Advanced Programming

2015 Exam

Alberto Terceño Ortega
bkx189

November 5, 2015

Contents

1 SubScript Parser

1.1 Parser Library

Code for this question can be found in the src/subs folder and in section 1.1 of the appendix. The library I have used for my parser is SimpleParse.hs, which uses monads for parsing. The reason I have chosen it is that I used it before for doing the Salsa Parser in Assignment 2 (A parser about a domain-specific language), so I have some experience dealing with it. I believe that this library is pretty easy to read and use as well as it gives you a clear overview of monadic parsing. What is more, the exam paper doesn't restrict any of its features so that is another good point about it.
As it is commented in the code, I have used different munch and munch1 functions from the ones declared in SimpleParse.hs. These modified functions were done by the Teacher Assistant Maya Saietz during one of my laboratory group sessions. The file which the functions where taken from is called Expr.hs and can be found in the Team τ folder in Absalon. The reason for using them instead of the original ones is that these functions are really simple (even though they use mutual recursion) and clear to use. I have also used the digit function, taken from the same file in order to parse the SubScript numbers.
Another function which I copied into my file is trim, which I found it in this Wikipedia link (https://en.wikipedia.org/wiki/Trimming_(computer_programming)#Haskell) so I could easily trim the string provided as an argument for our parser, making it parsing correctly (otherwise it won't remove the blank spaces after the last statement so the result won't be a correct solution).
Finally, I added to my file some functions and ideas taken from the Parser Assignment (such as using parseEof or parsing identifiers), some of them fairly modified so I can work with them in this parser. All these functions are commented in the code and the reason for I have used them is that both I got positive feedback from my TA when I used them in the assignment and the fact that they fit (more or less, I have obviously made some changes in some of them) well into the SubScript parser.
I could have included all these features in the SimpleParse file but I decided to keep working with the original version of this library (included in the src/subs directory and in Absalon), since I didn't want to make any changes on it.

1.2 Grammar

The parser runs under the version 1.3 of the grammar specified in the exam paper. Nevertheless, I had to make some changes to the original grammar in order to get an equivalent grammar which avoids left recursion and gives each operator its right priority.
The original and modified grammar can be found as a comment in the parser file but I will paste here the modify grammar, so I can easily discuss the steps I took to get it.

```
Stms ::= e | Stm Stms

Stm ::= 'var' Ident AssignOpt ';' | Expr ';'
```

```
TerminalExpr ::=  Number
          | String
          | 'true'
          | 'false'
          | 'undefined'
          | Ident
          | Ident = FunCall
          | '[' Exprs ']'
          | '[' 'for' '(' Ident 'of' Expr ')' ArrayCompr Expr ']'
          | '(' Expr ')'

Expr ::= Expr1 ',' Expr | Expr1

Expr1 ::= Topt5

Topt5 ::= Ident '=' Topt5 | Topt Topt4

Topt4 ::= '===' Topt Topt4 | Topt3

Topt3 ::= '<' Topt Topt3 | Topt2

Topt2 ::= '+' Topt Topt2 | '-' Topt Topt2 | e

Topt ::= TerminalExpr T

T ::= '*' TerminalExpr T | '%' TerminalExpr T | e
```

The first thing I did with the original grammar was separate the expressions I denominate "Terminal Expressions" and the "Non-Terminal" ones. Terminal expressions don't have left recursion so are easier to deal with them. Then, the next step was dealing with operators precedency. Having these terminal expressions, I named different levels of priority, namely, ToptX, where X denotes the level of the operator priority (it could be confusing but the operators in Topt(X-1) has more priority than the ones in ToptX). After designing these levels I added an Epsilon symbol in Topt2, in order to stop recursion. Notice how there is a path from ToptX to Topt2 so every level finishes the recursion.
After placing the terminal expressions in Topt and adding the highest priority operators in T, I had to include the '=' operator in this heap of levels as the operator with less precedence. Because of this I had to be careful when converting the grammar I had because after an identifier and the '=' operator there could be a FunCall variable in the grammar.

Finally I included the ';' symbol in the Stm variable since I consider that the semicolon is part of a statement as well as the parser for the statement should yield an error rather than a top-level parser.

1.3 Whitespace handling

Whitespace handling is mainly done in three points. The first one is when I trim the string provided for the parseString function. Doing this, I remove the spaces from the front (the parser would have removed them but it is nicer to get them removed before starting) and the back. In this latter case, since I use parseEof (see that I differentiate cases when I parse the string in ParseString returning different errors or the correct program) the parser wouldn't be able to remove the spaces after the last semicolon of the last statement, being unable to parse a possibly right string.
The second one is when 'var' and '.' show up. In the first case, I use many1 spaces so I can assure that at least there is going to be one space after parsing 'var'. On the other hand, parsing '.', I use the char function, so after parsing the last token (which didn't remove spaces after it for sure) there has to be the '.' symbol. Furthermore, afterwards I parse the identifiers without calling the token function.
Last but not least, the third point is that I use the function symbol in the terminal expressions, so I can be sure that all the arbitrary blanks will be removed.

1.4 Parsing testing

The test file can be found in the src/subs folder and in appendix 2.1. Instructions to run the tests can be found in that appendix section as well.
I've made 15 tests using the HUnit module since it is a really simple and effective way to test different cases for the parser. I believe that using QuickCheck would have been more elegant since it uses functional programming for generating random tests, but I've struggled with it all through the course and I haven't been enough confident to use it in the exam.
Anyway, the tests are labelled in different sections so it is easier to see what they try.
I have made some tests to check that strings are parsed right, since they cannot contain the single quote character. I coded some tests checking that parsing numbers is right (The minus sign is parsed ok and numbers cannot have more than eight digits). Several primitive functions are tested and I try to parse as well a function which is not included in the grammar, returning NoParse.
Finally, I made some tests regarding precedences and whitespace handling.
All the tests can be run compiling ParserTest.hs and executing the main function.

1.5 Assesment

Based on the tests I ran on my parser and the steps I took obtaining the modified grammar, I think that the code I provide for this question is correct as well as it provides the functions asked in the exam paper.
I have used the skeleton provided in the zip file for this question and I have barely modified its

declarations.
Regarding the code design, I have to say that I organised it in a way so the reader can see how the code is parsed from top to the bottom, i.e. first showing how a program is parsed, how the statements are parsed and so on. Thus, I believe that the code is simple and easy to understand. I also put comments above that functions which I think that could be a bit difficult to understand at first sight.
Nevertheless, I may have written too many auxiliary functions that I could have avoid using features from SimpleParse that I have simply missed.

2 SubScript Interpreter

2.1 Overview

Code for this question can be found in the src/subs folder and in section 1.2 of the appendix.
To begin with, I have to say that I have coded the file from the skeleton provided along the exam paper. I left the comments written on it and I haven't changed many previous declarations of the skeleton. Maybe the most significative change is that I have declared initialPEnv as the original Primitive Environment, which was previously wrapped in a where clause.
The Functor and Applicative type classes have been coded with the information given in the lectures slides to do so.
The bind operator for the monad has been defined using the do syntax but I also wrote an equivalent definition without the syntactic sugar in a comment above.
It is clear to see that the program is run recursively interpreting each of its statements. Finally, we get the result using an auxiliary function (which will be useful evaluating list of expressions as well).
After the monad and runProg definitions the code is structured in blocks: first we find the definition for the primitive functions, then the setters and getters of the monad, and finally the evalExpr function, which interprets each one of the expressions of SubScript.
All the helper functions for each one of the functions belonging to these blocks are located next to their correspondent functions where they are used. The reason I've done this is that there are many of this helper functions in this question so putting altogether in the bottom of the file would make the code quite unreadable.
Another good reason to support this argument is that the structure for the code is the right one if we are willing to improve SubScript with new features. See that the structure of the code in the parser provides the user an easy way to add more features to the grammar as well, so, in general, these new features shouldn't change and relocate a big amount of code in both files.

2.2 Array Comprehensions

According to the tests I ran on the interpreter, I can assure than the interpreter works fine, but not totally right with array comprehensions.

Although we can see that all evalExpr clauses are defined for array comprehensions, I haven't been able to implement the evalExprAux function (this function is called by evalExpr when array comprehensions have to be dealt with) for all the cases provided by array comprehensions.
The approach I used to face this expressions (which is not the right one at all since the amount of code generated by evalExprAux is absolutely enormous) is defining evalExprAux as a function which gets an expression (I decided to add the Compr constructor to make the function definition a bit clearer although I have to admit that it could be sometimes rather annoying while I was coding) and an environment. This environment provides the environment in the current scope of the array comprehension.
I need to say that for the cases where there is a single for expression the interpreter works fine. evalExprAux takes the environment (The current environment of our program), evaluating each one of the values in the array or string of the for, which is done in a temporary environment (recall that this environment is created from the one passed as an argument to the evalExprAux function) that will be deleted afterwards, returning to the previous environment. The interpreter also works correctly with for loops which contains a if expression with no more for loops nested on it.
The way it works is pretty similar to the previous working case, but in this case we have to analyse if evaluating the expression with the temporary environment returns a true or a false value. Then we will decide to return or not the value obtained.
We can notice that there are used many functions defined in the coded pretty similar to how map performs, but I haven't been able to use some function in order to take advantage of this situation. This makes me think that I could have saved many auxiliary functions in this part of the code.
As I have commented in the code, I couldn't manage to define evalExprAux for the case where a for loop wraps in an if expression which also contains a for loop, so this feature is unavailable in the code. The reason I have done this is because I didn't feel comfortable with my code, I tried different approaches but all of them ended up in writing many different cases and auxiliary functions, making me unable to use the functions I have previously defined in this part.
The main approach I have followed is to use evalExprAux, which get as arguments a for loop expression and an environment, which reflects the scope of the variables in that for expression. As it is done in expressions which have nested for loops, I try to create different temporary environments (using the kind of map functions I have previously mentioned) which are sent to evalExprAux again. However, I couldn't figure out how to restore the previous scopes and environments so many results in nested for loops are incorrect, as it is seen in the tests for this section. Maybe, the correct approach could have been creating a temporary environment which would be modified each time (but that I cannot find out how the previous environment should be restored after using the temporary one without destroying some changes I've made), rather than creating a list of temporary environments. The bad point about this possible approach is that we should pass this temporary environment around every function, and this is one of the reasons I've found more difficult about dealing with array comprehensions, namely, the excessive amount of parameters in the functions which evaluate them.

2.3 Interpreter Testing

Tests can be found in src/subs folder and in section 2.2 of the appendix.
The tests used for this part has been programmed in JavaScript files using SubScript syntax. They can be run using the command prompt “runhaskell Subs.hs <nameOfTestFile>.js”.
Notice that I also indirectly test the parser I have coded here, as the files are first parsed and then interpreted thanks to the Subs.hs file.
I have made three tests in order to show that there are things working right (interpreterTest_WorkingExpressions.js), things that don’t work the way it is desired to be (interpreterTest_WorkingWrongExpressions.js) and expressions that simply, don’t work (interpreterTest_NotWorkingExpressions.js).
It is possible to see that the variables used inside a for expression get back to their previous value, which is correct (running the provided test scope.js gets a satisfactory result). Some primitive functions are tested as well as some simple for expressions with both arrays and strings on it.
Although the array comprehension doesn’t work correctly in the second test, if we run the intro.js test provided in the zip file, we can see that the many_a variable holds the right result. This is due to the fact I mentioned that the variables scopes are not restored correctly in nested for loops in some cases.

2.4 Assesment

After seeing the results provided by the different tests I made, I cannot be sattisfied about the code I wrote, especially in the array comprehension part. As I have commented in the previous section, I try to deal with these expressions, but it hasn’t been satisfactory at all. The approaches I set haven’t been successful and the code generated for this section is pretty unreadable and gigantic.
Nevertheless I find that the rest of the code is well structured and nicely readable, since the monad definition has been very clear for me since the first sight. An example of this could be considered in the setters and getters for the monad, which I believe that have been easily defined, along the primitive functions definitions.
Regarding the warnings that are yielded by the compiler I have to say that I overlapped pattern matching in the code on purpose in order to return errors while interpreting, so the program can be easier to understand for the user who is running it.

3 Generic Replicated Server Library

3.1 Overview

Code for this question can be found in the src/alzheimer folder and in section 1.3 of the appendix.

Essentially, I have used two concepts which we have learned in the course during the Erlang part to program this question.
The first one is the “Supervisor and Worker” pattern used in robust systems. In the code it is easy to see that the Supervisor will be the Server Coordinator meanwhile the Replicas will act as Workers. Roughly, this pattern will allow the client to communicate with the Coordinator, who will resend the request to the Replicas, who will perform an operation (Reading or Writing). After the operation has finished (either yielding an exception or returning successfully) the Replica will send a return message to the client as well as it will send an event to the Coordinator saying whether the operation was ok or not. Depending on this result, the Coordinator will make a decision which could turn out to stop all the Replicas.
The second concept I have used is the gen_fsm behaviour, which has to do with the OTP Erlang Framework. This will make our code much simpler and easier to program and understand. I decided that both, Replicas and the Coordinator should use this behaviour since they will have different states (Coordinator with reading replicas, Coordinator with a writing replica, Replica stopped….). Because of this I joined these two concepts in order to create the Generic Replicated Server Library requested, which consists of two files: gen_replicated.erl (API as well as Coordinator implementation) and replica.erl (Implementation of a single Replica server).
I have structured the code of both files in different sections (API, states, gen_fsm callbacks and internal functions) so programming is easier and the resulting code is easy to read and maintain.
To end with, I have to say that all the messages and events passed around in both files are wrapped in requests that the two files can use. Needless to say that this implementation is hidden to the user, which can only use the functions requested in the exam paper. However, there are a couple of functions that have to be visible to replica.erl (finishReading/3 and finishWriting/2) so I exported them as well.
What is more, replica also offers a small API as the Coordinator has to interact with that file.

3.2 Coordinator design

As I have mentioned before, the Coordinator works as a Supervisor of the server library and it is implementation has been done under gen_fsm
The start function in the API starts the Coordinator finite machine, so its state is defined to ‘wait’, generating a initial state (Mod:init()) and starting the Replicas, using the API function exported by replica.erl with the map function, so we can track all their process id’s.
When a reading request is sent to the Coordinator, then an event is sent to it, which will be processed, unless the Coordinator is currently writing (Looking carefully at the states of gen_replicated we can see that all the client requests will have a response), sending the request to the first available replica (Using again the Replica API) and keeping track that the Replica used is now reading. This will make much easier to detect which Replicas are working just in case the server has to be stopped.
The states have been designed in such a way that reading operations can be done concurrently. Meanwhile, writing operations have to wait until the Coordinator returns to the wait state: first, the Coordinator sends the writing request to a single Replica and then the Coordinator waits until the Replica finishes its job.

The coordinator knows when a Replica finishes a job when it uses the Coordinator functions finishReading and finishWriting. This will generate an event updating the Coordinator state. There could be more changes: the data stored by the Coordinator could change (because the writing operation did it) or even more, the whole server could be stopped (Although finishReading(Pid, _ReplicaPid, stop) and finishWriting(Pid, stop) declarations might be redundant I believe that they both make the program more readable).
If the server is stopped then all the replicas are notified to stop (through stopReplica/1) and the Coordinator moved to a stopped state, where it cannot perform any operation (All the requests are discarded).
Regarding the gen_fsm callbacks, I use default values as returning values of these functions as they don't affect the library (I only change the code for terminate/3, where I added the atom coordinator_terminated_ok)

3.3 Replica design

Each one of the Replicas performs as a Worker for the Coordinator. At first, I was wondering about doing a big loop function for this server, but when I realized that exceptions were about to show up, then I changed my mind to gen_fsm. Although in this file I only use the states waiting and working (I don't even differentiate between reading Replicas and writing Replicas, as we did with the map and reduce workers in assignment number 5, which consisted of implementing a map and reduce framework using gen_fsm, Supervisors and Workers, because I feel that there wasn't a need to do so), the data stored in each of the states will be a key factor when the Coordinator tells a Replica to stop. This is because the Replica fsm will be stopped (stopReplica stops the fsm) so the terminate/3 function in the gen_fsm callbacks block will be called and it will alert the client that the current reading or writing operation was aborted. As it is clear to see, the rest of the callbacks are set to default as in the Coordinator file, since we don't need them.
Last but not least, when the Replica handles the possible exceptions and results from the handle_read and handle_write functions, it sends the right reply to the client and notifies the Supervisor (i.e, the Coordinator) about what happened.

3.4 Testing Generic Replicated Server Library

The instructions to run the 3 tests I've made are included in section 2.3 of the appendix. The test file can be found in src/alzheimer folder and in the mentioned section of the appendix.
According to one of the examples made during the lectures regarding a phonebook (The name of the files used are named in the test file as a comment), I picked some of the code and I modified it to implement the init/0, handle_read/2 and handle_write/2 functions used in the Generic Replicated Server Library.
The three functions implemented return different values as well as exceptions are thrown, so every part of the library I have created could be tested, analysing that everything goes ok. Many of the returns values may not make sense (failing to find a person makes the server to stop…), but this is intentional, since we are trying different return values and exceptions to test the library.

For example, in the first test, it is checked that adding people (making the Coordinator internal data state to update) performs well. It also reads ok (finding operation goes fine) and the stop request is well handled by the coordinator (when trying to find a non-existing person "Person4"). Finally, it is checked that is impossible to keep handling requests after the server is stopped.
The second tests checks that exceptions are controlled in the right way (see that updating a person that doesn't exist in the dictionary throws an exception) and that removing and updating people from the phonebook is correct too.
The third test finally checks that the stop result produced by calling handle_read has the desired effect, since it makes the Coordinator to stop.

3.5 Assesment

I think that the code I have handed for this section works correctly, according to the tests I have made (It would only need to be tested concurrently). Using gen_fsm and the Supervisor-Worker pattern makes the code simpler and easy to escalate (For example, we could restart the Coordinator with a request or Replicas could have more states and more functionality) and maintain. What is more, there are no possible deadlocks since all the states are well defined when they have an incoming event, there exists only communication between the Coordinator and the Replicas, and most of this communications is asynchronous (there is only a couple of synchronous communications when setting the working state for the Replicas that cannot produce deadlocks).
Finally I would like to mention that the client always have an answer of what is happening, either by a reply of the Replica or the Coordinator (There are no Replicas available to work, the server is stopped, there is a Replica writing, cannot write because there is at least one reading operation running….) which is rather user friendly.

4 AlzheimerDB

4.1 Overview

Code for this question can be found in the src/alzheimer folder and in section 1.4 of the appendix.
Since I programmed the Generic Replicated Server Library in the previous question, it has been much easier to use it in order to program the AlzheimerDB. This is because I would only need to take care about defining the three callback functions and format their return values correctly so they can be used by the library coded in the previous question.
Coding handle_read/2 and handle_write/2 can be done easily following the instructions from the exam paper. Then, if I provided a correct data structure for AlzheimerDB everything will be done, since the queries would be reading functions performed concurrently by the Replicas and the upsert functions would be writing functions performed (non-concurrently along

another writing or reading operations) by Replicas too.
Notice that I have defined a default number of Replicas for the AlzheimerDB on top of the file. This number should be modified depending on the estimated size of the database.

4.2 Data structure for the Database

The data structure chosen has been a simple one: an erlang dictionary, in which a row will be considered as a tuple of a key and value. The value is again a tuple with the rest of the values of the row.
One of the downsides of this implementation is that a conversion to lists is required every time a query is called. For this reason, maybe I could have considered a tree implementation, but the students were warned that the data structure was beyond the scope of the exam so I decided not to change it.

4.3 Testing AlzheimerDB

The test file can be found in the src/alzheimer folder and in section 2.4 of the appendix.
Instructions to run the tests are located in that section of the appendix as well.
Testing Alzheimer has been done in a similar way to the previous Erlang question.
Instructions to run the tests for this question can be found in the correspondent appendix.
I implemented the P/1 and F/1 functions needed for the Alzheimer database in such way that all the possible return values and exceptions will show up (and again I have to warn that this implementation is aimed to test AlzheimerDB, it doesn't have a real and useful intention). This time, Erlang pattern matching has been a successful tool to do it in the upsertFunction/1.
In the first test, two people are added to the DB and a query is sent to it. The result is an empty list (No rows returned true for the queryFunction/1, namely, P/1 in alzheimer.erl).
Then another person is added to the DB, the query is made again, and that last row is replied to the client since it returns true for the queryFunction/1.
Test number 2 inserts two people in the DB but then, a query is made, which produces an exception that is correctly caught and sent to the Client.
The last test checks that exceptions thrown by the upserFunction2/1 (i.e. F/1 in alzheimer.erl) are correctly caught and sent back to the Client as well.

4.4 Assesment

I believe that after testing it several times, the code is correct and works properly (It would only need to be tested concurrently). The fact that the Generic Replicated Server Library has been coded before has been a relief, because it has saved me a lot of time and effort to make this nice in-memory database.
There is not many code in the alzheimer.erl file but is good structured and there are some internal functions to make the file nicer to read.
I honestly think that this database can be easily maintained and escalated as I have mentioned previously with the library in Question 3.

Appendices

1 Code

1.1 SubScript Parser

1.1.1 SubsParser.hs

```
module SubsParser (
  ParseError(..),
  parseString,
  parseFile
) where

import SubsAst
import Control.Applicative hiding (Const)
import Data.Char
-- I'll use munch and munch1 defined in this document (I assume that
it would be nicer
-- that they should be included in SimpleParse.hs but I prefer not to
modify this library since is the original I took from the lectures)
import SimpleParse hiding (munch1, munch)

{-

--------------------
--------------------
Original Grammar (v1.3)
This parser is running under version 1.3 of the grammar provided in
the exam paper

Program ::= Stms
Stms ::= e | Stm ';' Stms
Stm ::= 'var' Ident AssignOpt | Expr
AssignOpt ::= e | '=' Expr1
Expr ::= Expr ',' Expr | Expr1
Expr1 ::= Number
        | String
        | 'true'
        | 'false'
        | 'undefined'
        | Expr1 '+' Expr1
        | Expr1 '-' Expr1
        | Expr1 '*' Expr1
        | Expr1 '%' Expr1
        | Expr1 '<' Expr1
```

```
        | Expr1 '===' Expr1
        | Ident AfterIdent
        | '[' Exprs ']'
        | '[' 'for' '(' Ident 'of' Expr1 ')' ArrayCompr Expr1 ']'
        | '(' Expr ')'

AfterIdent ::= e
            | '=' Expr1
            | FunCall
FunCall ::= '.' Ident FunCall
            | '(' Exprs ')'
Exprs ::= e
         | Expr1 CommaExprs
CommaExprs ::= e
            | ',' Expr1 CommaExprs
ArrayCompr ::= e
            | 'if' '(' Expr1 ')' ArrayCompr
            | 'for' '(' Ident 'of' Expr1 ')' ArrayCompr
```

Changed Grammar

With this new grammar I avoid left recursion as well as I provide each operator the right priority

This change will make the parser easier (Maybe I could have considered the function sepBy instead)

```
Stms ::= e | Stm Stms
Stm ::= 'var' Ident AssignOpt ';'
        | Expr ';'
```

I call Terminal expressions to all that expressions that don't make the parser go recursively

```
TerminalExpr ::=  Number
                | String
                | 'true'
                | 'false'
                | 'undefined'
                | Ident
                | Ident = FunCall
                | '[' Exprs ']'
                | '[' 'for' '(' Ident 'of' Expr ')' ArrayCompr Expr
']'
                | '(' Expr ')'
```

Adapted grammar in order to give each operator its priority. Notice how expressions from Topt4 could go to Topt2 and get an Epsilon. This is valid since we parse an expression from Topt5

```
Expr ::= Expr1 ',' Expr | Expr1
Expr1 ::= Topt5
Topt5 ::= Ident '=' Topt5 | Topt Topt4
Topt4 ::= '===' Topt Topt4 | Topt3
Topt3 ::= '<' Topt Topt3 | Topt2
Topt2 ::= '+' Topt Topt2 | '-' Topt Topt2 | e
Topt ::= TerminalExpr T
T ::= '*' TerminalExpr T | '%' TerminalExpr T | e
```

```
-}


-- Taken from SalsaInterp.hs (Assignment 2)
data ParseError = AmbiguousParse
                | NoParse
              deriving (Show, Eq)

-- List of reserved words
reservedWords :: [Ident]
reservedWords = ["true", "false", "undefined", "for", "of", "if"]

-- Special characters not valid in a String (Disallow characters to be
part of a Subscript String)
specChar :: String
specChar = "'"

-- Taken from Salsa SalsaInterp.hs (Assignment 2)
-- I remove the spaces from the end of the input so our parser would
parse correctly
-- Otherwise, it won't do it, since it doesn't remove spaces after the
last ';'
parseString :: String -> Either ParseError Program
parseString s = case parseEof parseProgram (trim s) of
                  [] -> Left  NoParse
                  [(p,_)] -> Right p
                  _ -> Left AmbiguousParse

-- Taken from
https://en.wikipedia.org/wiki/Trimming_(computer_programming)#Haskell
trim :: String -> String
trim = f . f
    where f = reverse . dropWhile isSpace

parseFile :: FilePath -> IO (Either ParseError Program)
parseFile path = fmap parseString $ readFile path


parseProgram :: Parser Program
parseProgram = do
               l <- parseStms
               return $ Prog l

-- Recall that the program might be empty so I use munch
parseStms :: Parser [Stm]
parseStms = munch parseStm


parseStm :: Parser Stm
parseStm = (do
            symbol "var"
            many1 space
            ident <- parseIdent
            assigned <- parseAssignOpt
            symbol ";"
            return $ VarDecl ident assigned)
            <|>
            (do
             expr <- parseExpr
             symbol ";"
             return $ ExprAsStm expr)
```

```
parseTerminalExpr :: Parser Expr
parseTerminalExpr = parseNumberExpr
                    <|>
                    parseStringExpr
                    <|>
                    (do
                     symbol "true"
                     return TrueConst)
                    <|>
                    (do
                     symbol "false"
                     return FalseConst)
                    <|>
                    (do
                     symbol "undefined"
                     return Undefined)
                     <|>
                     (do
                      ident <- token $ parseIdent
                      return $ Var ident
                     )
                     <|>
                     (do
                       ident <- token $ parseIdent
                       parseFunCall ident
                     )
                    <|>
                    (do
                     symbol "["
                     lExprs <- parseExprs
                     symbol "]"
                     return $ Array lExprs)
                    <|>
                    (do
                     symbol "["
                     symbol "for"
                     symbol "("
                     ident <- token $ parseIdent
                     symbol "of"
                     expr <- parseExpr1
                     symbol ")"
                     arrCompr <- parseArrayCompr
                     expr' <- parseExpr1
                     symbol "]"
                     return $ Compr (ident, expr, arrCompr) expr'
                     )
                    <|>
                    (do
                     symbol "("
                     expr <- parseExpr
                     symbol ")"
                     return expr)


parseAssignOpt :: Parser (Maybe Expr)
parseAssignOpt = return Nothing
                 <|>
                 (do
                  symbol "="
```

```
                  expr <- parseExpr1
                  return $ Just expr)


parseExpr :: Parser Expr
parseExpr = chainl1 parseExpr1 ( symbol "," >> return Comma)


parseExpr1 :: Parser Expr
parseExpr1 = parseTopt5


parseTopt5 :: Parser Expr
parseTopt5  = (do
                ident <- token $ parseIdent
                symbol "="
                expr <- parseTopt5
                return $ Assign ident expr
               )
                <|>
               (do
                expr <- parseTopt
                parseTopt4 expr
                )

parseTopt4 :: Expr -> Parser Expr
parseTopt4 expr = (do
                   symbol "==="
                   expr' <- parseTopt
                   expr'' <- parseTopt4 expr'
                   return $ Call "===" [expr, expr'']
                    )
                  <|>
                  parseTopt3 expr


parseTopt3 :: Expr -> Parser Expr
parseTopt3 expr = (do
                   symbol "<"
                   expr' <- parseTopt
                   expr'' <- parseTopt4 expr'
                   return $ Call "<" [expr, expr'']
                    )
                 <|>
                   parseTopt2 expr

parseTopt2 :: Expr -> Parser Expr
parseTopt2 expr = return expr
                <|>
                (do
                   symbol "+"
                   expr' <- parseTopt
                   expr'' <- parseTopt4 expr'
                   return $ Call "+" [expr, expr'']
                    )
                <|>
                (do
                   symbol "-"
                   expr' <- parseTopt
                   expr'' <- parseTopt4 expr'
```

```
                    return $ Call "-" [expr, expr'']
                     )

parseTopt :: Parser Expr
parseTopt = do expr <- parseTerminalExpr
               parseT expr

parseT :: Expr -> Parser Expr
parseT expr = return expr
            <|>
            (do
             symbol "*"
             expr' <- parseTerminalExpr
             expr'' <- parseT expr'
             return $ Call "*" [expr, expr''])
            <|>
            (do
             symbol "%"
             expr' <- parseTerminalExpr
             expr'' <- parseT expr'
             return $ Call "%" [expr, expr''])




parseFunCall :: Ident -> Parser Expr
parseFunCall ident = (do
                      char '.'
                      ident' <- parseIdent

                      symbol "("
                      lExprs <- parseExprs
                      symbol ")"
                      return $ Call (ident ++ "." ++ ident') lExprs
                      )
                     <|>
                     (do
                      symbol "."
                      ident' <- parseIdent
                      parseFunCall (ident ++ "." ++ ident') )


parseExprs :: Parser [Expr]
parseExprs = return []
             <|>
             (do
              expr <- parseExpr1
              lExps <- parseCommaExprs
              return $ [expr] ++ lExps)


parseCommaExprs :: Parser [Expr]
parseCommaExprs = return []
                  <|>
                  (do
                   symbol ","
                   expr <- parseExpr1
                   lExps <- parseCommaExprs
                   return $ [expr] ++ lExps)
```

```
parseArrayCompr :: Parser (Maybe ArrayCompr)
parseArrayCompr = return Nothing
                  <|>
                  (do
                   symbol "if"
                   symbol "("
                   expr <- parseExpr1
                   symbol ")"
                   arrCompr <- parseArrayCompr
                   return $ Just (ArrayIf expr arrCompr) )
                  <|>
                  (do
                   symbol "for"
                   symbol "("
                   ident <- token $ parseIdent
                   symbol "of"
                   expr <- parseExpr1
                   symbol ")"
                   arrCompr <- parseArrayCompr
                   return $ Just (ArrayForCompr (ident, expr,
arrCompr) ) )

-- Bit complicated but I avoid looking up the '-' character with
lookup function (Had to split the if - else cases because of do
notation)
parseNumberExpr :: Parser Expr
parseNumberExpr = token $ do
                           y <- item
                           if y == '-' then getMinusInt
                           else getPositiveInt y


getMinusInt :: Parser Expr
getMinusInt = do
                x <- token $ munch1 digit
                -- x has length at least 1 due to munch1 definition
                if length x <= 8 then return $ Number (- read x)
                else reject

getPositiveInt :: Char -> Parser Expr
getPositiveInt y = do
                    if isDigit y then getPositiveInt2 y
                    else reject

getPositiveInt2 :: Char -> Parser Expr
getPositiveInt2 y = do
                         x <- munch digit
                         -- [y]++x has length at least 1 (y has length
1)
                         if length ([y]++x) <= 8 then return $ Number $
read ([y]++x)
                         else reject


parseStringExpr :: Parser Expr
parseStringExpr = do
                  symbol "'"
                  s <- munch $ satisfy isChar
                  symbol "'"
                  return $ String s
```

```
isChar :: Char -> Bool
isChar c = if c `elem` specChar then False
           else True

-- The idea of doing these functions is taken from assignment 2 as
well
parseIdent :: Parser Ident
parseIdent = do
               c <- satisfy isIdentBegin
               c' <- munch $ satisfy isIdentChar
               if isReservedWord $ [c] ++ c' then reject
               else return $ [c] ++ c'


isIdentBegin :: Char -> Bool
isIdentBegin c = if isAlpha c || isUnderscore c then True
             else False

isUnderscore :: Char -> Bool
isUnderscore c = if c == '_' then True
                 else False

isIdentChar :: Char -> Bool
isIdentChar c = if isAlpha c || isDigit c || isUnderscore c then True
             else False

isReservedWord :: String -> Bool
isReservedWord s = if s `elem` reservedWords then True
                   else False

-- Functions taken from Expr.hs (Made by TA Maya Saietz in a
Laboratory session)
-- Could be nicer to include them in an improved version of
SimpleParse (Recall that I've used the original SimpleParse file
provided in the lectures)

munch :: Parser a -> Parser [a]
munch p = munch1 p <++ return []

munch1 :: Parser a -> Parser [a]
munch1 p = do
    x  <- p
    xs <- munch p
    return $ x : xs

digit :: Parser Char
digit = satisfy isDigit
```

1.2 SubScript Interpreter

1.2.1 SubsInterpreter.hs

```
module SubsInterpreter
       ( runProg
       , Error (..)
       , Value(..)
       )
       where

import SubsAst

-- You might need the following imports
import Control.Monad
import qualified Data.Map as Map
import Data.Map(Map)
import Data.Char -- Used to promote digits to chars

-- | A value is either an integer, the special constant undefined,
--   true, false, a string, or an array of values.
-- Expressions are evaluated to values.
data Value = IntVal Int
           | UndefinedVal
           | TrueVal | FalseVal
           | StringVal String
           | ArrayVal [Value]
           deriving (Eq, Show)

-- ^ 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'.

-- I left this constructor because it will be simpler to propagate the
errors through the program
data Error = Error String
             deriving (Show, Eq)

type Env = Map Ident Value
type Primitive = [Value] -> SubsM Value
type PEnv = Map FunName Primitive
type Context = (Env, PEnv)

initialEnv :: Env
initialEnv = Map.empty
initialPEnv :: PEnv
initialPEnv = Map.fromList [ ("===", primEq)
                           , ("<", primLT)
                           , ("+", primPlus)
                           , ("*", primMul)
                           , ("-", primMinus)
                           , ("%", primMod)
                           , ("Array.new", arrayNew)
                           ]
```

```haskell
initialContext :: Context
initialContext = (Map.empty, initialPEnv)



newtype SubsM a = SubsM {runSubsM :: Context -> Either Error (a, Env)}

-- Instatiated both type classes as it was provided in the lecture
slides
instance Functor SubsM where
  fmap f xs = xs >>= return . f

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

{-
I use the do syntax for the >>= operator because it is more readable.
Nevertheless, an equivalent implementation would be:
    f >>= m = SubsM $ \ (env, pEnv) ->
      case runSubsM f (env, pEnv) of
           Left $ Error s -> Left $ Error s
           Right (a, env') -> case runSubsM (m a) (env', pEnv) of
                                Left $ Error s -> Left $ Error s
                                Right v -> Right v


-}

instance Monad SubsM where
  return x = SubsM $ \ (env, _) -> Right (x, env)
  f >>= m = SubsM $ \ (env, pEnv) ->
                do (a, env') <- runSubsM f (env, pEnv)
                   (a', env'') <- runSubsM (m a) (env', pEnv)
                   return $ (a', env'')

  fail s = SubsM $ \ _ -> Left $ Error s


runProg :: Program -> Either Error Env
runProg prog = getResult (recursiveRun prog)

-- First I set the initial environment. Then I interpret the whole
program and finally I get the resulting environment
recursiveRun :: Program -> SubsM Env
recursiveRun p = do
                  putEnv initialEnv
                  program p
                  env <- getEnv
                  return $ env

program :: Program -> SubsM ()
program (Prog []) = return ()
program (Prog (x:xs)) = do
                         stm x
                         program $ Prog xs


-- For variables declared without a value I give them an "undefined"
value
```

```
stm :: Stm -> SubsM ()
stm (VarDecl ident Nothing) = do
                                modify (insertEnv ident UndefinedVal)
                                return ()

stm (VarDecl ident (Just expr) ) = do
                                    val <- evalExpr expr
                                    modify(insertEnv ident val)
                                    return ()
stm (ExprAsStm expr) = do
                        evalExpr expr
                        return ()

stm s = fail $ "The statement " ++ show s ++ " couldn't be interpreted
(Doesn't fit pattern matching for SubScript)"

arrayNew :: Primitive
arrayNew [IntVal n] | n > 0 = return $ ArrayVal(take n $ repeat
UndefinedVal)
arrayNew _ = fail ("Array.new called with wrong number of arguments")

-- I've suppossed that two undefinedVal are equal ( I recalled that
nil is equivalent to nil in other programming languages )
primEq :: Primitive
primEq [IntVal n, IntVal n' ] = if n == n' then return $ TrueVal
                                else return $ FalseVal
primEq [UndefinedVal, UndefinedVal] = return $ TrueVal
primEq [TrueVal, FalseVal] = return $ FalseVal
primEq [FalseVal, TrueVal] = return $ FalseVal
primEq [TrueVal, TrueVal] = return $ TrueVal
primEq [FalseVal, FalseVal] = return $ TrueVal
primEq [StringVal s, StringVal s'] = if s == s' then return $ TrueVal
                                     else return $ FalseVal
primEq [ArrayVal a1, ArrayVal a2] = if a1 == a2 then return $ TrueVal
                                    else return $ FalseVal
primEq _ = fail (" '===' called with wrong number of arguments ")

primLT :: Primitive
primLT [IntVal n, IntVal n'] = if n < n' then return $ TrueVal
                               else return $ FalseVal
primLT [StringVal s, StringVal s'] = if s < s' then return $ TrueVal
                                     else return $ FalseVal
primLT _ = fail (" '<' called with wrong number of arguments ")


primPlus :: Primitive
primPlus [IntVal n, IntVal n' ] = return $ IntVal $ n+n'
primPlus [IntVal n, StringVal s] = return $ StringVal $ (intToString
n) ++ s
primPlus [StringVal s, IntVal n] = primPlus [IntVal n, StringVal s]
primPlus [StringVal s, StringVal s'] = return $ StringVal $ s++s'
primPlus _ = fail (" '+' arguments must be either integers or one
string and one number ")

-- Convert each char of the string to a digit (I consider 0 as a
different case)
intToString :: Int -> String
intToString 0 = [intToDigit 0]
intToString n = intToStringRec n []

intToStringRec :: Int -> String -> String
```

```
intToStringRec 0 s = s
intToStringRec n s = intToStringRec (n `div` 10) ( [ intToDigit $ n
`mod` 10 ] ++ s )

primMul :: Primitive
primMul [IntVal n, IntVal n' ] = return $ IntVal $ n*n'
primMul _ = fail (" '*' arguments must be integers")

primMinus :: Primitive
primMinus [IntVal n, IntVal n' ] = return $ IntVal $ n-n'
primMinus _ = fail(" '-' arguments must be integers")

primMod :: Primitive
primMod [IntVal n, IntVal n' ] = return $ IntVal $ n `mod` n'
primMod _ = fail(" '%' arguments must be integers")

-- Get the result from the monad using the initial context
getResult :: SubsM a -> Either Error a
getResult m = case runSubsM m initialContext of
                Left e -> Left e
                Right (a,_) -> Right a

-- Getter for the current environment
getEnv :: SubsM Env
getEnv = SubsM $ \ (env, _) -> Right (env, env)

-- Getter for the environment of the build-in functions
getPEnv :: SubsM PEnv
getPEnv = SubsM $ \ (env, pEnv) -> Right (pEnv, env)

-- Setter for the current environment
putEnv :: Env -> SubsM ()
putEnv e = SubsM $ \ _ -> Right ((), e)

-- Modifier of the current environment
modify :: (Env -> Env) -> SubsM ()
modify f = do
             env <- getEnv
             putEnv (f env)

-- Inserting the identifier and the value of a variable returning a
function which takes an environment and returns the environment after
having inserted the name and the value
insertEnv :: Ident -> Value -> (Env -> Env)
insertEnv name val = Map.insert name val

-- Update the value of a variable already defined in the current
environment
updateEnv :: Ident -> Value -> SubsM ()
updateEnv name val = do
                       env <- getEnv
                       case Map.lookup name env of
                         Nothing -> fail $ "Variable " ++ name ++ " not
declared"
                         Just _ -> putEnv (Map.insert name val env)

-- Getter for the value of a given variable
getVar :: Ident -> SubsM Value
getVar name = do
                env <- getEnv
                case Map.lookup name env of
```

```
                    Nothing -> fail $ "Variable " ++ name ++ " not
declared"
                    Just v -> return v

-- Getter for primitive functions
getFunction :: FunName -> SubsM Primitive
getFunction name = do
                    pEnv <- getPEnv
                    case Map.lookup name pEnv of
                        Nothing -> fail $ "Primitive function " ++
name ++ " doesn't exist"
                        Just p -> return p


evalExpr :: Expr -> SubsM Value
evalExpr (Number n) = return $ IntVal n
evalExpr (String s) = return $ StringVal s

-- I stop if there is a expression inside the Array which couldn't be
interpreted
evalExpr (Array lExpr) = do
                            env <- getEnv
                            case mapExpr lExpr env of
                                Right v -> return $ ArrayVal v
                                Left s -> fail s



evalExpr Undefined = return $ UndefinedVal
evalExpr TrueConst =  return $ TrueVal
evalExpr FalseConst = return $ FalseVal
evalExpr (Var ident) = do
                        v <- getVar ident
                        return v




{-
    For evaluating array comprehension expressions I send all the
information to an auxiliar function which keeps track of the
environment used in every nested loop

    I only left as undefined the array comprehensions with an if
statement which holds another for loop. The cause is that I didn't
feel comfortable with the design I've approached.
    When I started to code this last case I realized that I couldn't
use any of the functions defined previously in evalExprAux and that I
had to consider more cases.
    This would make the code to grow exponentially so I assume that my
approach for interpreting array comprehensions is not the right one.
However, I haven't been able to get
    a different and easier one. There also a problem with the scopes
of the variables defined in the for loops

-}

evalExpr (Compr (ident, expr, Nothing) expr2) =
                do
                env <- getEnv
                evalExprAux (Compr (ident, expr, Nothing) expr2) env
```

```
evalExpr (Compr (ident, expr, Just (ArrayForCompr a ) ) expr2 ) = do

env <- getEnv

evalExprAux (Compr (ident, expr, Just (ArrayForCompr a)) expr2) env

evalExpr (Compr (ident, expr, Just (ArrayIf expr' Nothing) ) expr2 ) =
do

env <- getEnv

evalExprAux (Compr (ident, expr, Just (ArrayIf expr' Nothing ) ) expr2
) env

evalExpr (Compr (ident, expr, Just (ArrayIf expr' (Just arrComp) ))
expr2) = do

env <- getEnv

evalExprAux (Compr (ident, expr, Just (ArrayIf expr' (Just arrComp) ))
expr2) env



-- I call the mapExpr function to get the values of the Expression
list
evalExpr (Call funName lExpr) = do
                                prim <- getFunction funName
                                env <- getEnv
                                case mapExpr lExpr env of
                                    Right v -> prim v
                                    Left s -> fail s



evalExpr (Assign ident expr) = do
                                val <- evalExpr expr
                                updateEnv ident val
                                return $ val

evalExpr (Comma expr expr2) = do
                               evalExpr expr
                               evalExpr expr2

evalExpr expr = fail $ "Expression " ++ show expr ++ " couldn't be
interpreted (Doesn't fit pattern matching for SubScript)"

-- Maps a list of expressions to a list of values evaluating each one
of the elements of the first list. Notice that I have to call runSubsM
to get the value from
-- the monad with the current environment

mapExpr :: [Expr] -> Env -> Either String [Value]
mapExpr lExpr env = case x of
                  Left (Error s) -> Left s
                  Right (v,_) -> Right v
    where x = runSubsM (mapExpr2 lExpr) (env, initialPEnv)
```

```haskell
-- Recursive function of mapExpr
mapExpr2 :: [Expr] -> SubsM [Value]
mapExpr2 [] = return $ []
mapExpr2 (x:xs) = do
                  v <- evalExpr x
                  lVal <- mapExpr2 xs
                  return $ [v] ++ lVal

-- First of all, the auxiliar functions detect that the syntax of expr
in the array is correct, then the results will be treated different
since the strings require a bit of handling
evalExprAux :: Expr -> Env -> SubsM Value
evalExprAux (Compr (ident, expr, Nothing) expr2 ) env =
do
v <- evalExpr expr

case v of

StringVal s ->  stringCase env expr2 ident s

ArrayVal lVal -> arrayCase env expr2 ident lVal

_ -> fail $ "Expression " ++ show expr ++ " not valid in Array syntax"

evalExprAux (Compr (ident, expr, Just (ArrayForCompr a)) expr2) env =
do

v <- evalExpr expr

case v of

StringVal s -> stringCase2 env expr2 ident s a

ArrayVal lVal -> arrayCase2 env expr2 ident lVal a

_ -> fail $ "Expression " ++ show expr ++ " not valid in Array syntax"

evalExprAux (Compr (ident, expr, Just (ArrayIf expr' Nothing) ) expr2
) env = do

v <- evalExpr expr

case v of

StringVal s -> stringCase3 env expr2 ident s expr'

ArrayVal lVal -> arrayCase3 env expr2 ident lVal expr'

_ -> fail $ "Expression " ++ show expr ++ " not valid in Array syntax"

--evalExprAux (Compr (ident, expr, Just (ArrayIf expr' (Just arrComp)
)) expr2) env = undefined
-- Rather than leaving the previous function as undefined I prefered
to leave it as impossible to interpret (Assuming that I restricted the
SubScript language)
evalExprAux expr env = fail $ "The expression " ++ show expr ++ "
couldn't be interpreted with the following environment " ++ show env
```

```
-- In this case I have an extra parameter, namely, the expression used
for the if
arrayCase3 :: Env -> Expr -> Ident -> [Value] -> Expr -> SubsM Value
arrayCase3 env expr2 ident lVal expr' = do
                          v <- mapEvalExprAux2 ident lVal expr' expr2
                          putEnv env
                          return v


stringCase3 :: Env -> Expr -> Ident -> String -> Expr -> SubsM Value
stringCase3 env expr2 ident s expr' = do
           v <- mapEvalExprAux2 ident (toListStringVal s) expr' expr2
           putEnv env
           return v



mapEvalExprAux2 :: Ident -> [Value] -> Expr -> Expr -> SubsM Value
mapEvalExprAux2 _ [] _ _ = return $ ArrayVal []
mapEvalExprAux2 ident (x:xs) expr' expr2 = do

           modify (insertEnv ident x)
           e <- evalExpr expr'

           case e of

            TrueVal -> trueIf expr' expr2 ident xs

            FalseVal -> falseIf expr' expr2 ident xs

            _ -> fail $ "Expression " ++ show expr' ++ " not valid in
Array syntax"

-- If the evaluation of the if was true, then the expression is
evaluated and returned. Then, I keep on mapping environments
trueIf :: Expr -> Expr -> Ident -> [Value] ->  SubsM Value
trueIf expr' expr2 ident xs  = do
           e' <- evalExpr expr2
           ArrayVal l2 <- mapEvalExprAux2 ident xs expr' expr2
           return $ ArrayVal $ [e'] ++ l2

-- If the evaluation of the if was false, then the expression is not
evaluated but we keep on mapping environments
falseIf :: Expr -> Expr -> Ident -> [Value] -> SubsM Value
falseIf expr' expr2 ident xs = do
          ArrayVal l2 <- mapEvalExprAux2 ident xs expr' expr2
          return $ ArrayVal l2




-- These cases are pretty similar to the previous one (Terminal "for")
but here I had to add an ArrayFor as a parameter
arrayCase2 :: Env -> Expr -> Ident -> [Value] -> ArrayFor -> SubsM
Value
arrayCase2 env expr2 ident lVal a = do
```

```
            mapEvalExprAux ( mapEnv env (map (insertEnv ident) lVal) )
expr2 a env


stringCase2 :: Env -> Expr -> Ident -> String -> ArrayFor -> SubsM
Value
stringCase2 env expr2 ident s a = do
             mapEvalExprAux  ( mapEnv env (map (insertEnv ident)
(toListStringVal s)) ) expr2 a env


-- See how in the first case I restore the previous environment
mapEvalExprAux :: [Env] -> Expr ->  ArrayFor -> Env -> SubsM Value
mapEvalExprAux [] _ _ env = do
                            putEnv env
                            return $ ArrayVal []

-- In these case I glue the arrays calling evalExprAux (which could
have been called before) and mapEval recursively
mapEvalExprAux (x:xs) expr2 a env= do
                          putEnv x
                          ArrayVal l1 <- evalExprAux (Compr a expr2) x
                          ArrayVal l2 <- mapEvalExprAux xs expr2 a env
                          return $ ArrayVal $ l1++l2



-- If there is a terminal "for" expression then depending of the value
to iterate (either string or array) I'll have to insert
-- different values in the "temporary environment" used in
"appendValues". Eventually, the previous environment is restored
arrayCase :: Env -> Expr -> Ident -> [Value] -> SubsM Value
arrayCase env expr2 ident lVal = do
                             l1 <- appendValues ident lVal expr2
                             putEnv env
                             return $ ArrayVal l1

stringCase :: Env -> Expr -> Ident -> String -> SubsM Value
stringCase env expr2 ident s = do
                             l1 <- appendValues ident
(toListStringVal s) expr2
                             putEnv env
                             return $ ArrayVal l1

-- Glue together the values returned after evaluating the expression
with the temporary environments
appendValues :: Ident -> [Value] -> Expr -> SubsM [Value]
appendValues _ [] _ = return $ []
appendValues ident (x:xs) expr = do
                               modify (insertEnv ident x)
                               v <- evalExpr expr
                               lVal <- appendValues ident xs expr
                               return $ [v]++lVal


-- Convert a list of chars (String) in a list of characters seen as
Subs values (These characters are in fact a list of strings formed by
that only character)
toListStringVal :: String -> [Value]
toListStringVal [] = []
```

```
toListStringVal (x:xs) = [StringVal [x]] ++ toListStringVal xs

-- Create the temporary environments after the ident took the
different values of the string or the array
mapEnv :: Env -> [(Env -> Env)] -> [Env]
mapEnv _ [] = []
mapEnv env (x:xs) = [(x env)] ++ (mapEnv env xs)
```

1.3 Generic Replicated Server Library

1.3.1 gen_replicated.erl

```
-module(gen_replicated).
-export([start/2, stop/1, read/2, write/2, finishReading/3,
finishWriting/2]).


-behaviour(gen_fsm).

-import(replica, [startReplica/1, stopReplica/1, readReplica/2,
writeReplica/2 ]).

-export([init/1,wait/2, readingOp/2, writingOp/2, stopped/2,
handle_event/3, handle_sync_event/4, handle_info/3, terminate/3,
code_change/4 ]).

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

start(NumReplica, Mod) ->
    gen_fsm:start_link(gen_replicated, {Mod, NumReplica}, []).

% I send an event to stop the cordinator which will turn out to switch
down all of the replicas
stop(Server) ->
    gen_fsm:send_event(Server, stopCoord).

% I send an event with the Pid of the client. Notice that the message
received will be sent by a replica
read(Server, Req) ->
    Ref = self(),
    gen_fsm:send_event(Server, {read, {Ref, Req} }),
    receive
      Result -> Result
    end.

write(Server, Req) ->
    Ref = self(),
    gen_fsm:send_event(Server, {write, {Ref, Req} }),
    receive
       Result -> Result
    end.
```

```
% Using pattern matching, I differentiate cases when the replicas
finish their task
finishReading(Pid, _ReplicadPid, stop) ->
    stop(Pid);

finishReading(Pid, ReplicaPid, _) ->
    gen_fsm:send_event(Pid, {finishReading, ReplicaPid}).

finishWriting(Pid, stop) ->
    stop(Pid);

finishWriting(Pid, {updated, NewState} ) ->
    gen_fsm:send_event(Pid, {finishWriting, NewState});

finishWriting(Pid, _) ->
    gen_fsm:send_event(Pid, finishWriting).



%%%==========================================================================
======
%%%  gen_fsm states
%%%==========================================================================
======

% I keep the state produced by Mod:init() to update it and send it to
the replicas
init({Mod, NumReplica}) ->
    {ok, wait, {Mod:init(), startReplicas(Mod, NumReplica)}}.

% Stop all the replicas
wait (stopCoord , {_ModState, ListReplicas} ) ->
    lists:map( fun replica:stopReplica/1, ListReplicas),
    {next_state, stopped, []};

% Keep track of the replicas which are currently reading
wait( {read, Req}, {ModState, [Replica1 | RestReplicas]} ) ->
    replica:readReplica(Replica1, {{Req,Replica1,self()}, ModState}),
    { next_state, readingOp, {ModState, RestReplicas, [Replica1]} };

% Keep track of the replica which is currently writing (Notice that we
can only write if the fsm is in the wait state, i.e, there is no
current operations)
wait( {write, Req}, {ModState, [Replica1 | RestReplicas]}) ->
    replica:writeReplica(Replica1, {{Req,self()}, ModState}),
    { next_state, writingOp, {ModState, RestReplicas, Replica1} }.

% Cannot read and write concurrently
readingOp( {write, {Ref,_Req}}, State) ->
    async(Ref, "Cannot write in this moment. Reading on progress" ),
    { next_state, readingOp, State};

% Stop reading and not_reading replicas
readingOp( stopCoord, {_ModState,ListReplicas,ListReadingReplicas}) ->
    lists:map( fun replica:stopReplica/1, ListReplicas),
    lists:map( fun replica:stopReplica/1, ListReadingReplicas),
    { next_state, stopped, []};

% All the replicas are reading
readingOp( {read, {Ref,_Req}}, {ModState,[], ListReadingReplicas} ) ->
```

```
        async(Ref, "All the replica processes are reading data at this
moment. Wait until one of them finish"),
        { next_state, readingOp, {ModState, [], ListReadingReplicas}};

% One replica has finished reading. Could move to either wait or
reading state
readingOp( {finishReading, ReplicaPid}, {ModState, List,
ListReadingReplicas}) ->
        case lists:delete(ReplicaPid,ListReadingReplicas) of
            [] -> {next_state, wait, {ModState,[ReplicaPid | List]} };
           List2 -> {next_state, readingOp, {ModState, [ReplicaPid |
      List], List2 } }

        end;

% Allow to read concurrently
readingOp( {read, Req}, {ModState, [Replica1 | RestReplicas],
ListReadingReplicas}) ->
    replica:readReplica(Replica1, {{Req,Replica1,self()}, ModState}),
    { next_state, readingOp, {ModState, RestReplicas, [Replica1 |
ListReadingReplicas] }}.

% Stop the writing and the rest of replicas
writingOp( stopCoord, {_ModState, ListReplicas, WriterReplica} ) ->
    lists:map( fun replica:stopReplica/1, ListReplicas),
    replica:stopReplica(WriterReplica),
    { next_state, stopped, []};

% Cannot write and read concurrently
writingOp( {read, {Ref,_Req}}, State ) ->
   async(Ref, "Cannot perform reading operation. Writing on progress"
),
   { next_state, writingOp, State};

% The replica stopped writing, we may modify the state
writingOp(finishWriting, {ModState, List, ReplicaPid}) ->
    { next_state, wait, {ModState, [ReplicaPid | List]} };

writingOp({finishWriting, NewModState}, {_OldModState, List,
ReplicaPid}) ->
    { next_state, wait, {NewModState, [ReplicaPid | List]} }.

% Stopped state. Cannot do nothing
stopped( stopCoord, State) ->
   { next_state, stopped, State };

stopped( {_Event, {Ref,_Req}}, State) ->
    async(Ref, "Cannot handle request. Server is stopped" ),
    { next_state, stopped, State }.


%%%==========================================================================
======
%%%  gen_fsm callbacks
%%%==========================================================================
======

% These gen_fsm callbacks are set to the default values, since they
don't need them
code_change(_OldVsn, StateName, State, _Extra) ->
    {ok, StateName, State}.
```

```
handle_event(_Event, StateName, State) ->
    {next_state, StateName, State}.

handle_info(_Info, StateName, State) ->
    {next_state, StateName, State}.

handle_sync_event(_Event, _From, StateName, State) ->
    Reply = ok,
    {reply, Reply, StateName, State}.

% Return a value to check that the coordinator terminated correctly
terminate(_Reason, _State, _StateData) ->
    coordinator_terminated_ok.

%%%==================================================================
======
%%%  Internal Functions
%%%==================================================================
======

% Starting each of the replicas and joining all their Pid's in a List
startReplicas(Mod, NumReplica) ->
    case (NumReplica > 0) of
        true -> List = lists:map(fun replica:startReplica/1,
lists:duplicate(NumReplica, Mod)),
                getPid(List);
        false -> io:format("The number of replicas should be strictly
more than 0")
    end.


%% Taken from assignment 5. I filter the {ok, Pid} response in a first
list to get only the Pid's
getPid(List) ->
    lists:map( fun( {ok, Pid} )  -> Pid end, List).


 % Taken from the lecture slides. Non-blocking rpc
async(Pid, Request) ->
    Pid ! Request.
```

1.3.2 replica.erl

```
-module(replica).

-behaviour(gen_fsm).

-import(gen_replicated, [finishReading/3, finishWriting/2]).
-export([startReplica/1, stopReplica/1, readReplica/2,
writeReplica/2]).
-export([init/1, wait/3, working/2, handle_event/3,
handle_sync_event/4, handle_info/3, terminate/3, code_change/4]).

%%%==================================================================
======
%%%  API
```

```
%%%==========================================================================
======

startReplica(Mod) ->
    gen_fsm:start_link(replica, Mod, []).

% Make the replicas to either read or work. Notice that this is
similar to the Worker fsm used in assignment 5 but this time I don't
differentiate readerReplicas and writerReplicas
% I make the first event as sync to wait till the working state is set
in the replica fsm
readReplica(Pid, Req) ->
     gen_fsm:sync_send_event(Pid, {readReplica, Req}),
     gen_fsm:send_event(Pid, startWorkingReading).

writeReplica(Pid, Req) ->
     gen_fsm:sync_send_event(Pid, {writeReplica, Req}),
     gen_fsm:send_event(Pid, startWorkingWriting).

% Shutdown the replica fsm. Notice that we warn the client about this
through the terminate function (I send them an async message)since we
wave their Pid saved in the state
stopReplica(Pid) ->
     gen_fsm:stop(Pid).

%%%==========================================================================
======
%%%  gen_fsm states
%%%==========================================================================
======

init(Mod) ->
    {ok, wait, Mod}.

% Save all the data in order to work or to either yield an error to
the client or working with that
wait({readReplica, { {{Ref,Req},SelfPid,CoordPid}, ModState} }, _From,
Mod) ->
    {reply, ok, working, {Mod,Ref,SelfPid,Req,CoordPid,ModState}};

wait({writeReplica, {{{Ref, Req},CoordPid}, ModState}}, _From, Mod) ->
    {reply, ok, working, {Mod,Ref,Req,CoordPid,ModState}}.

% Using the try-catch block I assure that the client will get a
response eventually
working( startWorkingReading,
{Mod,ClientPid,SelfPid,Req,CoordPid,ModState}) ->
    try Mod:handle_read(Req, ModState) of
        {reply, Reply} -> async(ClientPid, Reply),
finishReading(CoordPid, SelfPid, reply);
        stop -> async(ClientPid, {'ABORTED', server_stopped}),
finishReading(CoordPid, SelfPid, stop)
    catch
        _ : Val -> async(ClientPid, {'ABORTED', exception, Val}),
finishReading(CoordPid, SelfPid, exception)
    end,
    { next_state, wait, Mod};

working( startWorkingWriting, {Mod,ClientPid,Req,CoordPid,ModState} )
->
   try Mod:handle_write(Req, ModState) of
```

```
        {noupdate, Reply } -> async(ClientPid,
Reply),finishWriting(CoordPid, noupdate );
        {updated, Reply, NewState} -> async(ClientPid, Reply),
finishWriting(CoordPid, {updated, NewState});
        stop -> async(ClientPid, {'ABORTED',
server_stopped}),finishWriting(CoordPid, stop)
    catch
        _ : Val -> async(ClientPid, {'ABORTED', exception, Val}),
finishWriting(CoordPid, exception)
    end,
    { next_state, wait, Mod}.



%%%==========================================================================
======
%%%  gen_fsm callbacks
%%%==========================================================================
======

% These gen_fsm callbacks are set to the default values, since for
some of them I don't require them
code_change(_OldVsn, StateName, State, _Extra) ->
    {ok, StateName, State}.


handle_event(_Event, StateName, State) ->
    {next_state, StateName, State}.

handle_info(_Info, StateName, State) ->
    {next_state, StateName, State}.

handle_sync_event(_Event, _From, StateName, State) ->
    Reply = ok,
    {reply, Reply, StateName, State}.

terminate(_Reason, wait, _StateData) ->
    replica_stopped_correctly;

terminate(_Reason, working, {_Mod, ClientPid, _SelfPid, _Req,
_CoordPid, _ModState}) ->
    async(ClientPid, {'ABORTED', server_stopped});

terminate(_Reason, working, {_Mod, ClientPid, _Req, _CoordPid,
_ModState}) ->
    async(ClientPid, {'ABORTED', server_stopped}).



%%%==========================================================================
======
%%%  Internal Functions
%%%==========================================================================
======

 % Taken from the lecture slides. Non-blocking rpc
async(Pid, Request) ->
    Pid ! Request.
```

1.4 AlzheimerDB

1.4.1 alzheimer.erl

```
-module(alzheimer).

-import(gen_replicated, [start/2, read/2, write/2]).

-export([start/0, upsert/3, query/2]).
-export([init/0, handle_read/2, handle_write/2]).

% Defining a constant will allow to change to number of replicas
depending on the size of the database
-define(NUMREPLICAS, 100).

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

% Using the gen_replicated.erl I have already done the database
becomes easier to implement (Queries are read operations for
% the replicas and inserting-updating operations are writing
operations for the writing replica)

start() ->
    gen_replicated:start(?NUMREPLICAS,alzheimer).

query(Aid, P) ->
    gen_replicated:read(Aid,P).

upsert(Aid, Id, F) ->
    gen_replicated:write(Aid, {Id,F}).


% I am going to implement a simple database with a dictionary. A row
consists of a tuple of a key and a value (Then calling the P function
is always correct).
% The key is going to be the Identifier used in the upsert function
(i.e, primary key for the database) and the value will be a tuple with
the rest of the values of the row
init() -> dict:new().

% See that if there is an exception in either getWritingResult or
getWritingResult2, it will be propagated
handle_write({Id,F}, Dict) ->
    case dict:find(Id,Dict) of
        error -> getWritingResult(F, Id, Dict);
        Val -> getWritingResult2(F, Id, Val, Dict)
    end.

% In this case I handle exceptions for the P function
handle_read(P, Dict) ->
        List = dict:to_list(Dict),
        try getResult(P,List) of
```

```
            Rows -> {reply, {ok, Rows} }
        catch
        _ : FailedRow -> { reply, {error, FailedRow} }
        end.


%% Internal functions

% Auxiliar function which formats the answer for handle_write or
rethrow again the exception.
getWritingResult(F,Id, Dict) ->
    try F({new, Id}) of
       {modify, NewData} -> {updated, {database_updated, NewData,
added_to, Id}, dict:store(Id,NewData,Dict)};
       ignore -> {noupdate, ignore}
    catch
    _ : Val -> throw(Val)
    end.

% Auxiliar function similar to getWritingResult which call F with
another arguments
getWritingResult2(F, Id, Val, Dict) ->
    try F({existing, {Id,Val} }) of
     {modify, NewData} -> {updated, {database_updated, NewData,
added_to, Id}, dict:store(Id,NewData,Dict)};
     ignore -> {noupdate, ignore}
    catch
    _ : Val -> throw(Val)
    end.

% Auxiliar function which build a list of Rows to be returned. It
rethrows an exception if there it catches an exception from P
getResult(_,[]) -> [];
getResult(P, [Row | RestRows]) ->
    try P(Row) of
        true -> [Row | getResult(P, RestRows)];
        false -> getResult(P,RestRows)
    catch
    _ : _ -> throw(Row)
    end.
```

2 Tests

2.1 Parser Tests

2.1.1 ParserTest.hs

The test should be run writing these commands in the interpreter:

```
:load “ParserTest.hs”
main
```

The code for the test is the following:

```
module ParserTest where

import Test.HUnit
import qualified SubsParser as Parser
import SubsAst

-- Test that strings are parsed correctly (Cannot contain ' character)
test1 :: Test
test1 = TestCase $ assertEqual ("The string to be parsed is valid")
(Right (Prog [ExprAsStm (String "TestString")])) (Parser.parseString
"'TestString';")

test2 :: Test
test2 = TestCase $ assertEqual ("The string to be parsed is invalid")
(Left Parser.NoParse) (Parser.parseString "'Test'invalidString;")

-- Test that parsing numbers are ok. Numbers of more than 8 digits are
not parsed
test3 :: Test
test3 = TestCase $ assertEqual ("The number to be parsed is correct")
(Right (Prog [ExprAsStm (Number 124542)])) (Parser.parseString
"124542;")

test4 :: Test
test4 = TestCase $ assertEqual ("The number to be parsed is correct")
(Right (Prog [ExprAsStm (Number $ -124542)])) (Parser.parseString "-
124542;")

test5 :: Test
test5 = TestCase $ assertEqual ("The number to be parsed is
incorrect")  (Left Parser.NoParse) (Parser.parseString "1245421453;")

test6 :: Test
test6 = TestCase $ assertEqual ("The number to be parsed is
incorrect")  (Left Parser.NoParse) (Parser.parseString "-1245421453;")
```

```
-- Test different functions of the grammar
test7 :: Test
test7 = TestCase $ assertEqual ("The function to be parsed is
correct")  (Right (Prog [ExprAsStm (Call "-" [Number 122,Number
122])])) (Parser.parseString "122 - 122;")


test8 :: Test
test8 = TestCase $ assertEqual ("The function to be parsed is
correct")  (Right (Prog [ExprAsStm (Call "+" [Number
122,TrueConst])])) (Parser.parseString "122 + true;")

test9 :: Test
test9 = TestCase $ assertEqual ("The function to be parsed is
correct")  (Right (Prog [ExprAsStm (Call "%" [Number
122,FalseConst])])) (Parser.parseString "122 % false;")

test10 :: Test
test10 = TestCase $ assertEqual ("The function to be parsed is
incorrect")  (Left Parser.NoParse) (Parser.parseString "122 / 122;")

-- Test over operator precedences
test11 :: Test
test11 = TestCase $ assertEqual ("Precedences to be parsed are wrong")
(Right (Prog [ExprAsStm (Comma (Call "<" [Number 42,Call "===" [Number
31,Call "*" [Number 2,Number 14]]]) (Call "+" [Number 13,Call "<"
[Number 12,Call "===" [Number 3,Call "%" [Number 12,Number 2]]]])])))
(Parser.parseString " 42 < 31 === 2 * 14 , 13 + 12 < 3 === 12 % 2 ;")

-- Test whitespace handling
test12 :: Test
test12 = TestCase $ assertEqual ("Whitespace handling is wrong")
(Right (Prog [ExprAsStm (Call "%" [Number 122,FalseConst])]))
(Parser.parseString "  122 % false;")

test13 :: Test
test13 = TestCase $ assertEqual ("Whitespace handling is wrong")
(Right (Prog [ExprAsStm (Call "%" [Number 122,FalseConst])]))
(Parser.parseString " 122 %   false;     ")

test14 :: Test
test14 = TestCase $ assertEqual ("Whitespace handling is wrong")
(Right (Prog [ExprAsStm (Call "%" [Number 122,FalseConst])]))
(Parser.parseString "  122 % false   ;")

test15 :: Test
test15 = TestCase $ assertEqual ("Whitespace handling is wrong")
(Right (Prog [ExprAsStm (Call "%" [Number 122,Var "varxs"])]))
(Parser.parseString "  122 %  varxs;")

tests :: Test
tests = TestList [TestLabel "Strings" $ TestList [test1, test2],
TestLabel "Numbers" $ TestList [test3, test4, test5, test6] ,
                  TestLabel "Operations" $ TestList [test7, test8,
test9, test10], TestLabel "Precedences" test11, TestLabel "Spaces" $
TestList [test12, test13, test14, test15] ]
```

```
main :: IO Counts
main = runTestTT tests
```

The results provided by the test are:

```
Cases: 15  Tried: 0  Errors: 0  Failures: 0
Cases: 15  Tried: 1  Errors: 0  Failures: 0
Cases: 15  Tried: 2  Errors: 0  Failures: 0
Cases: 15  Tried: 3  Errors: 0  Failures: 0
Cases: 15  Tried: 4  Errors: 0  Failures: 0
Cases: 15  Tried: 5  Errors: 0  Failures: 0
Cases: 15  Tried: 6  Errors: 0  Failures: 0
Cases: 15  Tried: 7  Errors: 0  Failures: 0
Cases: 15  Tried: 8  Errors: 0  Failures: 0
Cases: 15  Tried: 9  Errors: 0  Failures: 0
Cases: 15  Tried: 10  Errors: 0  Failures: 0
Cases: 15  Tried: 11  Errors: 0  Failures: 0
Cases: 15  Tried: 12  Errors: 0  Failures: 0
Cases: 15  Tried: 13  Errors: 0  Failures: 0
Cases: 15  Tried: 14  Errors: 0  Failures: 0

Cases: 15  Tried: 15  Errors: 0  Failures: 0
Counts {cases = 15, tried = 15, errors = 0, failures = 0}
```

2.2 Interpreter Tests

All the tests should be run writing: “runhaskell Subs.hs <nameOfTestFile>.js”

2.2.1 interpreterTest_WorkingExpressions.js

```
var xs = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
var s = 'abcde';
var x = 2;
var y = 3;
var z = 4;
x = y = z = 1;
var result = 2 * 8 < 3;
var result2 = ( 2  < 3 ) === false;
var arrCompr = [ for (x of xs) if (x < 6) x ];
var arrCompr2 = [ for (y of arrCompr) y = y + 10 ];
var arrCompr3 = [ for (z of s ) z = z + ' was evaluated correctly!'];
```

Result for interpreterTest_WorkingExpressions.js :

```
SubsInterpreter.hs:104:1: Warning:

    Pattern match(es) are overlapped

    In an equation for `stm': stm s = ...



SubsInterpreter.hs:227:1: Warning:

    Pattern match(es) are overlapped

    In an equation for `evalExpr': evalExpr expr = ...


arrCompr = [0, 1, 2, 3, 4, 5]

arrCompr2 = [10, 11, 12, 13, 14, 15]

arrCompr3 = ["a was evaluated correctly!", "b was evaluated
correctly!", "c was evaluated correctly!", "d was evaluated
correctly!", "e was evaluated correctly!"]

result = false

result2 = false

s = "abcde"

x = 1
```

```
xs = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

y = 1

z = 1
```

2.2.2 interpreterTest_WorkingWrongExpressions.js

```
var wrongValue = [ for (x of [0])
                   for (y of [1,2,3])
                   for (z of [1,2,3]) x = x + 1 ];
```

Result for interpreterTest_WorkingWrongExpressions.js :

```
SubsInterpreter.hs:104:1: Warning:

    Pattern match(es) are overlapped

    In an equation for `stm': stm s = ...

SubsInterpreter.hs:227:1: Warning:

    Pattern match(es) are overlapped

    In an equation for `evalExpr': evalExpr expr = ...


wrongValue = [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

2.2.3 interpreterTest_NotWorkingExpressions.js

```
var notWorking = [ for (x of [0])  if (x < 10) for( y of [0])  x ];

var notWorking2 = [ for (x of [0])  if (x < 10) if( x === 0)  x ];
```

```
Results for interpreterTest_NotWorkingExpressions.js :

SubsInterpreter.hs:104:1: Warning:

    Pattern match(es) are overlapped

    In an equation for `stm': stm s = ...
```

```
SubsInterpreter.hs:227:1: Warning:

    Pattern match(es) are overlapped

    In an equation for `evalExpr': evalExpr expr = ...

Subs.hs: Error "The expression Compr (\"x\",Array [Number 0],Just
(ArrayIf (Call \"<\" [Var \"x\",Number 10]) (Just (ArrayForCompr
(\"y\",Array [Number 0],Nothing))))) (Var \"x\") couldn't be
interpreted with the following environment fromList []"
```

2.3 Tests Generic Replicated Server Library

2.3.1 tests_gen_replicated.erl

Tests should be run writing these prompts in the Erlang shell (X denotes the number of replicas we want to use in the test):

```
c(gen_replicated).
c(replica).
c(tests_gen_replicated).

tests_gen_replicated:testing(X).
tests_gen_replicated:testing2(X).
tests_gen_replicated:testing3(X).
```

The code for the test is the following:

```
-module(tests_gen_replicated).

-import(gen_replicated, [start/2, stop/1, read/2, write/2]).
-export([testing/1, testing2/1, testing3/1, init/0, handle_read/2,
handle_write/2]).

% Add some contacts, get a couple of existiong ones from the list and
finally stop the server. Then, it's tried to keep sending requests
although the server is stopped
testing(NumReplicas) ->
    {ok,Server} = gen_replicated:start(NumReplicas,
tests_gen_replicated),
    Rep1 = gen_replicated:write(Server, {add, {"Person1",
"falseStreet123", "21332"} } ),
    Rep2 = gen_replicated:write(Server, {add, {"Person2",
"falseStreet9", "2131532"} } ),
    Rep3 = gen_replicated:write(Server, {add, {"Person3",
"falseStreet54", "21321332"} } ),
    Rep4 = gen_replicated:read(Server, {find, "Person1"}),
    Rep5 = gen_replicated:read(Server, {find, "Person3"}),
    Rep6 = gen_replicated:stop(Server),
    Rep7 = gen_replicated:read(Server, {add, {"Person3",
"falseStreet123", "21332"} }),
    {Rep1, Rep2, Rep3, Rep4, Rep5, Rep6, Rep7}.

% Trying to add, delete and update different contacts. Should yield
some errors (updating a non-added contact and removing a non-existing
contact)
% to proof that the code is correct
testing2(NumReplicas) ->
    {ok,Server} = gen_replicated:start(NumReplicas,
tests_gen_replicated),
    Rep1 = gen_replicated:write(Server, {add, {"Person1",
"falseStreet123", "21332"} } ),
    Rep2 = gen_replicated:write(Server, {delete, "Person1" } ),
    Rep3 = gen_replicated:write(Server, {update, {"Person1",
"newFalseStreet", "112"}} ),
```

```
    Rep4 = gen_replicated:write(Server, {add, {"Person2",
"falseStreet5", "22"} } ),
    Rep5 = gen_replicated:write(Server, {update, {"Person2",
"newFalseStreet2", "31"}}),
    Rep6 = gen_replicated:write(Server, {delete, "Person1"}),
    {Rep1, Rep2, Rep3, Rep4, Rep5, Rep6}.

% Add some contacts, find an existing one and find a non-added one.
The latter makes the server to stop. Then it is tried to stop the
server again (Yielding no errors)
% and it is sended another request which will be denied since the
server is stopped
testing3(NumReplicas) ->
    {ok,Server} = gen_replicated:start(NumReplicas,
tests_gen_replicated),
    Rep1 = gen_replicated:write(Server, {add, {"Person1",
"falseStreet123", "21332"} } ),
    Rep2 = gen_replicated:write(Server, {add, {"Person2",
"falseStreet53", "23"} } ),
    Rep3 = gen_replicated:write(Server, {add, {"Person3",
"falseStreet76", "42"} } ),
    Rep4 = gen_replicated:read(Server, {find, "Person1"}),
    Rep5 = gen_replicated:read(Server, {find, "Person4"}),
    Rep6 = gen_replicated:stop(Server),
    Rep7 = gen_replicated:read(Server, {add, {"Person3",
"falseStreet35", "213"} }),
    {Rep1, Rep2, Rep3, Rep4, Rep5, Rep6, Rep7}.

% Idea and some code taken from pb.erl and pbtest.erl used during the
lectures. The handlers return every possible
% value (Exceptions, stop events, updatings...) to try every branch of
the code
init() -> dict:new().
handle_write({delete, Name}, Contacts) ->
    case dict:is_key(Name, Contacts) of
        false -> {noupdate, {couldnt_find_contact, Name}  };
        true -> {updated , {Name ,removed_ok}, dict:erase(Name,
Contacts) }
    end;

handle_write({add, {Name, _Street, _PostCode} = Contact}, Contacts) ->
    case dict:is_key(Name, Contacts) of
        false -> {updated, {added_contact, Name}, dict:store(Name,
Contact, Contacts)};
        true  -> throw({Name, already_added})
    end;
handle_write({update, {Name, _, _} = Contact}, Contacts) ->
    case dict:is_key(Name, Contacts) of
        false -> throw({Name, not_added_cannot_update});
        true  -> {updated, {Name,contact_updated}, dict:store(Name,
Contact, Contacts)}
    end.

handle_read({find, Name}, Contacts) ->
    case dict:find(Name, Contacts) of
        error -> stop;
        {ok, Value} -> {reply, Value}
    end;
handle_read(list_all, Contacts) ->
    List = dict:to_list(Contacts),
    case List of
```

```
        [] -> throw(list_contacts_empty);
        L -> {reply, lists:map(fun({_, C}) -> C end, L)}
    end.
```

Results for tests_gen_replicated:testing(3). :

```
{{added_contact,"Person1"},
 {added_contact,"Person2"},
 {added_contact,"Person3"},
 {"Person1","falseStreet123","21332"},
 {"Person3","falseStreet54","21321332"},
 ok,"Cannot handle request. Server is stopped"}
```

Results for tests_gen_replicated:testing2(3). :

```
{{added_contact,"Person1"},
 {"Person1",removed_ok},
 {'ABORTED',exception,{"Person1",not_added_cannot_update}},
 {added_contact,"Person2"},
 {"Person2",contact_updated},
 {couldnt_find_contact,"Person1"}}
```

Results for tests_gen_replicated:testing3(3). :

```
{{added_contact,"Person1"},
 {added_contact,"Person2"},
 {added_contact,"Person3"},
 {"Person1","falseStreet123","21332"},
 {'ABORTED',server_stopped},
 ok,"Cannot handle request. Server is stopped"}
```

2.4 Tests AlzheimerDB

2.4.1 tests_alzheimer.erl

Tests should be run writing these prompts in the Erlang shell:

```
c(alzheimer).
c(tests_alzheimer).
tests_alzheimer:testing1().
tests_alzheimer:testing2().
tests_alzheimer:testing3().
```

The code for the test is the following:

```
-module(tests_alzheimer).

-export([testing1/0, testing2/0, testing3/0]).
-export([queryFunction/1, upsertFunction/1, upsertFunction2/1]).
-import(alzheimer, [start/0, upsert/3, query/2]).

% Inserts two people in the database. It queries but it'll get an
empty row (The function P returned false for every row). Then it adds
another row
% and finally queries to get the latter row added

testing1() ->
    {ok, Aid} = alzheimer:start(),
    Rep1 = alzheimer:upsert(Aid, "Person1", fun
tests_alzheimer:upsertFunction/1),
    Rep2 = alzheimer:upsert(Aid, "Person2", fun
tests_alzheimer:upsertFunction/1),
    Rep3 = alzheimer:query(Aid, fun tests_alzheimer:queryFunction/1),
    Rep4 = alzheimer:upsert(Aid, "WantedPerson", fun
tests_alzheimer:upsertFunction/1),
    Rep5 = alzheimer:query(Aid, fun tests_alzheimer:queryFunction/1),
    {Rep1,Rep2,Rep3,Rep4,Rep5}.

% Inserts two people in the database. Calls the query function and it
will get an exception.
testing2() ->
    {ok, Aid} = alzheimer:start(),
    Rep1 = alzheimer:upsert(Aid, "Person1", fun
tests_alzheimer:upsertFunction/1),
    Rep2 = alzheimer:upsert(Aid, "WantedPerson2", fun
tests_alzheimer:upsertFunction/1),
    Rep3 = alzheimer:query(Aid, fun tests_alzheimer:queryFunction/1),
    {Rep1,Rep2,Rep3}.

% Try to see what happens when the upsert function returns an
exception
testing3() ->
    {ok, Aid} = alzheimer:start(),
```

```
    alzheimer:upsert(Aid, "Person1", fun
tests_alzheimer:upsertFunction2/1).

% Function for queries (Returns every possible return value, including
exceptions)

queryFunction(Row) ->
    [Key, Value] = tuple_to_list(Row),
    case (lists:member( "WantedPerson", [Key] ) or lists:member(
"WantedPerson2", [Key] ))of
        true -> case lists:member("WantedData",tuple_to_lis(Value)) of
                        true -> true;
                        false -> throw(missing_info_for_WantedPerson)
                end;

        false -> false
    end.


% Function for inserting-updating (Returns every possible return
value, but exceptions)
upsertFunction({new, "WantedPerson"}) ->
    {modify, {"WantedData", invalid_info, invalid_info}};

upsertFunction({new, "WantedPerson2"}) ->
    {modify, {"NoData", invalid_info, invalid_info}};

upsertFunction({new, _}) ->
    {modify, {invalid_info, invalid_info, invalid_info}};

upsertFunction({existing, {_,Val}}) ->
    L =  tuple_to_list(Val),
    case lists:member( "GoodInformation", L) of
        true -> {modify, Val};
        false -> ignore
    end.

% Function used for checking exceptions in the inserting-updating
function
upsertFunction2(_Arg) -> throw(invalid_request_for_FunctionT).
```

Results for tests_alzheimer:testing1(). :

```
{{database_updated,{invalid_info,invalid_info,invalid_info},

                   added_to,"Person1"},

 {database_updated,{invalid_info,invalid_info,invalid_info},

                   added_to,"Person2"},

 {ok,[]},

 {database_updated,{"WantedData",invalid_info,invalid_info},

                   added_to,"WantedPerson"},

 {ok,[{"WantedPerson",
```

```
       {"WantedData",invalid_info,invalid_info}}]}}
```

Results for tests_alzheimer:testing2(). :

```
{{database_updated,{invalid_info,invalid_info,invalid_info},
                   added_to,"Person1"},
 {database_updated,{"NoData",invalid_info,invalid_info},
                   added_to,"WantedPerson2"},
 {error,{"WantedPerson2",
         {"NoData",invalid_info,invalid_info}}}}
```

Results for tests_alzheimer:testing3(). :

```
{'ABORTED',exception,invalid_request_for_FunctionT}
```