne(AntIds,NewPicture).


new_ant_clone(C,{X,Y},Angle) ->
```

```
      Pid = spawn(fun() -> ant_loop({X,Y}, false, Angle, [], C) end),
      async(C, {newAnt,Pid}),
      {ok,Pid}.



%%% Communication primitives

async(Pid, Msg) ->
    Pid ! Msg.

rpc(Pid, Request) ->
    Pid ! {self(), Request},
    receive
      {Pid, Response} ->
          Response
    end.

reply(From, Msg) ->
    From ! {self(), Msg}.







%%% Server loops

% Ant loop
ant_loop({X,Y},Pen,Angle, Picture, Colony) ->
      receive
            {moveForward, N} ->
                  case is_integer(N) of
                        false ->
                              rpc(Colony,{ant_died,Picture}),
                              io:format("Ant dead!");
                        true ->
                              if
                                    N < 0 ->
                                          rpc(Colony,{ant_died,Picture}),
                                          io:format("Ant dead!");
                                    true ->
                                          case Angle of
                                                0 ->
                                                      Mx = X,
                                                      My = Y+N;
                                                90 ->
                                                      Mx = X+N,
                                                      My = Y;
                                                180 ->
                                                      Mx = X,
                                                      My = Y-N;
                                                270 ->
                                                      Mx = X-N,
```

```
                                                My = Y
                                          end,
                                    if
                                          Pen == true ->
                                                Pts =
[{{X,Y},{Mx,My}}|Picture],
                                                io:format("old picture:
~p",[Picture]),
                                                ant_loop({Mx,My},true,
Angle, Pts, Colony);

                                          true  ->
                                                ant_loop({Mx,My},Pen,
Angle, Picture, Colony)
                                    end

                              end
                  end;

            {rotateLeft, D} ->
                  if
                        (D == 0) or (D == 90) or (D == 180) or (D == 270) ->
                              M = Angle + D,
                              A = M rem 360,
                              ant_loop({X,Y},Pen, A, Picture, Colony);
                        true ->
                              rpc(Colony,{ant_died,Picture}),
                              io:format("Ant dead!")
                  end;

            {rotateRight, D} ->
                  if
                        (D == 0) or (D == 90) or (D == 180) or (D == 270) ->
                              M = Angle - D + 360,
                              A = M rem 360,
                              ant_loop({X,Y},Pen, A, Picture, Colony);
                        true ->
                              rpc(Colony,{ant_died,Picture}),
                              io:format("Ant dead!")
                  end;

            {set_pen, P} ->
                  case P of
                        up ->
                              ant_loop({X,Y},true, Angle, Picture, Colony);
                        down ->
                              ant_loop({X,Y},false, Angle, Picture, Colony);
                        _ ->
                              rpc(Colony,{ant_died,Picture}),
                              io:format("Ant dead!")
                  end;

            {From, get_position} ->
                  reply(From, {ok, {X,Y}}),
                  ant_loop({X,Y},Pen, Angle, Picture, Colony);

            {From, clone_ant} ->
```

```
                  A = new_ant_clone(Colony,{X,Y},Angle),
                  reply(From,{ok,A}),
                  ant_loop({X,Y},Pen, Angle, Picture, Colony);
            stopAnt ->
                  io:format("Ant dead!");
            {From, take_picture} ->
                  reply(From,{ok,Picture}),
                  ant_loop({X,Y},Pen, Angle, Picture, Colony)

      end.

% ant colony loop
colony_loop (LivingAnts, NrDeadAnts, Graveyard) ->
      receive
            blast_colony ->
                  blastAllAnts(LivingAnts);

            {newAnt,Pid} ->
                  NewLivingAnts = [Pid| LivingAnts],
                  %io:format("ant added~p",[NewLivingAnts]),
                  colony_loop(NewLivingAnts,NrDeadAnts, Graveyard);

            {From, get_picture} ->
                  Pict = concatenateAllLivingAnts(LivingAnts),
                  reply(From, {ok,Pict}),
                  colony_loop(LivingAnts,NrDeadAnts, Graveyard);

            {From, all_ants} ->
                  reply(From, {ok,{LivingAnts,NrDeadAnts}}),
                  colony_loop(LivingAnts,NrDeadAnts, Graveyard);

            {From, get_graveyard} ->
                  reply(From, {ok, Graveyard}) ,
                  colony_loop(LivingAnts,NrDeadAnts, Graveyard);

            {From, {ant_died, DeadPicture}} ->
                  %io:format("To the graveyard! ~p", [DeadPicture]),
                  NewGraveyard = DeadPicture ++ Graveyard,
                  NewNrDeadAnts = NrDeadAnts + 1,
                  NewLivingAnts = LivingAnts -- [From],
                  reply(From,{died}),
                  io:format("To the graveyard! ~p", [NewGraveyard]),
                  colony_loop(NewLivingAnts,NewNrDeadAnts, NewGraveyard)

      end.
```

Appendix C.2 Anttalk test

```
-module(anttalktest).


-import(anttalk,[ %% Ant API
          forward/2,
          left/2,
          right/2,
          setpen/2,
              clone/2,
          position/1,

          %% Colony API
              start/0,
          blast/1,
          new_ant/1,
          picture/1,
          ants/1,
          graveyard/1]).
-include_lib("eunit/include/eunit.hrl").

%test basic init of a colony
anttalk_initcolony_test() ->
      {ok,C} = anttalk:start(),
      ?assert(is_pid(C)).

%test init of an ant
anttalk_initant_test() ->
      {ok,C} = anttalk:start(),
      {ok,A} = anttalk:new_ant(C),
      ?assert(is_pid(A)).



%test init of multiple ants
anttalk_init5ants_test() ->
      {ok,C} = anttalk:start(),
      anttalk:new_ant(C),
      anttalk:new_ant(C),
      anttalk:new_ant(C),
      anttalk:new_ant(C),
      anttalk:new_ant(C),
      {ok,{AL,N}} = anttalk:ants(C),
      ?assert((N==0) and (length(AL)==5)).

%test init of multiple ants and then kill one ant by sending wrong forward
parameter
anttalk_init5antsAndKill1_test() ->
      {ok,C1} = anttalk:start(),
      anttalk:new_ant(C1),
      anttalk:new_ant(C1),
      anttalk:new_ant(C1),
      anttalk:new_ant(C1),
      {ok, A1} = anttalk:new_ant(C1),
```

```
      anttalk:forward(A1,-9),
      timer:sleep(1),
      {ok,{AL,N}} = anttalk:ants(C1),
      ?assert((N==1) and (length(AL)==4)).

%test init of multiple ants and then kill all ants by sending wrong
%forward parameter , wrong setpen, left and right parameters
anttalk_init5antsAndKillAll_test() ->
      {ok,C} = anttalk:start(),
      {ok, A1} = anttalk:new_ant(C),
      {ok, A2} = anttalk:new_ant(C),
      {ok, A3} = anttalk:new_ant(C),
      {ok, A4} = anttalk:new_ant(C),
      {ok, A5} = anttalk:new_ant(C),
      anttalk:forward(A1,-9),
      anttalk:forward(A2,1.3),
      anttalk:setpen(A3,sdsaasd),
      anttalk:left(A4,45),
      anttalk:right(A5,92),
      timer:sleep(1),
      {ok,{AL,N}} = anttalk:ants(C),
      ?assert((N==5) and (length(AL)==0)).



%move ant and check new position
anttalk_moveAntAndCheckNewPosition_test() ->
      {ok,C} = anttalk:start(),
      {ok, A} = anttalk:new_ant(C),
      timer:sleep(1),
      anttalk:forward(A,9),
      timer:sleep(1),
      anttalk:left(A,90),
      timer:sleep(1),
      anttalk:right(A,90),
      timer:sleep(1),
      anttalk:forward(A,9),
      timer:sleep(1),
      {ok,{X,Y}} = anttalk:position(A),
      ?assert((X==0) and (Y==18)).


%test the functionality of clone function
anttalk_cloneAnts_test() ->
      {ok,C} = anttalk:start(),
      {ok, A} = anttalk:new_ant(C),
      {ok, B} = anttalk:new_ant(C),
      timer:sleep(1),
      anttalk:clone(A,9),
      timer:sleep(1),
      anttalk:clone(B,1),
      timer:sleep(1),
      {ok,{AL,N}} = anttalk:ants(C),
      ?assert((N==0) and (length(AL)==12)).

%test that ant acutally dies
anttalk_killAnt_test() ->
```

```
      {ok,C} = anttalk:start(),
      {ok, A} = anttalk:new_ant(C),
      S1 = is_process_alive(A),
      anttalk:forward(A,-2),
      timer:sleep(1),
      S2 = is_process_alive(A),
      timer:sleep(1),
      ?assert(S1 and (not S2)).

%test blast
anttalk_blast_test() ->
      {ok,C} = anttalk:start(),
      {ok, A} = anttalk:new_ant(C),
      S1 = is_process_alive(C),
      S2 = is_process_alive(A),
      anttalk:blast(C),
      timer:sleep(1),
      S3 = is_process_alive(C),
      S4 = is_process_alive(A),
      timer:sleep(1),
      ?assert(S1 and S2 and (not S3) and (not S4)).

%move ant and check picture
anttalk_picture_test() ->
      {ok,C} = anttalk:start(),
      {ok, A} = anttalk:new_ant(C),
      anttalk:setpen(A,up),
      timer:sleep(1),
      anttalk:forward(A,9),
      timer:sleep(1),
      anttalk:forward(A,9),
      timer:sleep(1),
      {ok,P} = anttalk:picture(C),
      ?assert(length(P)==2).

%move ant without pen true and check picture
anttalk_picturewithoutpen_test() ->
      {ok,C} = anttalk:start(),
      {ok, A} = anttalk:new_ant(C),
      anttalk:setpen(A,down),
      timer:sleep(1),
      anttalk:forward(A,9),
      timer:sleep(1),
      anttalk:forward(A,9),
      timer:sleep(1),
      {ok,P} = anttalk:picture(C),
      ?assert(length(P)==0).


%move ant, kill ant and check graveyard
anttalk_graveyard_test() ->
      {ok,C} = anttalk:start(),
      {ok, A} = anttalk:new_ant(C),
      anttalk:setpen(A,up),
      timer:sleep(1),
      anttalk:forward(A,9),
      timer:sleep(1),
```

```
      anttalk:forward(A,9),
      anttalk:forward(A,-9),
      timer:sleep(1),
      {ok,P} = anttalk:graveyard(C),
      ?assert(length(P)==2).
```

Appendix C.2 svg

```
-module(svg).
-export([pictureToSvg/1,
             write_to_file/1]).


%%--------------------------------------------------------------------
%% @doc Create a string that is a SVG image representing the given picture.
%%
%% @spec pictureToSvg(Pic :: Picture) -> string()
%%  Position = {integer(), integer()}
%%  LineSeg  = {Position, Position}
%%  Picture  = [LineSeg]
%%
%% @end
%%--------------------------------------------------------------------
pictureToSvg (Picture) ->
    Points = lists:append(tuple_to_list(lists:unzip(Picture))),
    {Minx,Maxy} = lists:foldl (fun({X1,Y1}, {X2,Y2}) ->
                                       {min(X1, X2), max(Y1, Y2)} end,
                               hd(Points), tl(Points)),
    Xdir = 1 + if Minx < 0 -> abs(Minx); true -> 0 end,
    Ydir = 1 + Maxy,
    Lines = lists:map(fun svgline/1, Picture),
    lists:flatten(["<svg xmlns=\"http://www.w3.org/2000/svg\">",
                   string_format("<g transform=\"translate(~B, ~B) scale(1,-
1)\">~n",
                                 [Xdir, Ydir]),
                  Lines,
                  "</g></svg>"]).


svgline ({{X1,Y1}, {X2,Y2}}) ->
    string_format("<line style=\"stroke-width: 2px; stroke:black;
fill:white\"\
          \ x1=\"~B\" x2=\"~B\" y1=\"~B\" y2=\"~B\" />~n" , [X1, X2, Y1,
Y2]).

string_format(S, L) ->
    lists:flatten(io_lib:format(S, L)).

write_to_file(S) ->
file:write_file("/Picture.svg", io_lib:fwrite("~p.\n", [S])).
```