# AP Exam 2013

Arni Asgeirsson lwf986

November 7, 2013

# Contents

# 1 Introduction

The following is a report documenting my implementations of a Salsa parser, Sala interpreter written in Haskell and a multi process atomic transaction server written in Erlang, and all relevant thoughts, concerns and a discussion of interesting problems that arose while completing the work.

I believe that the greatest focus should be on these thoughts, concerns and discussions as they play a big part in the general assessment of arguments for my code, therefore these will also have great focus in this report. A note to the reader is that I tend to have many thoughts, concerns and discussions with my self, and my have a tendency to write more that others, although it is *usually* of high quality. I have noticed the very high page number, although don't fret as much of it is section headers and stuff like that stealing pages.

I am handing in a total of 13 files and these are: *src/at_ server/at_ server.erl*, *src/at_ server/at_ extapi.erl*, *src/test_ at_ server.erl*, *src/salsa/SalsaParser.hs*, *src/salsa/SalsaInterp.hs*, *src/salsa/Test_ Parser.hs*, *src/salsa/Test_ Interp.hs*, *src/salsa/Gpx.hs*, *src/salsa/SalsaAst.hs*, *src/salsa/test_files/multi.salsa*, *src/salsa/test_files/simple.salsa*, *src/salsa/test_files/empty.salsa*, *src/salsa/test_files/invalid.salsa.*

Note that I have not altered the *Gpx.hs* file nor the *SalsaAst.hs* file, these are merely added for convention.

All the files are also included to this report and can be viewed in **Appendix E**.

I have decided to let all my tests be described in **Appendix A** and therefore separated from the rest of the report and individual assessments. When describing my tests I also make my final assessments on the individual parts. I use both *QuickCheck* for Haskell, *HUnit* and regular unit tests when testing.

I have used the -Wall flag when compiling my Haskell code.

Note that I am aware that the function `read` is a partial function, although I have been taught through the Advanced Programming course that this one is OK to use.

# 2 Question 1: The Salsa Language

The following describes my implementation of a Salsa parser in Haskell.

I have chosen to use ReadP as it allows me to check wether my grammar is ambiguous.

## 2.1 Fixing the grammar

The given grammar is ambiguous and this must be dealt with before proceeding with the actual implementation. The given grammar is shown in **Appendix B** and we shall call this G0.

### 2.1.1 Precedence and Associativity

It is assumed that the four operators +, -, @ and || all are left-associative.

I assume that + and - have the same precedence, and as the assignment text specifies then @ has higher precedence than ||, meaning that if we have the following pseudo input “cmd || cmd @ Vident” then this corresponds to the input “cmd || ( cmd @ Vident )”.

I further assume that . has higher precedence than + and - meaning that “r . x + c . y” is the same as “(r . x) + (c . y)”.

The associativity is already in place and the precedence of +, - and . are also already supported by the current grammar. What is missing is that @ and || have the correct precedence. We can ensure this by changing the grammar with the following algorithm and perform it on the non-terminal Command from G0. We might need to use the bias choice <++ operator instead of the fully symmetric +++ operator some places, but if needed we will come to that.

```
A ::= A a1 A
    | A a2 A
    | a3 .
->
A ::= A a1 B
    | B .
B ::= B a2 C
    | C .
C ::= a3 .
```

where a1 & a2 are terminals, a3 is something 'else' and A,B,C are non-terminals. Now a2 will have higher precedence than a1.

The resulting grammar is shown in **Appendix B** and we shall name it G1.

### 2.1.2 Left factorization

We can see that G1 contains the case of left factorization. We therefore fix this by doing left factorization on G1.

I have summarized how I do left factorization in the following algorithm:

```
A ::= B a1
    | B a2.
->
A ::= B A' .
A'::= a1
    | a2 .
```

where A and A' are non-terminals and B, a1 & a2 could be anything.

The resulting grammar shall be called G2, and is shown in **Appendix B**.

### 2.1.3 Left recursion

When we look at G2 we can quickly see that there is a few cases of direct left-recursion. Therefore the next step is for me to remove left-recursion from our current grammar G2, with the following algorithm:

```
A ::= A a1
    | b1 .
->
A ::= b1 A' .
A'::= a1 A'
    | e .
```

where A and A' are non-terminals and a1 & b1 can be anything

When applying the algorithm on G2 we get the grammar G3 which is shown in **Appendix B**.

In the rest of the report, G3 will be the referred grammar unless something else is specified.

```
data Error = NoParsePossible String
           | AmbiguousGrammar [(Program, String)]
           | UnexpectedRemainder Program String
```

Figure 1: My Error date type.

## 2.2 My assumptions on the grammar

All my assumptions for the grammar can be viewed Appendix D.

## 2.3 The Code

The code for the Salsa parser can be found in the file *SalsaParser.hs* which implements the module *SalsaParser* and exports the two function `parseString` and `parseFile`, and the `Error` data structure. The accompanied test file is called *Test_ Parser.hs* and is described in **Appendix A**.

Note that I am aware that *HLint* spits out a hint of "Reduce duplication" although I believe that fixing the hint would negatively affect the readability of my code.

### 2.3.1 Interface functions

**parseString :: String -> Either Error Program** This function is rather straight forward as it simply parses a given input string to a `Program` or an `Error` if the input string is invalid.

**parseFile :: FilePath -> IO (Either Error Program)** This function extends the functionality of `parseString` by reading the input string from a given file path.

### 2.3.2 DataTypes

**Error** I have defined the Error type to be of what is shown in Figure 1.

Here we can see that a invalid salsa program can either return that it cannot be parsed to anything at all, a parsed program and some remainder or multiply parsed programs showing that the grammar is ambiguous, or at least that the implementation of the grammar is.

### 2.3.3 Parser functions

These functions are generally not that interesting as they follow the grammar very strict, the following will therefore not be a very in-depth explanation, as there is not much depth to talk about.

**parse :: ReadP Program -> String -> Either Error Program** This function takes a parser `p` of type `ReadP Program` and a string `s` and parses `s` with `p` and and returns the parsed program or the appropriate error if the input did not adhere to the grammar.

**runParser :: ReadP Program** This parser is a wrapper for the `pProgram` parser and simply makes sure to skip spaces both before and after parsing a `Program` and also makes sure that we are at the end of the input after parsing a program.

**pProgram :: ReadP Program** Parses a `Program` by trying to parse a list of definitions and or commands.

**pDefComs :: ReadP [DefCom]** Parses a single definition or command and then tries to parse a list of definitions and or commands.

**pDefComs' :: ReadP [DefCom]** Parses either a list of commands and or definitions, nothing or a symmetric choice of both.

**pDefCom :: ReadP DefCom** Parses either a command or a definition.

**pDefinition :: ReadP Definition** Parses a definition by either parsing a view definition, rectangle, circle, view or a group definition. I decided to split `pDefinition` into five individual parsers `hViewdef`, `hRectangle`, `hCircle`, `hView` and `hGroup` to make it more readable and intuitive to understand what is happening.

**pCommand :: ReadP Command** Is the top-level parser for parsing a command by using the `pCommand'`, `pCommand2`, `pCommand2'` and `pCommand3` parsers.

This parses a command2 type which is either a move, bracketed or a series of '@' commands and then parses a command' with the previous command as incoming value.

**pCommand' :: Command -> ReadP Command** Either parses a '||' command followed by another command' parsing or nothing.

**pCommand2 :: ReadP Command** First parses a command3 then parses a command2' with the command3 as incoming value.

**pCommand2' :: Command -> ReadP Command** Either parses a '@' command followed by another command2' parse or nothing.

**pCommand3 :: ReadP Command** Parses either a move command or a command wrapped in a set of curly-brackets.

**pVIdents :: ReadP [Ident]** Parses a single vident followed by possible a list of vidents.

**pVIdents' :: ReadP [Ident]** Parses a list of vidents or nothing.

**pSIdents :: ReadP [Ident]** Parses a single sident followed by possible a list of sidents.

**pSIdents' :: ReadP [Ident]** Parses a list of sidents or nothing.

**pPos :: ReadP Pos** Parses a position of either the `Abs` or `Rel` type.

**pExpr :: ReadP Expr** Parses an expression by parsing a prim and then using that as incoming value to call `pExpr'`.

**pExpr' :: Expr -> ReadP Expr** Starts by parsing either a plus or minus expression `e` with the incoming value and then another expr' with `e`, otherwise it parses nothing by just returning the incoming value.

**pOp :: Expr -> ReadP Expr** Takes an incoming expression and either parses a plus or a minus, then the associated prim and returns either a plus or minus expression.

**pPrim :: ReadP Expr** Parses either an integer, an expression contained within a set of parenthesis or either an `Xproj` or `Yproj` expression.

**pProj :: Ident -> ReadP Expr** This parser takes an ident as an incoming value and parses either an 'x' or a 'y' and then returns the appropriate Expression of type `Xproj` or `Yproj`.

**pColour :: ReadP Colour** Parses one of the defined colours.

**pVIdent :: ReadP Ident** Simply parses a vident and skips all white spaces both before and after the word.

**pSIdent :: ReadP Ident** Parses a sident and, as same as `pVIdent`, also skips all white spaces both before and after the word. If the parsed word is one of the reserved words or one of the colour names then it fails, otherwise it returns the parsed word.

**pInteger :: ReadP Integer** Parses an integer and skips all white spaces both before and after the integer.

#### 2.3.4 Helper functions

**bracks :: Char -> ReadP b -> Char -> ReadP b** Takes in two characters `a` and `b` and a parser `p`, and parses the `a` and `b` around the parsing of `p` with the use of `charT` and returns the result from `p`.

**stringT :: String -> ReadP ()** Parses a given string `s` and skips white spaces around `s`.

**charT :: Char -> ReadP ()** Parses a given character `c` and skips white spaces around `c`.

# 3 Question 2: Interpreting Salsa

In this section I will try and implement a Salsa interpreter in Haskell with the use of two monads. First I want to list the assumptions that I have taken while working on the implementation, followed by a description of my code.

## 3.1 My assumptions

- As the assignment text specifies then I assume that the given input `Program` is a valid `Program` in that sense that it will not cause any compile errors. Allowing me to use the Haskell function `error` to throw an error if an invalid `Program` should be used in the interpreter.

- I was unsure of what a nested `At` command should be interpreted to. I decided that an inner `At` does not consider an outer and visa versa meaning that a nested `At` command will only be visible for the innermost `At` command. Consider the case "b -> (0,0) @ A @ B", here first B will be set as the active view, then A and then the move is performed while A is active and this no effect on B.
- I assume that when defining a group the same name cannot appear twice in the list.
- I assume that you cannot have group names inside a the list of views in a group definition.
- I assume that no two names can be used to define two things.
- I assume that the first `DefCom` will always be a view definition, as it does not make sense to perform any operations on a non-existing view.
- A shape can only be moved around within a view it has been defined, i.e. a shape that has not been defined within a view A can never appear in A.
- The list of frames goest from left to right, i.e. the first keyframe is the first element of the list and the last keyframe is the last element.
- I assume that a shape is allowed to move outside of the bounds of a view that it has been define in.

## 3.2 The Code

My implementation of the Salsa interpreter can be found in the file *SalsaInterp.hs* which defines and implements the module *SalsaInterp*. It exports the type `Position` and the two functions `interpolate` and `runProg`. The associated test file is *Test_Interp.hs* and is further described in **Appendix A**.

### 3.2.1 Interface functions

**interpolate :: Integer -> Position -> Position -> [Position]** This function takes an `Integer n`, a start `Position p1` and an end `Position p2` and interpolates between the two positions. In other words the function calculates the distribution of points in a straight line from `p1` to `p2`, including `p2` and excluding `p1`, where the number of calculated points are equal to `n`, where their is an equal distance from any point to the neighbor point.

The function assumes that `n` is non-negative.

It is implemented by considering a straight line between `p1` and `p2` and then 'slice' it up in n equal sizes and then returning each 'slice' point and including the end point, or excluding the start point if you will. I have avoided using floating point values, and the chaos they apparently brought with them, to consider the length of each slice to be from $0 - 100$ instead of $0.0 - 1.0$, which does lose me some precession, but I gain a lot more readable code and as we have to return integers anyway, the loss is very slim.

**runProg :: Integer -> Program -> Animation** This function initiates and runs an interpretation of a giving Program p with a given frame rate n, and returns the produced Animation. It initiates a `Salsa` monad with an empty context and then interprets each `DefCom` in the given `Program`.

### 3.2.2 Context

As one can see in Figure 2 I have decided to let the context be a two part thing, where the first part represents the read-only environment, which is read-only for the SalsaCommand monad and the second part is the read-write state.

I have named the first part ConEnvironment which is a type alias for a tuple of three elements: an Environment for binding idents to definitions, a list of the active views and an integer representing the given frame rate.

The second part is named State and is a mapping from an identifier Ident to a list of views and positions, describing the current location of each shape in each view they have been defined in.

```
data Context = Context ConEnvironment State
type ConEnvironment = (Environment, [Ident], Integer)
type Environment = M.Map Ident Definition
type State = M.Map Ident [(Ident,Position)]
```

Figure 2: My implementation of the Context data type.

I have also implemented a few functions to work on a Context and Animation. These are described in **Appendix D**.

### 3.2.3 The Monads

I have created two monads that when used together does the interpretation of a given Program.

The first one is named SalsaCommand and its type declaration can be seen in Figure 3.

```
newtype SalsaCommand a = SalsaCommand {runSC :: Context -> (a,State)}
```

Figure 3: My definition of the SalsaCommand monad.

As one might notice then it takes a Context and returns some value a and a State, disallowing the monad to manipulate with the given Context.

The second one is the Salsa monad which has the declaration that can be seen in Figure 4.

```
data Salsa a = Salsa ((Context,Animation) -> (a,(Context,Animation)))
```

Figure 4: My declaration of the Salsa monad.

This is a state monad that carries around a Context and the Animation that is being built. I decided to let the Salsa monad carry these two around to easily manipulate with the two and return the completed Animation once done.

The actual Monad instantiating of the two types shown above can be seen in **Appendix E**.

My overall strategy for using the two monads is to let Salsa be the top dog and the interface to the actual interpretation, it self interprets all definitions and builds up the context environment (part 1) and it then uses the SalsaCommand monad to interpret the commands. After the Salsa monad has let a SalsaCommand monad interpret a command and return the next State of the Context, the Salsa monad compares the previous State with the new returned State and generates all the needed draw instructions by interpolation between the two States for each shape in each view where it has moved.

Salsa then updates the global Context to contain the new State returned from the SalsaCommand.

The following is a short description of a set of functions to update the monads.

**askCmd :: SalsaCommand Context** This function simply fetches the Context inside a SalsaCommand monad.

**updateState :: (Context -> State) -> SalsaCommand ()** updateState takes a function f that performs some operation on a Context and returns a State and performs f on the Context within SalsaCommand to alter the eventual returned State.

**askCont :: Salsa Context** Does the same as askCmd, except now we fetch the Context inside a Salsa monad instead.

**runSalsa :: Context -> Salsa a -> (a,(Context,Animation))** As the name implied this function takes some initial Context, a Salsa monad and runs the monad, i.e. starts it.

**updateContext :: (Context -> Context) -> Salsa ()** This function takes a function f that takes a Context and returns a Context, and applies this function f to the inner Context of a Salsa monad.

**updateAnimation :: (Animation -> Animation) -> Salsa ()** updateAnimation takes a function f, that performs some operations on an Animation and returns another Animation, and applies f to the Animation part of the state of a Salsa monad.

### 3.2.4 Interpret functions

**command :: Command -> SalsaCommand ()** The command function interprets a Command cmd within a SalsaCommand monad.

If cmd is a Move command then for each given shape id sid we must for each view that sid is located in, check if any of these are one of the currently active views, and if so we must update our position accordingly to the Move command.

If is an At command then we must update the active views to be of some temporary value, then execute the given command and then revert the active views to their previous self. But, as the SalsaCommand cannot change the ConEnvironment part, then we cannot update the currently active views, what we can do instead is to alter the State value of the Context.

My trick is to use the setTmpActiveViews helper function to go through the State and look at each view in each binding and if the view is one of the temporary active views then I map it to one of the actual active views, and remember this mapping. This is done through the entire State, followed by now mapping all the actives views that are distinct from the set of temporary views to some random value that I know no one can have, namely something with an underscore as the first letter, of course these mappings are also remembered.

Now I can run the given command with the updated State, and when it returns I use my mappings from before to swap back, and then I have not changed the Context and was still able to update the list of active views. Although this sounds very time consuming and I bet it is, a more reasonable way would merely to update the first part of the Context, and then run the command, but as I cannot see how this can be done without change the type of the monad, this is not an option.

The `Par` command is a really easy one as it simply just executes the two commands.

**defCom :: DefCom -> Salsa ()** This interpret function either interprets a `Definition` or a `Command`, if it is a definition then nothing further needs to be done beside interpreting it, but if it wants to interpret a `Command` then a `SalsaCommand` is used with the current `Context` of the running `Salsa` monad, when the monad returns then we must compare the `State` returned from the `SalsaCommand` with our own and generate a set of instructions to go from the previous state to the next. These instructions must be added to the current frame and then as we went from one keyframe to the next, we must also ensure this by beginning a new frame.

**definition :: Definition -> Salsa ()** This function interprets a given `Definition`. If it is a view definition then we evaluate the given size expressions, add the definition to the environment, update it to be the list of active views, and add the view to the list of defined views in the `Animation`.

If the definition is a simple `View` defeinition meaning that it is not the same as above, but a statement to change the set of active views, then we do so by retrieving the list that corresponds to the given id and update the `Context`.

If it is a group definition then we add the binding to our environment and set the given list of view names to be the list of active views.

If we get a shape definition then we evaluate the provided values, add it to our environment, draw the shape in the current keyframe and in all current views. Furthermore we update our `State` value to contain the new shape in the current active views.

### 3.2.5 Interpret helper functions

**evalNextPoint :: Pos -> Position -> SalsaCommand Position** This function simply takes a Pos and a Position pos and evaluates Pos and returns either a absolut position or a reletive one acording to Pos.

**setTmpActiveViews :: State -> [Ident] -> [Ident] -> SalsaCommand (State,[((Ident,Ident),Ident)])** As already mentioned then this function takes some `State`, a list of active views and a list of temporary views then it maps all occurrences of the temporary views in the `State` with one of the active views, and then follows by mapping all occurrences of the active views that are distinct from the list of temporary list with a somewhat random value prefixed with an underscore, then it returns the new state and the list of all the mappings.

**revertActiveViews :: State -> [((Ident,Ident),Ident)] -> SalsaCommand State** This function takes some `State s` and a list of mappings and reverts those mappings on `s` and returns this in the monad SalsaCommand.

**setNextPosition :: Eq t => [t] -> Pos -> (t, Position) -> SalsaCommand (t, Position)** This function takes a list of views `w`, a point and a tuple with a view `v` and a position. If `v` is in the list of views `w`, then the position must be updated in the tuple to be the result of the point and the position evaluated into one `Position`. If not then the same tuple that came in is also returned.

**getLowestPosition :: Position -> (ViewName,Position) -> Position** Returns the lowest x-coordinate and the lowest y-coordinate to be found between the two given Positions.

**compareStates :: State -> State -> Salsa [GpxInstr]** This function compares two `State`s and generates draw instructions for each shape `s` and for each view `s` has been moved in.

**generateInstructions :: Definition -> Integer -> [(ViewName, Position)] -> [(ViewName, Position)] -> Salsa [Frame]** This function is a helper to the compareStates function and generates a list of instructions for the given Definition, frame rate, list of old positions and list of new positions.

**positionToInstr :: Definition -> ViewName -> Position -> Salsa GpxInstr** This functions creates a single draw instruction with for the specific shape, view and position.

**evalExpr :: Monad m => Expr -> m Context -> m Integer** This function takes an `Expr e` to be evaluated and something that can provide us with some context to use when evaluating `Xproj` and `Yproj` as these require that we look in the State of the `Context` to determine what is the lowest either `x` or `y` value of a given shape. I have made the function generic by letting the 'ask' function be a parameter so that both the `Salsa` and the `SalsaCommand` monad can use the function.

### 3.2.6 Helper functions

**lookupKey :: Ord a => a -> M.Map a b -> b** This function simply performs a lookup in a given tree with the key `a` and returns the mapped value, if no mapping was found then an error is raised.

**removeDouble :: [a] -> [a] -> [a]** This function takes a list `x` and a list `y` and returns the list `z`, where `z` is the list that is left of `x` when all elements that both exist in `x` and `y` has been pulled out of `x`.

**evalColour :: Colour -> ColourName** `evalColour` merely returns the respected string representation for a Colour type.

**getLast :: [a] -> ([a],[a])** getLast takes in a list and returns a tuple where the first element is the whole input list expect for the last element and the second element of the tuple is the last element of the list wrapped in a list.

## 4 Question 3: Atomic Transaction Server in Erlang

In this section I will implement an atomic transactions server using Erlang.

I decided to use OTP for implementing my atomic transaction server, as this allows me to save a lot of time building up the server from the ground up, and provides me and the reader with a standardized interface and behavior. I chose *gen_server* over the other OTP like *gen_fsm* because our server is not a finite state machine but instead a server taking handling a lot of requests, where *gen_server* fits this task the best. This also explains all the 'comments' in the file.

## 4.1 My assumptions

- I always assume that the given process id, of an atomic transaction server, to all the interface function is indeed an atomic transaction server. I do no error handling nor check on this input.
- I make no assumptions on the given input State.
- I assume that no one tries to tamper with the server nor the transactions by guessing their process ids and sending them unwanted messages.

## 4.2 General structure of the server

Before proceeding to my actual implementation I want try and describe the overall structure of my atomic transaction server (ATS) and its managed transactions and requests, and some of the problems that arose.

### 4.2.1 Who talks to who?

When a new ATS has been started it is started in a new process and no transactions can be started nor do any live at startup. When a client wants to start a transaction the API function `begin_t/1` is called and this makes the ATS spawn a new transaction, which is in fact an ATS process in its self but 'loops' with another state data and expects different requests, but referred to as a 'transaction' or 'transactions'. All communication to a transaction is done through the ATS in which they were begun. All communication is done synchronously between the ATS and its transactions, except for the `update_t/3` call and when a timeout occurs while trying to stop a transaction, then a asynchronous stop message is sent instead.

Each transaction could also just have been a tuple containing the unique reference, their current state and some status and not a whole process and the ATS would then simply just maintain a list of these, although this would disallow the opportunity to do any parallel work on the transactions which we can now with `update_t/3`, without doing some silly workaround.

### 4.2.2 The state data

The ATS loops with its current state, which is provided and updated by the client, and a pool of transactions. These can be idle transactions, ready to be initiated as new transactions, or ongoing transactions.

A transaction also loops with its current state and a status. This status allows the ATS to know whether the transaction is aborted, ready or idle.

### 4.2.3 The status of each transaction and the management

As mentioned above a transaction can have one of the three status; aborted, ready or idle. aborted means that the transaction has been aborted and any further use while in this status should return aborted. ready is when the transaction has been begun but not aborted yet. idle means that the transaction is not used for anything right now and can be initiated by the ATS to act as a fresh transaction.

Although to avoid a lot of time spent communicating and waiting for responds from non-existing processes, then the ATS maintains it self a list of each transactions, by keeping their status, process id and the associated unique reference. Where it is the unique reference that acts as the key into this list allowing the ATS to respond quickly with `aborted` if the process does not exist, is idle or already aborted. Of course this lookup operation would always take $O(n)$ where $n$ is the length of the list, but the alternative would be to ask the transactions each time to know what their status is, and they might be busy with some expensive update function, and the alternative would also include the same lookup operation on a list of the same length, unless the ATS does not keep track of its transactions and therefore might risk waiting for responds from a process that might never respond.

There is one time the ATS needs to ask a transaction before knowing, and this is when a transaction failed to update its state through a `update_t/3` call as this is done asynchronously the server won't know until it asks the transaction. Although once the server knows then it updates its own list, so it won't have to ask again.

There is also a slight overhead of maintaining this list but I believe that it overcomes the mentioned alternative. An improvement could be to use another data structure to that has faster lookup and replace operations, as these are the most frequent operations used.

When looking through my code one might notice that this status is not always maintained by the transaction it self, and this is because the ATS gives more credit to its own list, and will only ask transaction if it is listed to be in status ready.

To sum um the above by answering to the posed problem:

> "Also, processes waiting for answers from aborted transactions must be answered with aborted as quickly as possible."

My solution is to let the ATS maintain a list of the statuses of each ongoing transaction to respond quickly by avoiding the hassle to communicate with a process before answering.

### 4.2.4 The pool

I like to use pools. There, I said it. But I am aware that this might not be the intention in this assignment, and that is why I have defined a 'global-variable' in top of the *at_server* module `MIN_POOL` that if set to true the ATS will not keep any idle transactions in its pool, with other words, not maintain a pool.

The fact that I map the unique reference to a process id in the ATS allows me to maintain a pool of idle workers as when an idle transaction gets initiated as a new transaction, a new reference is merely created and the old association is updated to the new reference which is also returned to the client.

An argument for maintaining a pool could be that if the server is heavy used and many transactions are begun and many are aborted due to either failing functions or commits. A lot of overhead will come from destroying and creating all these processes and in such a case it could be better them let them 'stick-around' instead of being destroyed for a quicker revival. Of course if the server is barely used, a pool would be of not much use, and might even cause extra overhead to keep these processes alive. An alternative could be to add these idle processes in a sleeping-queue avoiding that they wastes any cpu cycles being idle.

As a side-note then I tried to do some small upscaled tests to test if any difference were to be found using the pool or not, but no noticeable difference were to be discovered.

## 4.3 The Code

The following is a description of my implementation of the *at_ server* module and the *at_ extapi* module, which can be found in the files *at_ server.erl* and *at_ extapi.erl*, and most of the interesting problems that arose. As already mentioned then I have defined the `MIN_POOL` value at the top of the *at_ server* module, I have also defined a `TIME_OUT` value to allow for quick change of the accepted timeout value sent with each synchronous call to the processes, this is mostly used to go from the general default value *5000ms* to `infinity`. I advice you to not set the `TIME_OUT` value to `infinity` when running my tests as some of them will cause the server in a dead_lock state where the server is waiting for a transaction and visa versa.

### 4.3.1 The Server API / Interface functions

Note that to try and avoid letting the processes crash I have tried to catch the timeout error around each synchronous call too and within the server, although if the `TIME_OUT` value is set to something very low my `tryUpdate/0` test will fail with `timeout` in its fourth test case, which I for some reason cannot explain nor fix.

**start(State)** This function creates a new atomic transaction server and returns the message `{ok,Pid}` where `Pid` is the process id of the newly created ATS.

Several ATSs can be started using `start/1` and run independently.

Note that I don't make any assumptions on the given `State` value, as this should be allowed to be anything and is defined through the functions that act upon it.

**stop(AT)** `stop/1` makes a synchronous call to the ATS with the given process id `AT`, and orders it to shutdown and return the current state. `stop/1` will not return until all processes handled by the ATS, including it self, has been shutdown, allowing the caller to be certain that when `stop/1` returns everything is cleaned up. Instead of making the call asynchronous and forcing the caller to be in doubt and ultimately the need to wait some random amount of time before the processes are shutdown, or have some unexpected behavior.

Of course with the exception that the `TIME_OUT` value has been set to something other than `infinity`.

**doquery(AT, Fun)** This function performs the given function `Fun` on the current state of the ATS with pid `AT` and returns the result in the form `{ok, Result}`.

If `Fun` fails `error` is returned. `doquery/2` does not update the state of the server in any way.

**begin_t(AT)** Here a new transaction is initialized and retrieves a copy of the current state of the given ATS, and a unique reference to the transaction is returned. Updates and queries can then be performed on the new transaction, although it belongs to the ATS that it was created within, and can therefore not be used to update the state of some other ATS.

For the unique reference, `make_ref()` is used as it guarantees that approximately the first $2^{82}$ created references are unique[1]. One could just return the process id of the transaction, but this makes it easier for the user to send unwanted messages to the transactions, stop it or in any other way mess it up.

[1] http://www.erlang.org/doc/man/erlang.html#make_ref-0

**query_t(AT, Ref, Fun)** `query/3` makes a synchronous call to the specified ATS which in turn makes a synchronous call to the specified transaction `Ref` and performs `Fun` on the state of `Ref` and returns the result to the original caller. The state of `Ref` is not updated in any way.

If the function fails in any way, then the transaction is aborted and `aborted` is returned to the caller.

If `Ref` is an already aborted transaction or non-existing transaction then `aborted` is returned.

**update_t(AT, Ref, Fun)** `update_t/3` makes an asynchronous call to the ATS and further on to the specified transaction `Ref` to update its state with the result from calling `Fun` with its current state. As the call is asynchronous `update_t/3` always returns immediately with `ok` and the caller cannot be sure wether it was a successful update or not. If the function fails then the transaction gets aborted and any future query or commit calls to that transaction `Ref` will forever return `aborted`.

If the transaction was already aborted or the given `Ref` does not exist then the update call is ignored.

**commit_t(AT, Ref)** To commit a transaction `commit_t/2` is used. It takes a process id `AT` of an ATS, a transaction reference `Ref` and tries to commit that transaction, i.e. update the state of the ATS with the state of `Ref`. If the given transaction has been aborted or does simply not exist, then `aborted` is returned.

If the commit was successful then the state of the ATS is updated to be the current state of the given transaction, followed by all current transactions being aborted, including the one that was committed. Meaning that with every successful commit every transaction is aborted, and all future action on any previous transaction will return aborted, or ignored if the action is `update_t/3`, and the ATS is at a 'clean' state. The assignment text specifies several times that all the *other* transactions must aborted when doing a successful commit, but it also states that committing an already successfully committed transaction must return `aborted`, inclining that this transaction has been aborted or will be aborted no matter what, and therefore I choose that it must be aborted after it has been committed.

I have chosen to make `commit/2` a synchronous call to the ATS to let it be possible to return the proper return value to the caller, and to lock the master while he is committing to try and disallow any outside interference.

**get_pids(AT)** `get_pids/1` is an interface function that I have added for the sake of testing, it makes a synchronous call to a given ATS and returns a list of process id over all the living processes that the given ATS is managing, including it self.

### 4.3.2 Extended API

All the four functions of the extended API are blocking functions.

**abort(AT, Ref)** This function forces a transaction `Ref` to be aborted. I have implemented this functionality by querying the transaction with a function that always raises an error, as shown below in Figure 5.

```
abort(AT, Ref) ->
    at_server:query_t(AT,Ref,fun(_) -> error(force_abort) end).
```

Figure 5: My implementation of the Erlang function `abort/2`.

This forces the query function to fail and cause the transaction to be aborted.

By using `query_t/3` I also cause the `abort/2` function to block, i.e. by making a synchronous call to the server. I could have used `update_t/3` with the same error-raising-function, and allow `abort/2` to return immediately, this would though open up for the possibility that we could try and query, update or even commit the transaction successfully due to the nature of the message order, before our error-function is run and aborts the transaction. By using `query_t/3` I can guarantee the caller that when `abort/2` returns the transaction is indeed aborted.

I have defined the return value of `abort/2` to be the return value of `query_t/3`, which implies that `abort/2` can only return `aborted`.

**tryUpdate(AT, Fun)** As the name implies; `tryUpdate/2` tries to update a specified ATS with the given function `Fun`, if the `Fun` fails then error is returned but if a transaction is successfully updated with `Fun` then the return value of `tryUpdate/2` is the return value of the commit attempt.

To allow `tryUpdate/2` to return `error` if `Fun` fails I start by querying the spawned transaction and checking the result. Then to avoid performing `Fun` twice, as it may be expensive, the provided update function to the transaction is simply just a function that does not care about the previous state and merely returns the result from the query. Although this does add a slight unnecessary overhead of transporting the result from the query back and forth, but allows us to easily return `error` when `Fun` fails.

The tryUpdate/2 function starts by beginning a transaction within AT, then does the query and if it succeeds then the transaction is updated and at last committed.

**ensureUpdate(AT, Fun)** `ensureUpdate/2` takes a pid of an ATS and a function `Fun` and promises to update the ATS with the function `Fun` and return `ok`, except if `Fun` fails, then `error` is returned.

I felt that the assignment text was not clear of which state `ensureUpdate/2` should update. As one can easily imagine the case of someone else making a commit before `ensureUpdate/2` is able to actual update a transaction and performing a commit, forcing the function to try again. But which state should `ensureUpdate/2` now update, the new state of the ATS or the initial state that the AST had when `ensureUpdate/2` was first called? I decided to go with the second case, as I felt that this would make most sense. Although this approach would mean that we have to rollback those commits that snuck in while we were trying to update the state of the ATS, but this also allows us to only compute the function `Fun` one time instead of recalculating it whenever we have to try again, because someone else made a commit before us, and if `Fun` is expensive; this could go on for a very long time.

`ensureUpdate/2` starts by beginning a new transaction `T` and then query `T` with `Fun`, if the result is aborted then I conclude that `Fun` failed and return `error`. If some new state `S` was returned start a loop function `ensureLoop/2` that will keep trying to update the state of the ATS to be `S`, until it succeeds. This ensures that it was the initial state of the ATS that is updated and not the in-between committed states.

**choiceUpdate(AT, Fun, Val_list)** `choiceUpdate/3` takes, as always, a pid for an ATS, a function `Fun(State,E)` and a list of element `Val_list`, where `State` is the state of the ATS and `E` is an element from `Val_list`. `choiceUpdate/3` then tries for each element in `Val_list` to update the state of the ATS with `Fun` in parallel. The first one to finish its update function is also the one that `choiceUpdate/3` tries to commit and the return value is then the return value from that commit. If all update functions fail then `error` is returned.

It is assumed that `Val_list` is indeed a list.

The function starts by beginning an amount of transactions equal to the number of elements in `Val_list`. Then foreach of these transactions they are asked to update their state with `Fun` and the respective element `E`. The update function has been altered to include a try and catch around `Fun`, which makes sure to send the correct message back to us. We then enter a loop function called `choiceLoop/2` where we wait for the first transaction to succeed its update function or that all transactions failed to update their state.

An important and somewhat trivial aspect of `choiceUpdate/3` is that all the transactions must have been begun before any one of them starts updating otherwise one transaction might have finished its update function before another had even been created. Therefore all are begun before any one is asked to update, to ensure a greater fairness to each transaction. Of course running through the list and asking each to update one by one, also favors the first transactions in the list, but this is how far the fairness is possible to stretch in this case.

Parallelism is achieved by utilizing that `update_t/3` is called asynchronously, and that each transaction runs in its own process.

The assignment text specifies that `choiceUpdate/3` must return the value of the commit of the first transaction to update successfully, but this is not possible if none update succeeds. Instead of letting the function hang and wait forever for some transaction to update successfully, I have added the try-catch inside the update function to be able to let `choiceUpdate/3` know if some transaction was aborted and ultimately allowing it to know if all fail and then return `error`.

There is also the slight chance that someone else makes a commit before any of our transactions gets through, `choiceUpdate/3` will still could the commit with the first successful transaction to update, if any, but as implied the return value would then be `aborted`.

As one may have noticed if glanced upon my implementation, I associate an unique reference to each call to choiceUpdate/3 this is because in a previous call to to choiceUpdate/3 one of the transactions may be still working on its update function, and if it finishes while a future choiceUpdate/3 call is waiting for responds from its own transactions then this old one while interfere and send 'false' messages to the choiceUpdate/3, by associated a unique reference to each session allows the function to discard any old obsolete messages.

### 4.3.3 Callback functions

The `format_status/2` function is not used, and as it is an optional callback function I have not implemented it.

Some of the callback functions are rather big and mixed with callbacks to both the ATS, transactions and both, therefore when describing some of these I will divide them into the three cases ATS, Transaction and Both equal to their separation in the actual file *at_server.erl*, to try and make it more readable.

**init(Args)** The `init/1` function is called by the *gen_server* when an atomic server or a transaction must be spawned. It is used to set the init state of the created server.

When an ATS is being created from the `start/1` interface function, then I have wrapped the initial given `State` in a tuple with the atom `server` as the first element to know this in the init function and set the init pool size to be an empty list.

If a new transaction is being created by the ATS then the atom `transaction` is passed along in the same fashion as `server` and the initial status is set to `ready`.

**handle_call/3** `handle_call/3` is called by the *gen_server* when a synchronous request is send to the server. Here I have split the description into the three parts.

**ATS** If the `stop_at_server` request is send to an ATS then it propagates the stop request to all the transactions that it manages, before terminating it self with the reason `normal`, and the reply `{ok,State}` where `State` is the current state of the ATS.

When a `{doquery_t, {Ref, Fun}}` request is received then the ATS looks `Ref` up in its table of transactions, if it exist and has the status `ready` then we try to query the transaction with the given `Fun`, if it returns `error` then we remove it from our pool of transactions or mark it state to be `aborted` and reply `aborted`, if not then we return the result. If the look up failed then we also reply `aborted`.

If an ATS received a `begin_t` query then it creates a unique reference and then tries to find an idle transaction within its pool, which it then initializes to be an fresh and ready transaction to be used. If no transaction was found, then a new one is started/spawned and the ATS adds it to its pool, at last the unique reference is then replied to the caller.

When `{commit_t, Ref}` is received then the ATS starts by look up in its pool if the `Ref` is present and in `ready` status, if it is found then the transaction is queried for its state `S` and if the transaction has not been aborted then the state of the ATS is updated to be `S`, and all transactions are either killed or given the `idle` status. `aborted` is replied if the look up failed or if the `doquery` request returned `error`.

If the `get_pid` request is received then the ATS replies with all process ids found in its pool and includes its own.

**Transaction** A transaction can receive an `initialize` request along with some `InitState` value, this forces the transaction to update its entire state data with the value of `InitState`. The transaction can also receive a `stop_at_trans` request which forces the transaction to stop, i.e. shutdown with the reason `normal`. If the transaction receive a `doquery` request but has the aborted status then it ignores the request and replies with `aborted`. The case where it does not have the `aborted` status is described below.

**Both** Both an ATS and a transaction can receive a `{doquery, Fun}` request and if so then `Fun` is computed with the current state and the result is replied back to the sender, if the function fails then we catch this and reply with `error` instead. We do not update the state of the ATS or transaction.

I have added a generic case to make sure that any unexpected request is simply just ignored instead of crashing the server.

**handle_cast/2** When ever an asynchronous request is sent to the server then the `handle_cast/2` callback is called by the OTP.

**ATS** When an ATS received the request `{update_t, {Ref, Fun}}` it will try to look up the given `Ref` in its pool of transactions if it exists and is `ready`, meaning that it has not been aborted before and do exist, then the ATS sends an asynchronous update request to the transaction with the given function `Fun`. The state and pool of the ATS is not altered in any way.

**Transaction** If an aborted transaction receives an update request `{update, Fun}` then the request is just ignored. If it is in the `ready` status then it tries to perform the update and if the function `Fun` fails then its state is not updated but its status is set to `aborted`. If the function succeeds then the state is updated to be the result of the function and the status remains unchanged.

A transaction can also be asynchronously asked to stop with the `stop_at_trans` messages, and so it will.

**Both** Also here is a generic case added to make sure that any unexpected request is simply just ignored instead of crashing the server.

**handle_info(Info, State)** I do not utilize this function.

**terminate(Reason, State)** `terminate/2` is called by the *gen_ server* right before the process is killed allowing you to do some clean up before shutdown. As already mentioned I have decided to do this in `handle_call/3`. Although if the ATS is terminated for some unexpected reason then I must try to stop any ongoing transactions incase it is the ATS that is being terminated. I the reason is `normal` then I assume that everything is under control.

**code_change(OldVsn, State, Extra)** The function `code_change/3` is not used nor implemented to any extend beyond some mere default state as the *gen_ server* behavior expects this callback function to be implemented.

### 4.3.4 Helper functions

**ensureLoop(AT, Fun)** Takes a pid of an ATS and a function `Fun`. It begins a transaction, updates it with `Fun` and tries to commit. If the commit succeeds then it returns `ok`, if not then it will try again. The function assumes that `Fun` never fails, and will only stop looping when a commit was successful.

**choiceLoop(AT, AllTrans)** `choiceLoop/2` takes an ATS and a list of tuples `AllTrans` where it assumes that the first element each tuple is of type `{ok,Ref}` where `Ref` is some transaction reference. If `AllTrans` is empty then `error` is returned, otherwise it will listen for the messages `{R,done}`, `{R,error}` and `E`, where the first message indicates that `R` is done with its update function and must be committed, and the return value of this commit is returned to the caller. If the error message is received then the reference is removed from `AllTrans` and the loop is called once again. If some unexpected message `E` is received then this is discarded and the loop function is called recursively.

It is assumed that the first instance of `{R,done}` is also the first to update its state, even though this cannot be completely guaranteed due to the process communication.

# 5 Appendix A Tests

## 5.1 Question 1 Testing

Testing of the parser implemented in the file *SalsaParser.hs* is done through the module `Test_Parser` implemented in the file *Test_ Parser.hs*. I decided to only test my implementation through the interface of the `SalsaParser` module.

### 5.1.1 *Test_ Parser.hs*

This test file implements the module `Test_Parser` which exports the function `runAllTests()` and `runAllTestsWith(n)`, where n is the number of *QuickCheck* test cases, 100 is the default value. This test file uses both *QuickCheck* and *HUnit*.

The test file is divided into three parts; a *QuickCheck* part, testing the parser with a valid input string, a *HUnit* part testing parsing of invalid input strings and that the correct precedence and associativity is maintained and a the third part testing that the parsing of files works as intended.

**Part 1** The QuickCheck part defines a set of lists containing definitions of the different components of the Salsa grammar, not to be confused with the Salsa definitions described in the grammar. These are used when picking a random element to generate, when generating the different components. The generated test cases tend to get rather big and slow down the test I have therefore added a few more cases of "move" and "const" respectively to the lists commands and exprList to easily add a greater chance that the generated data won't grow to large.

The test checks that any[2] valid input string is parsed to the correct abstract syntax tree. When generator a test case then nothing is predefined, all values are random generated by `QuickCheck Gen` monad so everything is tested, although each value is restricted to be a valid value, and every command and expr is within parenthesis meaning that this test does not test for precedence or associativity of the operators.

So far the tests keeps succeeding every time I run it. Based on this I assume that it works for any valid input string, but not precedence and associativity.

**Part 2** This part defines a lot of HUnit tests. These are divided into two categories: eleven precedence cases and forty-one error cases.

As these are unit tests I will only test a small subset of input area in the individual tests, but based on these tests I assume that it also works for the whole area.

**Precedence** The first five precedence test cases shows that the operators '@', '||','+' and '-' are all left associative.

The next two show that '@' has higher precedence than '||'.

Test eight and nine show that '+' and '-' has the same precedence and therefore it is the order that matters, even though the end result will be the same as + and - in themselves are commutative.

The last two shows that '.' has higher precedence than both '+' and '-'.

[2] Of course not all possible input values but a random subset of the possible input each time the test is run.

**Error** The first test shows that an empty string cannot be parsed to anything and is therefore not a valid input string.

The next five tests that a sident cannot be used where a vident is expected.

Test case seven to eleven shows that a vident cannot be used in place of a sident.

The next four tests shows some of the invalid characters that cannot be used in a Salsa program.

The next two test cases tests that numbers cannot be used as vidents or sidents.

The following two tests shows that letters cannot be used in place of numbers.

Test case twenty to twenty-two show that not any parenthesis or brackets can be used when surrounding a command, group or point.

The next three test cases shows some of the unsupported operators that does not work for the Salsa language.

The next four tests shows that a reserved word nor a colour can be used in place of a sident.

The thirtieth test case shows that a colour must be one of the specified colour names.

Test next four test cases shows that with lacking white spaces you either get an error or an unexpected program.

Test number 35 shows that when defining a group the given list must be non-empty.

Test case number 36 to 38 show that the Salsa language is case-sensitive.

The next two test cases show that negative integers are not supported by the grammar.

The last test show that the above restriction can be bypassed by simply creating a expression that subtracts a number from a smaller number, which would produce a negative integer when executed.

All these tests return the expected result and I therefore assume that precedence and associativity works as intended and adheres to the grammar, furthermore that the parser returns expected error with a given invalid input string.

**Part 3** This third part contains a set of five unit tests for testing the `parseFile` function and that it works as intended, it works as intended when parsing the content of the file with `parseString` yields the same result as parsing the file with `parseFile`.

The first test shows that an empty file is parsed as expected.

The second and the third test shows that valid Salsa files are also parsed as expected.

The fourth test shows that a invalid Salsa file is parsed as expected.

The fifth test case show that when parsing a file that does not exist then it returns the expected IO exception. Note that if the file do exist for some reason then this test will acts as a test on the content instead.

Based on these few simple tests that returned the expected values, I assume that function `parseFile` works as intended.

**Conclusion** Furthermore based on the tests described above and that they all return the expected results, I assume that my grammar and my implementation of the `SalsaParser` module works as intended.

## 5.2 Question 2 Testing

To test my implementation of the Salsa interpreter defined in the module *SalsaInterp* I have implemented a QuickCheck test to do so in the file *Test_Interp.hs* which defines and implements the module *Test_Interp*. This module exports two functions `runAllTestsI()` and `runAllTestsIWith(n)`. Each run the same QuickCheck test but `runAllTestsIWith(n)` allows you to specify the number of test cases to run. The default is set to 100.

I am aware that *hlint* spits out a hint of "Reduce duplication" but as in the other files I believe it would hurt the readability to resolve it.

### 5.2.1 *Test_Interp.hs*

As I can assume that the interpreter should not concern much of errors and wrong input, then this will not be a focus in my test.

My test file is divided into two sections; a *QuickCheck* part and a *HUnit* part, I therefore find it adequate to divide my test description into two parts as well.

**Part 1** I have defined a type `TestAnimation` which becomes an instance `QuickCheck.Arbitrary` and generates a valid Salsa Program `p` along with the expected Animation output from interpreting `p`. I do not intend to go in depth of the generator functions nor helper functions here, instead I want to describe my results.

The generated programs do not contain any Par commands and do not generate At commands that specify a group identifier. It does neither use Xproj or Yproj as expressions.

With that said then it works rather and and returns the expected results with every run.

**Part 2** This part is a set of *HUnit* test cases that tests the missing functionality from **Part 1**.

The first set of tests tests the `Par` command. The case is very simple yet the expected result is returned.

The second case tests that the `At` command and `Group` definition works in the same program.

The third case tests that the `Xproj` and `Yproj` expressions work as intended.

The fourth set of tests show that the interpolate function returns the expected results.

**Conclusion** As my *QuickCheck* property holds for all Salsa `Programs` without the `Par` command at `At` commands with group identifiers I assume that that part of my Salsa interpreter works as intended. Furthermore based on the unit tests that try to cover what *QuickCheck* left behind, which are somewhat scraped to a very few number of tests due to a approaching deadline, I (almost) dare to assume that the rest of my Salsa interpreter also works as intended.

## 5.3 Question 3 Testing

The following is my tests for the atomic transaction server. I have created a single file *test_at_server.erl* and module *test_at_server* that tests the interface functions of the server and the extra API functions. The module exports the function `runTests/0` that runs all the tests.

I decided to let each test section focus on a single interface/API function at a time, and I will use the same distinction when describing my tests here.

I assume that no one tries to manipulate the servers nor transactions while I test them.

### 5.3.1 *test_at_server.erl*

**testStart()** This function test the functionality of `start/1` with two test cases.

The first test shows that we can start a server with some partly-random state value and that this is alive.

The second test shows that multiply servers can be started as expected.

Based on the two test cases I assume that my implementation of `start/1` works as intended.

**testBegin()** `testbegin/0` test the function `begin_t/1` with four test cases.

The first test shows that only one process is alive at start up.
Then the second test case tests that the number of living processes increase with the amount of transactions begun.
Test three shows that a transaction created with an ATS `A` cannot be used within another ATS.
The fourth test shows that the state of a not-updated transaction is the same as the ATS.

Based on this I assume that `begin_t/1` works as intended.

**testStop()** Here we test that `stop/1` returns all processes within a given ATS is stopped.

The first test tests that an ATS with no ongoing transactions is stopped after calling the stop function.
The second test shows that an ATS with multiply ongoing transactions are also stopped as expected if calling `stop/1`.

Based on these two simple test cases I assume that my implementation of `stop/1` works as intended.

**testDoquery()** Here we test the `doquery/2` API function. The intended functionality is that when provided with a function `F` it returns the same value as if `F` was run locally on the same state, that it does not update the state of the ATS and that if the function fails `error` is returned and the process is unchanged.

The first two test cases show that the query with a function `F` returns the same as running `F` locally.
The third test shows that the state of the ATS has not been changed.
The last test tests that if queried with some error-prone function then `error` is returned and the ATS remains unchanged.

Based on these test I assume that `doquery/2` works as intended.

**testQuery_t()** Here I test the `query_t/3` function with seven test cases. I expect that if queried with a function `F` then the result is the same as evaluating `F` locally, that `query_t/3` does not update the state of the transaction except if it fails, in which case would return aborted and the transaction remains aborted.

The first two test cases tests that if queried with a function `F` then the result is the same as using `F` locally.
The third case shows that the state of the queried transaction is unchanged.
The fourth case shows that if run with a function that will fail then `aborted` is returned as expected.

Test case five then tests that the transactions remains aborted if queried again.

Test six tests that if another transaction was started earlier then this remains intact.

The seventh test case tests that if a query is sent to some non-existing transaction then `aborted` is returned as expected.

All seven test cases works as expected and based on this I assume that my implementation of `query_t/3` works as intended.

**testUpdate_t()** With five test cases I test that my implementation of `update_t/3` works as intended.

The first test case tests that if we update some transaction with a function `F` then its state is updated to be the result of `F` and this works as expected.

The second test that if the given function fails then the transaction is aborted.

The third case test that if trying to update an already aborted transaction yields no change to the status of the transaction.

The fourth test tests that even though one transaction has been aborted then previous begun transactions are still working.

The fifth and last test case show that if calling update with a unrecognized transaction reference then no change has happened.

The above five tests return the expected results and I therefore assume that my implementation of update_t/3 works as intended.

**testCommit_t()** This function tests my implementation of `commit_t/3` with six test cases.

The first test case tests that you do not need to update a transaction before commit it as expected the commit is successful and the state of the ATS remains the same.

The second test case show the transaction used before has now been aborted, and that trying to commit that transaction again returns `aborted` and it remains aborted.

Third test case tests that if a transaction is updated before being committed then the updated state becomes the new state of the ATS.

The fourth test case show that if the provided update function failed then when trying to commit the transaction `aborted` is returned.

The fifth test case tests that we can have several transactions going and if one is successfully committed then all are aborted and the state of ATS is updated to the expected one.

The last test case tests that `aborted` is returned if trying to call `commit_t/3` with a wrong transaction reference.

All six test cases return the expected values and based on this I assume that my implementation of `commit_t/3` is correct and works as intended.

**testAbort()** `abort/2` is tested in this function with five test cases.

The first test case tests that `aborted` is returned if calling `abort/2` on some transaction that has been started.

The second test show that `aborted` is returned if calling `abort/2` on some transaction that has been aborted previously.

The third show that several ongoing transactions can be aborted without trouble.

The fourth test case tests that no one else is affected when one is aborted.

The fifth test case tests that `aborted` is also returned if called with an unknown transactions reference.

All of the five test cases return the expected results and therefore I assume that `abort/2` work as intended.

**testTryUpdate()** This function tests the functionality of my implementation of `tryUpdate/2` with four test cases.

The first one shows that if no other transaction has been started then `tryUpdate/2` successfully updates the given ATS.

The second test case shows that if the function fails then error is returned and the state of the ATS remains unaltered.

The third case tests that if other transactions have been begun, then these are aborted if `tryUpdate/2` succeeds.

The last case tries to test that if someone else gets to sneak a commit in while `tryUpdate/2` is trying to update the state, then `aborted` is returned. Note that I wrote 'tries to' as this is very hard to test due to my implementation of `tryUpdate/2` as clearly a transaction who is waiting only *500ms* should be able to get his commit through before `tryUpdate/2` who waits *3500ms*. Since `tryUpdate/2` blocks the ATS by using `query_t/3` to do the computation, disallowing the previous transaction to commit after *500ms*. It seems like that once the query is done, then the other transaction is not allowed to commit as `tryUpdate/2` is allowed to commit unbothered.

Even though the last test case is hard to set up, I believe that my implementation works as intended as it can either return `error` or the return value from the commit statement which can be `ok` or `aborted`, as shown earlier, and hence `tryUpdate/2` will also return `aborted` in the case that the commit returns `aborted`. With this in mind and the fact that the other test cases returned the expected result I assume that my implementation of `tryUpdate/2` works as intended.

**testEnsureUpdate()** As my naming convention might have revealed by now, this function tests my implementation of `ensureUpdate/2`, and does so with three test cases.

The first case tests that if no other transaction has been begun, `ensureUpdate/2` returns `ok` and the state of the ATS is updated accordingly.

The second case tests that if the given function fails then `error` is returned and the state of the ATS remains unchanged.

The third test case has the intention to show that if someone gets to commit before `ensureUpdate/2`, but after it has begun trying, then it will try again, and eventually rollback the other commit. But it doesn't work presumably of the query call in `ensureUpdate/2`.

Based on these three simple test cases I assume that `ensureUpdate/2` works as expected.

**testChoiceUpdate()** My last test function tests the functionality of the `choiceUpdate/3` function with six test cases.

The first case tests that no other transactions has been begun and `choiceUpdate/3` is run with a single element, then this gets through, `ok` is returned and the AST has been updated.

The second test case tests that if all choices fail with their function except one, then this one gets through, `ok` is returned and the AST is correctly updated.

The third case tests that the shorter function of two is the one who is chosen and gets to be committed to be AST. Although due to the internal workings of the OS, Erlang environment and the alike, I cannot guarantee that this case will always return the expected result, but I do dare to say that a process waiting *500ms* should only in the most strangest cases be allowed to (busy)wait all *500* before a process only waiting *1ms* gets scheduled to run and eventually make its commit.

The fourth test case tests that if someone else gets to commit before any of the choices have successfully updated their state then the return value is `aborted` and `choiceUpdate/3` does not get to update the AST.

The fifth case shows that if all choices fail then `error` is returned.

The sixth and last test case tests that if the given list is empty then `error` is returned.

All the above functions return the expected result and based on this I assume that my implementation works as intended (big surprise).

**Conclusion** Apart from some anal edge cases that I cannot provide the correct environment for, all my tests return the expected results and behave as intended, and I therefore dare to assume once more that my implementation the the *at_ server* and *at_ extapi* modules work as intended.

# 6 Appendix B Grammar

## 6.1 G0

```
Program ::= DefComs .
DefComs ::= DefCom
    | DefCom DefComs .
DefCom ::= Command
    | Definition .
Definition ::= 'viewdef' VIdent Expr Expr
    | 'rectangle' SIdent Expr Expr Expr Expr Colour
    | 'circle' SIdent Expr Expr Expr Colour
    | 'view' VIdent
    | 'group' VIdent '[' VIdents ']' .
Command ::= SIdents '->' Pos
    | Command '@' VIdent
    | Command '||' Command
    | '{' Command '}' .
VIdents ::= VIdent
    | VIdent VIdents .
SIdents ::= SIdent
    | SIdent SIdents .
Pos ::= '(' Expr ',' Expr ')'
    | '+' '(' Expr ',' Expr ')' .
Expr ::= Prim
    | Expr '+' Prim
    | Expr '-' Prim .
Prim ::= integer
    | SIdent '.' 'x'
    | SIdent '.' 'y'
    | '(' Expr ')' .
Colour ::= 'blue' | 'plum' | 'red' | 'green' | 'orange' .
```

## 6.2 G1

```
Program ::= DefComs .
DefComs ::= DefCom
    | DefCom DefComs .
DefCom ::= Command
    | Definition .
Definition ::= 'viewdef' VIdent Expr Expr
    | 'rectangle' SIdent Expr Expr Expr Expr Colour
    | 'circle' SIdent Expr Expr Expr Colour
    | 'view' VIdent
    | 'group' VIdent '[' VIdents ']' .
Command ::= Command '||' Command2
    | Command2 .
Command2 ::= Command2 '@' VIdent
    | Command3 .
Command3 ::= SIdents '->' Pos
```

```
    | '{' Command '}' .
VIdents ::= VIdent
    | VIdent VIdents .
SIdents ::= SIdent
    | SIdent SIdents .
Pos ::= '(' Expr ',' Expr ')'
    | '+' '(' Expr ',' Expr ')' .
Expr ::= Prim
    | Expr '+' Prim
    | Expr '-' Prim .
Prim ::= integer
    | SIdent '.' 'x'
    | SIdent '.' 'y'
    | '(' Expr ')' .
Colour ::= 'blue' | 'plum' | 'red' | 'green' | 'orange' .
```

## 6.3 G2

```
Program ::= DefComs .
DefComs ::= DefCom DefComs' .
DefComs' ::= DefComs
    | e .
DefCom ::= Command
    | Definition .
Definition ::= 'viewdef' VIdent Expr Expr
    | 'rectangle' SIdent Expr Expr Expr Expr Colour
    | 'circle' SIdent Expr Expr Expr Colour
    | 'view' VIdent
    | 'group' VIdent '[' VIdents ']' .
Command ::= Command '||' Command2
    | Command2 .
Command2 ::= Command2 '@' VIdent
    | Command3 .
Command3 ::= SIdents '->' Pos
    | '{' Command '}' .
VIdents ::= VIdent VIdents' .
VIdents' ::= VIdents
    | e .
SIdents ::= SIdent SIdents' .
SIdents' ::= SIdents
    | e.
Pos ::= '(' Expr ',' Expr ')'
    | '+' '(' Expr ',' Expr ')' .
Expr ::= Prim
    | Expr Rest2 .
Op :: = '+' Prim
    | '-' Prim .
Prim ::= integer
    | '(' Expr ')'
    | SIdent '.' Rest3 .
Proj ::= 'x'
    | 'y' .
Colour ::= 'blue' | 'plum' | 'red' | 'green' | 'orange' .
```

## 6.4 G3

```
Program ::= DefComs .
DefComs ::= DefCom DefComs' .
DefComs' ::= DefComs
    | e .
DefCom ::= Command
    | Definition .
Definition ::= 'viewdef' VIdent Expr Expr
    | 'rectangle' SIdent Expr Expr Expr Expr Colour
    | 'circle' SIdent Expr Expr Expr Colour
    | 'view' VIdent
    | 'group' VIdent '[' VIdents ']' .
Command ::= Command2 Command' .
Command' ::= '||' Command2 Command'
    | e .
Command2 ::= Command3 Command2' .
Command2' ::= '@' VIdent Command2'
    | e .
Command3 ::= SIdents '->' Pos
    | '{' Command '}' .
VIdents ::= VIdent VIdents' .
VIdents' ::= VIdents
    | e .
SIdents ::= SIdent SIdents' .
SIdents' ::= SIdents
    | e.
Pos ::= '(' Expr ',' Expr ')'
    | '+' '(' Expr ',' Expr ')' .
Expr ::= Prim Expr' .
Expr' ::= Rest2 Expr'
    | e .
Op :: = '+' Prim
    | '-' Prim .
Prim ::= integer
    | '(' Expr ')'
    | SIdent Rest3 .
Proj ::= '.' 'x'
    | '.' 'y' .
Colour ::= 'blue' | 'plum' | 'red' | 'green' | 'orange' .
```

# 7 Appendix C Grammar assumptions

## 7.1 My assumptions on the grammar

All my assumptions for the grammar can be viewed Appendix D.

Some of the non-terminals in the grammar are not specified in the grammar, and only partly described in the assignment text. I will therefore list my assumptions and the definitions that I use in my implementation.

### 7.1.1 Case sensitivity

I assume that Salsa is case sensitive.

### 7.1.2 integer

Is a non-negative integer number and can therefore be written with the following regular expression:

$$[0-9]^+$$

### 7.1.3 VIdent

As the assignment text specifies then *VIdent* is a nonempty sequence of letters, digits and underscore, which starts with a uppercase letter and can therefore be written with the following regular expression:

$$[A-Z]^+[A-Za-z0-9\_]^*$$

### 7.1.4 SIdent

*SIdent* is the same as a *VIdent* except it cannot be one of the reserved words, described below, and has to start with a lowercase letter, which can be described as:

$$[a-z]^+[A-Za-z0-9\_]^*$$

### 7.1.5 White spaces

White spaces are what most would expect, spaces, any tabs and newlines. As I am using ReadP I will let the the function skipSpaces[3] define the exact representation of white spaces.

The reserved words, color names, *VIdent*, and *SIdent* are separated by at least one white space of any kind. Symbolic tokens are separated by 0 or more white spaces and so are symbolic tokens and alpha-numeric tokens from each other.

### 7.1.6 Reserved Words

The reserved words are: ’viewdef’, ’rectangle’, ’circle’, ’group’ and ’view’.

### 7.1.7 Color Names

The names of the color are also considered to be reserved words and are the following: ’blue’, ’plum’, ’red’, ’green’ and ’orange’.

[3] http://hackage.haskell.org/package/base-4.6.0.1/docs/Text-ParserCombinators-ReadP.html

# 8 Appendix D

## 8.1 Context functions

**createEmptyContext :: Integer -> Context** This initializes an empty context with the given Integer `n` and returns it, where `n` is the frame rate.

**lookupViews :: Ident -> Environment -> [Ident]** This functions takes an `Ident k` specifying either a `View` or a `Group` and an `Environment env` and tries to lookup `k` in `env` and returns the list of views the lookup returned.

The function assumes that the given `k` is the name of either a `View` or a `Group` and that it exists in the environment.

**bindCommand :: Ident -> [(Ident,Position)] -> State -> State** This function takes an `Ident k` and a list of `Ident`s and `Position`s `l` and maps `k` to `l` in a given `State` and returns the new `State`.

**addToState :: (State -> State) -> Context -> State** `addToState` takes a function `f` that takes a `State` and returns a new `State` and some `Context con` and returns the result from applying `f` to the `State` within `con`.

**bindDefinition :: Ident -> Definition -> Environment -> Environment** This functions binds an `Ident` to a given `Definition` into an given `Environment` and returns the updated `Environment`.

**addToEnvironment :: (Environment -> Environment) -> Context -> Context** This functions takes a function `f`, that does some computations on an `Environment`, a `Context con` and returns the updated `Context` from applying `f` to the `Environment` within `con`.

**updateActiveViews :: [Ident] -> Context -> Context** This function takes a list of `Ident`s and updates these to be the list of active views in a given `Context` and returns the updated `Context`.

**placeShapeInActiveViews :: Definition -> Context -> Context** This function takes a `Definition def` and a `Context` and updates the `State` within the given `Context` by mapping the given shape and its position in all the currently active views. The function actually works as a wrapper around `placeShapeHelper` that does the actual job, which has just been described.

`placeShapeInActiveViews` assumes that the given `def` is a shape definition, e.i. either rectangle or a circle.

## 8.2 Animation functions

**goToNextFrame :: Animation -> Animation** This function simply starts a new keyframe by appending an empty list to the back of the list of frames.

**addInstructions :: [GpxInstr] -> Animation -> Animation** This function adds a given list of instructions to the current keyframe. That is by appending the instructions to the last frame found in the last of frames.

**placeShapeInCurrentFrame :: Definition -> [ViewName] -> Animation -> Animation** This function generates instructions to draw a given shape in each view in a given list of views and add these instructions to the current keyframe. This function assumes that the given `Definition` is either a rectangle or circle. This function is a wrapper around the helper function `placeShapeFrameHelper` that does the described work.

**addViewToAnimation :: (ViewName, Integer, Integer) -> Animation -> Animation** `addViewToAnimation` adds a given view to the list of view definitions in a given `Animation` and returns this updated `Animation`.

# 9 Appendix E Code

## 9.1 at_server.erl

```
%%%-------------------------------------------------------------------
%%% Student name: Arni Asgeirsson
%%% Student KU-id: lwf986
%%%-------------------------------------------------------------------

-module(at_server).

-behaviour(gen_server).

% Interface functions
-export([start/1, stop/1, begin_t/1, doquery/2, query_t/3, update_t/3, commit_t/2]).
% Extra interface functions
-export([get_pids/1]).
% gen_server callback functions
-export([init/1,handle_call/3,handle_cast/2,handle_info/2,terminate/2,code_change/3]).

%% NOTE: I do no error_handling on these values,
%% therefore set them to anything other than true/false
%% and int values on your own risk
-define(MIN_POOL, true).
%% Default timeout value is 5000 ms for call/3
-define(TIME_OUT, 5000).

%%%-------------------------------------------------------------------
%%% API
%%%-------------------------------------------------------------------

%% I always assume that AT is a valid at_server process id, this is never
%% checked and if called with invalid value may result in unexpected error, behaviour or
%% and endless waiting for a never responding process.


start(State) ->
    gen_server:start(at_server, {server, State}, []).

%% call/2 is a synchronous call
stop(AT) ->
    tryCall(gen_server:call(AT,stop_at_server,?TIME_OUT)).

doquery(AT, Fun) ->
    tryCall(gen_server:call(AT,{doquery,Fun},?TIME_OUT)).

% Returns a reference
begin_t(AT) ->
    tryCall(gen_server:call(AT, begin_t,?TIME_OUT)).

query_t(AT, Ref, Fun) ->
    tryCall(gen_server:call(AT, {doquery_t, {Ref, Fun}},?TIME_OUT)).

%% Cast is the async requests
update_t(AT, Ref, Fun) ->
    gen_server:cast(AT, {update_t, {Ref, Fun}}).

commit_t(AT, Ref) ->
    tryCall(gen_server:call(AT,{commit_t, Ref},?TIME_OUT)).

%%% Extra API
```

```
%% Returns {ok, ListOfPids}
get_pids(AT) ->
    tryCall(gen_server:call(AT,get_pids,?TIME_OUT)).

%%%--------------------------------------------------------------------
%%% Callback functions
%%%--------------------------------------------------------------------

%%%--------------------------------------
%% Module:init(Args) -> Result
%% -----Types:
%% Args = term()
%% Result = {ok,State} | {ok,State,Timeout} | {ok,State,hibernate}
%%       | {stop,Reason} | ignore
%% State = term()
%% Timeout = int()>=0 | infinity
%% Reason = term()
%%%--------------------------------------

%% --------------------------------------
%% --------------- ATS ---------------
init({server, Args}) ->
    {ok,{Args,[]}};
%% --------------------------------------
%% ----------- Transaction -----------
init({transaction, Args}) ->
    {ok,{Args,ready}}.

%%%--------------------------------------
%% Module:handle_call(Request, From, State) -> Result
%% -----Types:
%% Request = term()
%% From = {pid(),Tag}
%% State = term()
%% Result = {reply,Reply,NewState} | {reply,Reply,NewState,Timeout}
%%       | {reply,Reply,NewState,hibernate}
%%       | {noreply,NewState} | {noreply,NewState,Timeout}
%%       | {noreply,NewState,hibernate}
%%       | {stop,Reason,Reply,NewState} | {stop,Reason,NewState}
%% Reply = term()
%% NewState = term()
%% Timeout = int()>=0 | infinity
%% Reason = term()
%%%--------------------------------------

%% I assume that no one will try and guess the pids of the transactions and send
%% them random messages or try and manipulate with them being going past the api functions

%% --------------------------------------
%% --------------- ATS ---------------
handle_call(stop_at_server, _, {State,Transactions}) ->
    stopAllTransactions(Transactions),
    {stop,normal,{ok,State},[]}; %% No reason to carry the state anymore
%% --------------------------------------
%% --------------- ATS ---------------
handle_call({doquery_t, {Ref, Fun}}, _, {State,Transactions}) ->
    {Reply,NewTransactions} =
  case lists:keyfind(Ref,1,Transactions) of
      {Ref,TrPid,ready} ->
    try gen_server:call(TrPid, {doquery, Fun},?TIME_OUT) of
        error ->
      case ?MIN_POOL of
```

```
          false ->
        {aborted, lists:keyreplace(Ref,1,Transactions,{Ref,TrPid,aborted})};
          true ->
        stopTransaction(TrPid),
        {aborted, lists:keydelete(Ref,1,Transactions)}
      end;
        Result -> {Result,Transactions}
    catch
        _:_ -> {timeout,Transactions}
    end;
      _ ->
    {aborted,Transactions}
  end,
    {reply, Reply, {State, NewTransactions}};
%% ---------------------------------------------
%% ------------------ ATS ------------------
handle_call(begin_t, _, {State,Transactions}) ->
    URef = make_ref(),
    NewTransactions =
  case lists:keyfind(idle,3,Transactions) of
      false ->
    {ok, TrPid} = gen_server:start(at_server, {transaction, State}, []),
    [{URef,TrPid,ready}|Transactions];
      {Ref,TrPid,idle} ->
    %% Make sure to update its state to be of ours
    try gen_server:call(TrPid,{initialize,{State,ready}},?TIME_OUT) of
        ok ->
      lists:keyreplace(Ref,1,Transactions,{URef,TrPid,ready})
    catch
        _:_ ->
      {ok, TrPid} = gen_server:start(at_server, {transaction, State}, []),
      [{URef,TrPid,ready}|Transactions]
    end
  end,
    {reply,{ok,URef},{State,NewTransactions}};
%% ---------------------------------------------
%% ------------------ ATS ------------------
handle_call({commit_t, Ref}, _, {State, Transactions}) ->
    {Reply, NewState, NewTransactions} =
  case lists:keyfind(Ref,1,Transactions) of
      {Ref,TrPid,ready} ->
    try gen_server:call(TrPid, {doquery,fun(I) -> I end},?TIME_OUT) of
        error ->
      {aborted,State,lists:keyreplace(Ref,1,Transactions,{Ref,TrPid,aborted})};
        {ok, NS} ->
      %% Abort all transactions now,
      %% ei set their state to idle
      %% Note that their state does not get 'cleaned up' this is done in begin_t
      case ?MIN_POOL of
          false ->
        NT = lists:map(fun({R,P,_}) -> {R,P,idle} end, Transactions),
        {ok,NS,NT};
          true ->
        stopAllTransactions,
        {ok,NS,[]}
      end
    catch
        _:_ -> {timeout,State,Transactions}
    end;
      _ ->
    {aborted,State,Transactions}
  end,
    {reply,Reply,{NewState,NewTransactions}};
%% ----------------------------------------------
%% ------------------- ATS ---------------------
handle_call(get_pids, _, {State, Transactions}) ->
    AllPids = [self()|lists:flatmap(fun({_,P,_}) -> [P] end, Transactions)],
    {reply, {ok, AllPids}, {State, Transactions}};
%% ----------------------------------------------
%% -------------- Transaction ------------------
handle_call({initialize,InitState}, _, _) ->
    {reply, ok, InitState};
%% ----------------------------------------------
%% -------------- Transaction ------------------
handle_call(stop_at_trans, _, {State, _}) ->
    {stop,normal,{ok,State},[]}; %% No reason to carry the state anymore
%% ----------------------------------------------
%% -------------- Transaction ------------------
handle_call({doquery,_}, _, {State, aborted}) ->
    {reply,error,{State,aborted}};
%% ----------------------------------------------
%% ------------------- Both --------------------
handle_call({doquery,Fun}, _, {State,Satalite}) ->
    Reply = try Fun(State) of
    Result -> {ok,Result}
      catch
    _:_ -> error
      end,
    {reply,Reply,{State,Satalite}};
%% ----------------------------------------------
%% ------------------- Both --------------------
handle_call(Msg,_,State) ->
    {reply,{unrecognized_message,Msg},State}.

%%%---------------------------------------------
%% Module:handle_cast(Request, State) -> Result
%% ----Types:
%% Request = term()
%% State = term()
%% Result = {noreply,NewState} | {noreply,NewState,Timeout}
%%      | {noreply,NewState,hibernate}
%%      | {stop,Reason,NewState}
%% NewState = term()
%% Timeout = int()>=0 | infinity
%% Reason = term()
%%%---------------------------------------------

%% ----------------------------------------------
%% ------------------- ATS ---------------------
handle_cast({update_t, {Ref, Fun}}, {State, Transactions}) ->
    case lists:keyfind(Ref,1,Transactions) of
  {Ref,TrPid,ready} ->
      gen_server:cast(TrPid,{update, Fun});
  _ ->
      do_nothing
    end,
    {noreply, {State, Transactions}};
%% ----------------------------------------------
%% -------------- Transaction ------------------
handle_cast({update, _}, {State, aborted}) ->
    {noreply,{State,aborted}};
handle_cast({update, Fun}, {State, ready}) ->
    NewState = try Fun(State) of
       Result -> {Result,ready}
```

```
         catch
       _:_ -> {State,aborted}
         end,
    {noreply,NewState};
handle_cast(stop_at_trans, State) ->
    {stop,normal,State};
%% -----------------------------------
%% -------------- Both --------------
handle_cast(_,State) ->
    {noreply,State}.

%%%-----------------------------------
%% Module:handle_info(Info, State) -> Result
%% ----Types:
%% Info = timeout | term()
%% State = term()
%% Result = {noreply,NewState} | {noreply,NewState,Timeout}
%%        | {noreply,NewState,hibernate}
%%        | {stop,Reason,NewState}
%% NewState = term()
%% Timeout = int()>=0 | infinity
%% Reason = normal | term()
%%%-----------------------------------

%% -----------------------------------
%% -------------- Both --------------
handle_info(_, State) ->
    {noreply,State}.

%%%-----------------------------------
%% Module:terminate(Reason, State)
%% ----Types:
%% Reason = normal | shutdown | {shutdown,term()} | term()
%% State = term()
%%%-----------------------------------

%% -----------------------------------
%% -------------- Both --------------
terminate(normal, _) ->
    ok;
%% -----------------------------------
%% -------------- ATS ---------------
terminate(Error, {State,[H|T]}) ->
    io:format(
      "#####Error: at_server with state: ~p~n"
      ++"#####Terminating due to some unexpected error: ~p!~n",[State, Error]),

    %% Try to shutdown each living transaction
    stopAllTransactions([H|T]);
%% -----------------------------------
%% ---------- Transaction ----------
terminate(Error, State) ->
    io:format(
      "#####Error: transaction with state: ~p~n"
      ++"#####Terminating due to some unexpected error: ~p!~n",[State, Error]),
    ok.

%%%-----------------------------------
%% Module:code_change(OldVsn, State, Extra) -> {ok, NewState} | {error, Reason}
%% ----Types:
%% OldVsn = Vsn | {down, Vsn}
%% Vsn = term()
%% State = NewState = term()
%% Extra = term()
%% Reason = term()
%%%------------------------------------------

%% The code_change/3 callback is not used and therefore not really implemented,
%% although present due to the expected callback exports

% ------------------------------------------
% ---------------- Both ----------------
code_change(_, State, _) ->
    {ok,State}.


%%%-----------------------------------------------------------------------------
%%% Helper server-functions
%%%-----------------------------------------------------------------------------

stopAllTransactions(Transactions) ->
    lists:foreach(fun({_,P,_}) -> stopTransaction(P) end, Transactions).

stopTransaction(Pid) ->
    try gen_server:call(Pid,stop_at_trans,?TIME_OUT)
    catch
  _:_ -> gen_server:cast(Pid,stop_at_trans)
    end.

tryCall(Call) ->
    try Call of
  Result ->
      Result
    catch
  timeout:_ ->
      timeout
    end.
```

## 9.2 at_extapi.erl

```
%%%-----------------------------------------------------------------------------
%%% Student name: Arni Asgeirsson
%%% Student KU-id: lwf986
%%%-----------------------------------------------------------------------------

-module(at_extapi).

-export([abort/2, tryUpdate/2, ensureUpdate/2, choiceUpdate/3]).

%%%-----------------------------------------------------------------------------
%%% Extended API
%%%-----------------------------------------------------------------------------

abort(AT, Ref) ->
    at_server:query_t(AT,Ref,fun(_) -> error(force_abort) end).

tryUpdate(AT, Fun) ->
    {ok,Ref} = at_server:begin_t(AT),
    %% By querying the transaction first, we can be sure the function returns an error
    %% to stick to the api, and if we do that then we do not need to recalculate that again
    %% although adding some overhead of transporting the data back and forth
    case at_server:query_t(AT,Ref,Fun) of
```

```
  {ok,State} ->
      %% No reason to evaluate the result again
      ok = at_server:update_t(AT,Ref,fun(_) -> State end),
      %% By now we either get a succesfull commit or got aborted
      %% because someone else made a commit before us.
      at_server:commit_t(AT,Ref);
  aborted ->
      error
    end.


ensureUpdate(AT, Fun) ->
    {ok,Ref} = at_server:begin_t(AT),
    case at_server:query_t(AT,Ref,Fun) of
  {ok,State} ->
      %% ensureLoop begins a new transaction, making R obsolete
      %% but it will be cleaned up with the next commit.
      ensureLoop(AT,fun(_) -> State end);
  aborted ->
      %% There is a slight change that someone made a commit before
      %% we were able to call query (not after we begun querying)
      %% and after we begun the transaction. This is accepted.
      error
    end.

ensureLoop(AT,Fun) ->
    {ok,R} = at_server:begin_t(AT),
    ok = at_server:update_t(AT,R,Fun),
    case at_server:commit_t(AT,R) of
  ok ->
      ok;
  aborted ->
      %% Ugh, we try again
      ensureLoop(AT,Fun)
    end.

%% Note that this does not 100% that the first is the one to
%% get through, (though it does locally) as the message queue is not guerenteed globally.
%% It is assumed that Val_list is indeed a list
choiceUpdate(AT, Fun, Val_list) ->
    AllTrans = lists:map(fun(E) -> {at_server:begin_t(AT),E} end, Val_list),
    Me = self(),
    URef = make_ref(),
    lists:foreach(fun({{ok,R},E}) ->
        ok = at_server:update_t(
         AT,
         R,
         fun(State) ->

           try Fun(State,E) of
               Res ->
             info(Me,{URef,R,done}),
             Res
           catch
               _:_ ->
             info(Me,{URef,R,error}),
             %% Remember to fail so its
             %% state is updated properly
             Fun(State,E)
           end
         end)
      end,
        AllTrans),
    choiceLoop(AT, AllTrans, URef).

%% Used by choiceUpdate
choiceLoop(_,[],_) ->
    error;
choiceLoop(AT, AllTrans, URef) ->
    %% Note that the messages are not guarenteed to arrive in the same order
    %% they are sent, therefore it could be that R is not the one who finished first
    %% but then again if R would sent the commit message himself, we still are not
    %% sure the someone won't skip in front of him.
    %% If we let the transaction to themself commit, we still need to let them
    %% send us a message to protect against the case of where all functions fail.
    receive
  {URef,R,done} ->
      at_server:commit_t(AT,R);
  {URef,R,error} ->
      RestTrans = lists:keydelete({ok,R},1,AllTrans),
      choiceLoop(AT, RestTrans, URef);
  _ ->
      choiceLoop(AT, AllTrans, URef)
    end.


%%%-------------------------------------------------------------------
%%% Communication primitives
%%%-------------------------------------------------------------------

%% asynchronous communication

info(Pid, Msg) ->
    Pid ! Msg.
```

## 9.3 test_at_server.erl

```
%%%-------------------------------------------------------------------
%%% Student name: Arni Asgeirsson
%%% Student KU-id: lwf986
%%%-------------------------------------------------------------------

-module(test_at_server).

-export([runTests/0]).

-define(SLEEP_TIME, 40).

%%%-------------------------------------------------------------------
%%% Interface
%%%-------------------------------------------------------------------

%% Run all tests
runTests() ->
    io:format("-------------------------------------------------------------------~n"),
    io:format("----------- Running tests for the at_server module -------------~n"),
    io:format("-------------------------------------------------------------------~n~n"),
    io:format("--------------------- Running Unit tests --------------------------~n"),
    io:format("Testing start/1:............"),
    io:format("~p~n",[testStart()]),
    io:format("Testing begin_t/1:.........."),
    io:format("~p~n",[testBegin()]),
    io:format("Testing stop/1:.............."),
    io:format("~p~n",[testStop()]),
    io:format("Testing doquery/2:..........."),
    io:format("~p~n",[testDoquery()]),
    io:format("Testing query_t/3:..........."),
    io:format("~p~n",[testQuery_t()]),
    io:format("Testing update_t/3:.........."),
    io:format("~p~n",[testUpdate_t()]),
    io:format("Testing commit_t/2:.........."),
    io:format("~p~n",[testCommit_t()]),
    io:format("Testing abort/2:............."),
    io:format("~p~n",[testAbort()]),
    io:format("Testing tryUpdate/2:........."),
    io:format("~p~n",[testTryUpdate()]),
    io:format("Testing ensureUpdate/2:......"),
    io:format("~p~n",[testEnsureUpdate()]),
    io:format("Testing choiceUpdate/3:......"),
    io:format("~p~n",[testChoiceUpdate()]).

%%%-------------------------------------------------------------------
%%% Test Functions
%%%-------------------------------------------------------------------

% ---------------------------------------
% ------------ ATS API ------------

%% Test start/1
testStart() ->
    %% Init test data
    StateA = [asd,"d",233],

    %% 1. Test that we can start a server with some state

    {ok, Pid1} = at_server:start([]),
    timer:sleep(?SLEEP_TIME),
    Test1 = assertEquals(true,isProcessAlive(Pid1)),

    %% 2. Test that we can start multiply servers

    {ok, Pid2} = at_server:start(Pid1),
    timer:sleep(?SLEEP_TIME),
    Test21 = assertEquals(true,isProcessAlive(Pid2)),

    {ok, Pid3} = at_server:start(StateA),
    timer:sleep(?SLEEP_TIME),
    Test22 = assertEquals(true,isProcessAlive(Pid3)),
    Test2 = areTrue([Test21,Test22]),

    %% Clean up
    {ok,[]} = at_server:stop(Pid1),
    {ok,Pid1} = at_server:stop(Pid2),
    {ok,StateA} = at_server:stop(Pid3),

    areTrue([Test1,Test2]).

%% Test begin_t/1
testBegin() ->
    %% Init test data
    State = some_state,
    {ok, Pid1} = at_server:start(State),
    {ok, Pid2} = at_server:start(State),
    timer:sleep(?SLEEP_TIME),

    %% 1. Test that only one process exist on start up

    {ok, AllPids} = at_server:get_pids(Pid1),
    Test1 = assertEquals(1,length(AllPids)),

    %% 2. Test that we can start transactions and these spawn a the correct amount
    %% of transactions/processes

    begin_transactions(Pid1,1),
    timer:sleep(?SLEEP_TIME),
    {ok, AllPids1} = at_server:get_pids(Pid1),
    Test21 = assertEquals(2, length(AllPids1)),

    begin_transactions(Pid1,5),
    timer:sleep(?SLEEP_TIME),
    {ok, AllPids3} = at_server:get_pids(Pid1),
    Test22 = assertEquals(7, length(AllPids3)),

    begin_transactions(Pid1,28),
    timer:sleep(?SLEEP_TIME),
    {ok, AllPids4} = at_server:get_pids(Pid1),
    Test23 = assertEquals(35, length(AllPids4)),

    Test24 = assertEquals(true, areProcessesAlive(AllPids4)),
    Test2 = areTrue([Test21, Test22, Test23, Test24]),

    %% 3. Test that we cannot create a transaction in A and use it in B
    {ok,R} = at_server:begin_t(Pid1),
    Test3 = assertEquals(true, isAborted(Pid2,R)),

    %% 4. Test that the state of a transaction is the same as the ATS
    Test4 = assertEquals(at_server:doquery(Pid1,fun identity/1),
        at_server:query_t(Pid1,R,fun identity/1)),

    %% Clean up
    {ok,State} = at_server:stop(Pid1),
    {ok,State} = at_server:stop(Pid2),

    areTrue([Test1, Test2, Test3, Test4]).

%% Test stop/1
testStop() ->
    %% Init test data
    State = some_state,
    {ok, Pid1} = at_server:start(State),
    {ok, Pid2} = at_server:start(State),
    timer:sleep(?SLEEP_TIME),

    %% 1. Test that an at_server dies after stop/1 has been called

    Test11 = assertEquals(true, isProcessAlive(Pid1)),
    Test12 = assertEquals({ok,State}, at_server:stop(Pid1)),
    timer:sleep(?SLEEP_TIME),
    Test13 = assertEquals(true, isProcessDead(Pid1)),
    Test1 = areTrue([Test11, Test12, Test13]),

    %% 2. Test that all initiated transactions are also stopped with the at_server

    begin_transactions(Pid2,4),
    timer:sleep(?SLEEP_TIME),
    {ok, AllPids} = at_server:get_pids(Pid2),
    Test21 = assertEquals(true, areProcessesAlive(AllPids)),
    {ok, State} = at_server:stop(Pid2),
    timer:sleep(?SLEEP_TIME),
    Test22 = assertEquals(true, areProcessesDead(AllPids)),
    Test2 = areTrue([Test21, Test22]),

    %% Clean up

    areTrue([Test1, Test2]).


%% Test doquery/2
testDoquery() ->
    %% Init test data
    State = "I am not an A",
    {ok, Pid1} = at_server:start(State),
    timer:sleep(?SLEEP_TIME),

    %% 1. Test doquery returns the state with an identity function
    Test1 = assertEquals({ok,State},
       at_server:doquery(Pid1,fun identity/1)),

    %% 2. Test that it returns what is returned by the given function
    Test21 = assertEquals({ok,mapMult2(State)},
        at_server:doquery(Pid1,fun mapMult2/1)),
    Test22 = assertEquals({ok,mapToA(State)},
        at_server:doquery(Pid1,fun mapToA/1)),
    Test2 = areTrue([Test21, Test22]),

    %% 3. Show that the doquery doesn't update the state data
    Test3 = assertEquals({ok,State},
       at_server:doquery(Pid1,fun identity/1)),

    %% 4. Test what happens if the function causes some error
    Test41 = assertEquals(error,
        at_server:doquery(Pid1,fun onlyEmpty/1)),
    Test42 = assertEquals(true, isProcessAlive(Pid1)),
    Test43 = assertEquals({ok,State},
        at_server:doquery(Pid1,fun identity/1)),
    Test4 = areTrue([Test41, Test42, Test43]),

    %% Clean up
    {ok,State} = at_server:stop(Pid1),

    areTrue([Test1, Test2, Test3, Test4]).

%% Test query_t/3
testQuery_t() ->
    %% Init test values
    State = [1,2,3,4,5,6],
    {ok,Pid1} = at_server:start(State),
    {ok,R1} = at_server:begin_t(Pid1),
    {ok,R2} = at_server:begin_t(Pid1),
    timer:sleep(?SLEEP_TIME),

    %% 1. Test that an unaltered trans state returns the initial state when used with identity
    Test1 = assertEquals({ok, State},
       at_server:query_t(Pid1,R1,fun identity/1)),

    %% 2. Test that it returns the same as when run on the state here
    Test21 = assertEquals({ok,mapMult2(State)},
        at_server:query_t(Pid1,R1,fun mapMult2/1)),
    Test22 = assertEquals({ok,mapToA(State)},
        at_server:query_t(Pid1,R1,fun mapToA/1)),
    Test2 = areTrue([Test21, Test22]),

    %% 3. Show that query_t doesnt update its state
    Test3 = assertEquals({ok, State},
       at_server:query_t(Pid1,R1,fun identity/1)),

    %% 4. Test what happens if the function causes some error
    Test41 = assertEquals(aborted,
        at_server:query_t(Pid1,R1,fun onlyEmpty/1)),
    Test42 = assertEquals(true, isProcessAlive(Pid1)),
    Test4 = areTrue([Test41, Test42]),

    %% 5. Show that aborted is also returned when trying to query it again (even with a valid functi
    Test5 = assertEquals(aborted,
       at_server:query_t(Pid1,R1,fun identity/1)),

    %% 6. Test that even though R1 is aborted R2 is still good
    Test6 = assertEquals({ok,State},
       at_server:query_t(Pid1,R2,fun identity/1)),

    %% 7. Test that a wrong ref_id is considered to be an aborted transaction
    WrongRef = make_ref(),
    Test71 = assertEquals(aborted,
        at_server:query_t(Pid1,WrongRef,fun identity/1)),
    Test72 = assertEquals(aborted,
        at_server:query_t(Pid1,also_wrong,fun identity/1)),
    Test7 = areTrue([Test71, Test72]),

    %% Clean up
    {ok, State} = at_server:stop(Pid1),

    areTrue([Test1, Test2, Test3, Test4, Test5, Test6, Test7]).

%% Test update_t/3
testUpdate_t() ->
    %% Init test values
    State = [1,2,3,4,5,6],
    {ok,Pid1} = at_server:start(State),
    {ok,R1} = at_server:begin_t(Pid1),
    {ok,R2} = at_server:begin_t(Pid1),
    ok = at_server:update_t(Pid1,R2,fun removeEven/1),
    {ok,R3} = at_server:begin_t(Pid1),
    timer:sleep(?SLEEP_TIME),

    %% 1. Test that if we update it then it contains the new data
    Test11 = assertEquals({ok,State},
        at_server:query_t(Pid1,R1,fun identity/1)),
    ok = at_server:update_t(Pid1,R1,fun removeEven/1),
    timer:sleep(?SLEEP_TIME),
    Test12 = assertEquals({ok,removeEven(State)},
        at_server:query_t(Pid1,R1,fun identity/1)),
    Test1 = areTrue([Test11, Test12]),

    %% 2. Test what happends if the update function fails
    %% I.e. Show that it is aborted
    ok = at_server:update_t(Pid1,R1,fun onlyEmpty/1),
    timer:sleep(?SLEEP_TIME),
    Test2 = assertEquals(true, isAborted(Pid1,R1)),

    %% 3. Test calling update on a aborted transaction
    ok = at_server:update_t(Pid1,R1,fun removeEven/1),
    timer:sleep(?SLEEP_TIME),
    Test3 = assertEquals(true, isAborted(Pid1,R1)),

    %% 4. Show that even though it is aborted R2 & R3 still maintain their state and are fully funct
    Test41 = assertEquals({ok,removeEven(State)},
        at_server:query_t(Pid1,R2,fun identity/1)),
    Test42 = assertEquals({ok,State},
        at_server:query_t(Pid1,R3,fun identity/1)),

    ok = at_server:update_t(Pid1,R3,fun removeEven/1),
    timer:sleep(?SLEEP_TIME),
    Test43 = assertEquals({ok,removeEven(State)},
        at_server:query_t(Pid1,R3,fun identity/1)),
    Test4 = areTrue([Test41, Test42, Test43]),

    %% 5. Test what happens with a wrong ref_id
    {ok, AllPids} = at_server:get_pids(Pid1),
    Test51 = assertEquals(true, areProcessesAlive(AllPids)),
    ok = at_server:update_t(Pid1,wrong_ref,fun removeEven/1),
    timer:sleep(?SLEEP_TIME),
    Test52 = assertEquals(true, areProcessesAlive(AllPids)),
    Test53 = assertEquals({ok,removeEven(State)},
        at_server:query_t(Pid1,R3,fun identity/1)),
    Test5 = areTrue([Test51, Test52, Test53]),

    %% Clean up
    {ok,State} = at_server:stop(Pid1),

    areTrue([Test1, Test2, Test3, Test4, Test5]).

%% Test commit_t/2
testCommit_t() ->
    %% Init test values
    StateA = [1,2,3,4,5,6,7,8,9,10],
    StateB = removeEven(StateA),
    {ok,Pid1} = at_server:start(StateA),
    {ok,R1} = at_server:begin_t(Pid1),
    timer:sleep(?SLEEP_TIME),

    %% 1. Test that after a commit without first doing a update the state is still the same
    %% And that it is still treated as a commit, ie the process is aborted
    Test11 = assertEquals({ok,StateA},
        at_server:doquery(Pid1, fun identity/1)),
    Test12 = assertEquals(ok, at_server:commit_t(Pid1,R1)),
    Test13 = assertEquals({ok,StateA},
        at_server:doquery(Pid1, fun identity/1)),
    Test1 = areTrue([Test11, Test12, Test13]),

    %% 2. Test that after a commit the transactions are all aborted
    Test21 = assertEquals(true, isAborted(Pid1,R1)),
    Test22 = assertEquals(aborted, at_server:commit_t(Pid1,R1)),
    Test23 = assertEquals(true, isAborted(Pid1,R1)),
    Test2 = areTrue([Test21, Test22, Test23]),

    %% 3. Test that the state changes to the correct value after a commit and update
    {ok,R2} = at_server:begin_t(Pid1),
    Test31 = assertEquals({ok,StateA},
        at_server:doquery(Pid1, fun identity/1)),
    ok = at_server:update_t(Pid1,R2, fun removeEven/1),
    timer:sleep(?SLEEP_TIME),
    ok = at_server:commit_t(Pid1,R2),
    Test32 = assertEquals({ok,StateB},
        at_server:doquery(Pid1, fun identity/1)),
    Test3 = areTrue([Test31, Test32]),

    %% 4. Test if we try to commit after the update function have failed
    {ok,R3} = at_server:begin_t(Pid1),
    ok = at_server:update_t(Pid1,R3, fun onlyEmpty/1),
    Test4 = assertEquals(aborted, at_server:commit_t(Pid1,R3)),

    %% 5. Test that we can have several different transactions going at one time
    %% And that all are aborted when one is committed
    {ok,R4} = at_server:begin_t(Pid1),
    {ok,R5} = at_server:begin_t(Pid1),
    {ok,R6} = at_server:begin_t(Pid1),
    {ok,R7} = at_server:begin_t(Pid1),
    timer:sleep(?SLEEP_TIME),

    Mult2 = fun(NS) -> lists:map(fun(N) -> N*2 end,NS) end,
    Mult4 = fun(NS) -> lists:map(fun(N) -> N*4 end,NS) end,
    Mult8 = fun(NS) -> lists:map(fun(N) -> N*8 end,NS) end,

    ok = at_server:update_t(Pid1,R4, Mult2),
    ok = at_server:update_t(Pid1,R5, Mult4),
    ok = at_server:update_t(Pid1,R6, fun onlyEmpty/1),
    ok = at_server:update_t(Pid1,R7, Mult8),
    timer:sleep(?SLEEP_TIME),

    Test51 = assertEquals({ok,Mult2(StateB)},
        at_server:query_t(Pid1,R4,fun identity/1)),
    Test52 = assertEquals({ok,Mult4(StateB)},
        at_server:query_t(Pid1,R5,fun identity/1)),
    Test53 = assertEquals(aborted,
        at_server:query_t(Pid1,R6,fun identity/1)),
    Test54 = assertEquals({ok,Mult8(StateB)},
        at_server:query_t(Pid1,R7,fun identity/1)),

    ok = at_server:commit_t(Pid1,R5),
    StateC = Mult4(StateB),
    Test55 = assertEquals({ok,StateC},
        at_server:doquery(Pid1, fun identity/1)),
    Test56 = assertEquals(true, isAborted(Pid1,R4)),
    Test57 = assertEquals(true, isAborted(Pid1,R5)),
    Test58 = assertEquals(true, isAborted(Pid1,R6)),
    Test59 = assertEquals(true, isAborted(Pid1,R7)),
    Test5 = areTrue([Test51,Test52,Test53,Test54,Test55,Test56,
         Test57,Test58,Test59]),

    %% 6. Test with wrong ref
    Test6 = assertEquals(aborted, at_server:commit_t(Pid1,wrong_ref)),

    %% Clean up
    {ok,StateC} = at_server:stop(Pid1),

    areTrue([Test1, Test2, Test3, Test4, Test5, Test6]).

%% ---------------------------------------------------------
%% -------------------- EXT API --------------------

%% Tests abort/2
testAbort() ->
    %% Init test data
    State = abcdef,
    {ok,Pid1} = at_server:start(State),
    {ok,R1} = at_server:begin_t(Pid1),
    {ok,R2} = at_server:begin_t(Pid1),
    {ok,R3} = at_server:begin_t(Pid1),
    {ok,R4} = at_server:begin_t(Pid1),
    {ok,R5} = at_server:begin_t(Pid1),
    timer:sleep(?SLEEP_TIME),

    %% 1. Test that the transaction is aborted
    Test11 = assertEquals(aborted, at_extapi:abort(Pid1,R1)),
    Test12 = assertEquals(true, isAborted(Pid1,R1)),
    Test1 = areTrue([Test11, Test12]),

    %% 2. Test what happens if calling aborted again
    Test2 = assertEquals(aborted, at_extapi:abort(Pid1,R1)),

    %% 3. Test that several can be aborted
    Test31 = assertEquals(aborted, at_extapi:abort(Pid1,R2)),
    Test32 = assertEquals(aborted, at_extapi:abort(Pid1,R3)),
    Test3 = areTrue([Test31, Test32]),

    %% 4. Test that no one else is affacted when one is being aborted
    Test41 = assertEquals({ok,State}, at_server:query_t(Pid1,R4,fun identity/1)),
    ok = at_server:update_t(Pid1,R5,fun atom_to_list/1),
    timer:sleep(?SLEEP_TIME),
    Test42 = assertEquals({ok,atom_to_list(State)}, at_server:query_t(Pid1,R5,fun identity/1)),
    Test4 = areTrue([Test41, Test42]),

    %% 5. Test what happens with a unknown ref id
    WrongRef = make_ref(),
    Test5 = assertEquals(aborted, at_extapi:abort(Pid1,WrongRef)),

    %% Clean up
    {ok,State} = at_server:stop(Pid1),

    areTrue([Test1, Test2, Test3, Test4, Test5]).

%% Tests tryUpdate/2
testTryUpdate() ->
    %% Init test data
    StateA = [1,2,3,4,5,6],
    StateB = removeEven(StateA),
    {ok,Pid1} = at_server:start(StateA),
    timer:sleep(?SLEEP_TIME),

    %% 1. Test that if no one else is doing a transaction we get our update through
    Test11 = assertEquals(ok, at_extapi:tryUpdate(Pid1,fun identity/1)),
    Test12 = assertEquals({ok,StateA}, at_server:doquery(Pid1, fun identity/1)),
    Test13 = assertEquals(ok, at_extapi:tryUpdate(Pid1,fun removeEven/1)),
    Test14 = assertEquals({ok,StateB}, at_server:doquery(Pid1, fun identity/1)),
    Test1 = areTrue([Test11, Test12, Test13, Test14]),

    %% 2. Test that if the function fails, no update happens and we get error returned
    Test21 = assertEquals(error, at_extapi:tryUpdate(Pid1,fun onlyEmpty/1)),
    Test22 = assertEquals({ok,StateB}, at_server:doquery(Pid1, fun identity/1)),
    Test2 = areTrue([Test21, Test22]),

    %% 3. Test that if others is doing a transaction they get aborted
    {ok,R1} = at_server:begin_t(Pid1),
    {ok,R2} = at_server:begin_t(Pid1),
    {ok,R3} = at_server:begin_t(Pid1),
    timer:sleep(?SLEEP_TIME),
    ok = at_server:update_t(Pid1,R2,fun onlyEmpty/1),
    ok = at_server:update_t(Pid1,R3,fun removeEven/1),
    timer:sleep(?SLEEP_TIME),

    Test31 = assertEquals(ok, at_extapi:tryUpdate(Pid1,fun identity/1)),
    Test32 = assertEquals(true, isAborted(Pid1,R1)),
    Test33 = assertEquals(true, isAborted(Pid1,R2)),
    Test34 = assertEquals(true, isAborted(Pid1,R3)),
    Test3 = areTrue([Test31, Test32, Test33, Test34]),

    %% 4. Test that if someone commits while we are trying to update we get aborted
    %% -> Really hard to do due to the implementation of tryUpdate/2
    {ok,R} = at_server:begin_t(Pid1),
    ok = at_server:update_t(Pid1,R,fun(_) ->
            T = expensive1(500),
            at_server:commit_t(Pid1,R),
            T end),
    Test4 = assertNotEquals(aborted,
          at_extapi:tryUpdate(Pid1,fun(_) -> expensive1(3500) end)),

    %% Clean up
    {ok,3500} = at_server:stop(Pid1),

    areTrue([Test1, Test2, Test3,Test4]).

%% Tests ensureUpdate/2
testEnsureUpdate() ->
    %% Init test data
    StateA = [1,2,3,4,5,6],
    StateB = removeEven(StateA),
    {ok,Pid} = at_server:start(StateA),
    timer:sleep(?SLEEP_TIME),

    %% 1. Test that if we are the only one here we get our update through
    Test11 = assertEquals(ok, at_extapi:ensureUpdate(Pid,fun identity/1)),
    Test12 = assertEquals({ok,StateA}, at_server:doquery(Pid, fun identity/1)),
    Test13 = assertEquals(ok, at_extapi:ensureUpdate(Pid,fun removeEven/1)),
    Test14 = assertEquals({ok,StateB}, at_server:doquery(Pid, fun identity/1)),
    Test1 = areTrue([Test11, Test12, Test13, Test14]),

    %% 2. Test that if the function fails we get error and nothing is updated
    Test21 = assertEquals(error, at_extapi:ensureUpdate(Pid,fun onlyEmpty/1)),
    Test22 = assertEquals({ok,StateB}, at_server:doquery(Pid,fun identity/1)),
    Test2 = areTrue([Test21, Test22]),

    %% 3. Test that if someone commits while we are trying to update we still get our
    %% commit through on the original state and the other is rolled backed
    {ok,R} = at_server:begin_t(Pid),
    ok = at_server:update_t(Pid,R,fun(_) ->
            expensive1(500),
            at_server:commit_t(Pid,R) end),
    Test3 = assertEquals(ok,
       at_extapi:ensureUpdate(Pid,fun(_) -> expensive1(3500) end)),

    %% Clean up
    {ok,3500} = at_server:stop(Pid),

    areTrue([Test1, Test2,Test3]).

%% Tests choiceUpdate/3
testChoiceUpdate() ->
    %% Init test data
    Val_listA = [1],
    Val_listB = [a,2,c],
    %% TODO timeout error happens if 500 is set to 5000+
    Val_listC = [500,1],
    Val_listD = [a,b,c],

    Add = fun(State,E) -> lists:map(fun(N) -> N+E end,State) end,
    StateA = [1,2,3,4,5],
    StateB = Add(StateA,lists:nth(1,Val_listA)),
    StateC = Add(StateB,lists:nth(2,Val_listB)),
    StateD = lists:nth(2,Val_listC),

    {ok,Pid} = at_server:start(StateA),
    timer:sleep(?SLEEP_TIME),

    %% 1. Test that if only one then that gets through
    Test11 = assertEquals(ok, at_extapi:choiceUpdate(Pid,Add,Val_listA)),
    Test12 = assertEquals({ok,StateB}, at_server:doquery(Pid,fun identity/1)),
    Test1 = areTrue([Test11, Test12]),

    %% 2. Test that if all fail except one, then that gets through
    Test21 = assertEquals(ok, at_extapi:choiceUpdate(Pid,Add,Val_listB)),
    Test22 = assertEquals({ok,StateC}, at_server:doquery(Pid,fun identity/1)),
    Test2 = areTrue([Test21, Test22]),


    %% 3. Test that a shorter function will be the one to come through rather than a long function
    %% Although this cannot be guerenteed!
    Test31 = assertEquals(ok, at_extapi:choiceUpdate(Pid, fun expensive/2, Val_listC)),
    Test32 = assertEquals({ok,StateD}, at_server:doquery(Pid,fun identity/1)),
    Test3 = areTrue([Test31, Test32]),

    %% 4. Test that if someoneelse commits before any of us, we get aborted when trying to commit
    {ok,R} = at_server:begin_t(Pid),
    Test41 = assertEquals(aborted,
        at_extapi:choiceUpdate(Pid,fun(_,_) ->
                   at_server:commit_t(Pid,R)
                 end,Val_listA)),
    Test42 = assertEquals({ok,StateD}, at_server:doquery(Pid, fun identity/1)),
    Test4 = areTrue([Test41,Test42]),

    %% 5. Test that if all fail then error is returned
    Test5 = assertEquals(error, at_extapi:choiceUpdate(Pid,Add,Val_listD)),

    %% 6. Test that if the list is empty
    Test6 = assertEquals(error, at_extapi:choiceUpdate(Pid,Add,[])),

    %% Clean up
    {ok,StateD} = at_server:stop(Pid),

    areTrue([Test1, Test2, Test3, Test4, Test5, Test6]).

%%%-------------------------------------------------------------------
%%% Helper Functions
%%%-------------------------------------------------------------------

areTrue([]) ->
    false;
areTrue(List) ->
    lists:foldl(fun(A,B) -> A andalso B end,true,List).

assertEquals(A,B) ->
```

```
    A == B.

assertNotEquals(A,B) ->
    A /= B.

isAborted(Pid,R) ->
    A1 = aborted == at_server:query_t(Pid,R, fun identity/1),
    A2 = aborted == at_server:commit_t(Pid,R),
    A1 andalso A2.

% Returns true if the given Pid is a running process
% otherwise false.
isProcessAlive(Pid) ->
    case process_info(Pid) of
  undefined ->
      false;
  _ -> true
    end.

isProcessDead(Pid) ->
    not(isProcessAlive(Pid)).

areProcessesAlive(Pids) ->
    lists:foldl(fun(P,B) -> isProcessAlive(P) andalso B end,true,Pids).

areProcessesDead(Pids) ->
    lists:foldl(fun(P,B) -> (isProcessDead(P)) andalso B end,true,Pids).

begin_transactions(A,N) ->
    case N > 0 of
  true ->
      {ok, R} = at_server:begin_t(A),
      [R|begin_transactions(A,N-1)];
  false -> []
    end.

%%%---------------------------------------------------------------------
%%% Update Functions
%%%---------------------------------------------------------------------

expensive(_,Time) ->
    receive
    after
  Time -> Time
    end.

expensive1(Time) ->
    expensive(Time,Time).

identity(X) ->
    X.

onlyEmpty([]) ->
    [].

mapToA(_) ->
    "A".

mapMult2(Ns) ->
    lists:map(fun(X) -> X*2 end,Ns).

removeEven(X) ->
```

```
    lists:filter(fun(N) -> N rem 2 /= 0 end, X).
```

## 9.4 Gpx.hs

```haskell
module Gpx where

type ViewName = String
type ColourName = String
type Frame = [GpxInstr]
type Animation = ([(ViewName, Integer, Integer)], [Frame])
data GpxInstr = DrawRect Integer Integer Integer Integer ViewName ColourName
              | DrawCirc Integer Integer Integer ViewName ColourName
              deriving (Eq, Show)
```

## 9.5 SalsaAst.hs

```haskell
module SalsaAst where

type Program = [DefCom]
data DefCom = Def Definition
            | Com Command
            deriving (Show, Eq)
data Definition = Viewdef Ident Expr Expr
                | Rectangle Ident Expr Expr Expr Expr Colour
                | Circle Ident Expr Expr Expr Colour
                | View Ident
                | Group Ident [Ident]
                deriving (Show, Eq)
data Command = Move [Ident] Pos
             | At Command Ident
             | Par Command Command
             deriving (Show, Eq)
data Pos = Abs Expr Expr
         | Rel Expr Expr
         deriving (Show, Eq)
data Expr = Plus Expr Expr
          | Minus Expr Expr
          | Const Integer
          | Xproj Ident
          | Yproj Ident
          deriving (Show, Eq)
data Colour = Blue | Plum | Red | Green | Orange
            deriving (Show, Eq)
type Ident = String
```

## 9.6 SalsaInterp.hs

```haskell
--------------------------------------------------------------------------------
--- Student name: Arni Asgeirsson
--- Student KU-id: lwf986
--------------------------------------------------------------------------------
module SalsaInterp
       (Position, interpolate, runProg)
where

```

```haskell
import SalsaAst
import Gpx
import qualified Data.Map as M
import qualified Data.Maybe as Mb
import qualified Control.Monad as Mo

------------------------------------------------------------
--------------------- The interface ------------------------
------------------------------------------------------------

type Position = (Integer, Integer)

interpolate :: Integer -> Position -> Position -> [Position]
interpolate 0 _ _ = []
interpolate framerate (x1,y1) pe@(x2,y2) =
  let
    rate = 100 `div` framerate
    xdis = (x2 - x1) * rate
    ydis = (y2 - y1) * rate
  in
   [(x,y)| i <- [1..(framerate-1)], x <- [x1+((xdis*i) `div` 100)],
          y <- [y1+((ydis*i) `div` 100)]]++[pe]

runProg :: Integer -> Program -> Animation
runProg n p =
  let (_,(_,anim)) = runSalsa (createEmptyContext n)
                   (Mo.join (Salsa $ \con -> (mapM defCom p, con)))
  in
   anim

------------------------------------------------------------
------------------------ Context ---------------------------
------------------------------------------------------------

--------------------- Data Structure -----------------------

data Context = Context ConEnvironment State
             deriving (Show)
type ConEnvironment = (Environment, [Ident], Integer)
type Environment = M.Map Ident Definition
type State = M.Map Ident [(Ident,Position)]

-------------------- Working on the DS ---------------------

createEmptyContext :: Integer -> Context
createEmptyContext n = Context (M.empty, [], n) M.empty

lookupViews :: Ident -> Environment -> [Ident]
lookupViews key env =
  case lookupKey key env of
    (Viewdef view _ _) ->
      [view]
    (Group _ views) ->
      views
    _ ->
      error $ "Tried to look up "++key++" found something not expected"

bindCommand :: Ident -> [(Ident,Position)] -> State -> State
bindCommand = M.insert

addToState :: (State -> State) -> Context -> State
addToState f (Context _ state) = f state
```

```haskell

bindDefinition :: Ident -> Definition -> Environment -> Environment
bindDefinition = M.insert

addToEnvironment :: (Environment -> Environment) -> Context -> Context
addToEnvironment f (Context (env,active,fr) state) =
   Context (f env,active,fr) state

updateActiveViews :: [Ident] -> Context -> Context
updateActiveViews views (Context (env,_,fr) state) =
   Context (env,views,fr) state

placeShapeInActiveViews :: Definition -> Context -> Context
placeShapeInActiveViews (Rectangle id_ (Const x) (Const y) _ _ _) con =
  placeShapeHelper id_ (x,y) con
placeShapeInActiveViews (Circle id_ (Const x) (Const y) _ _) con =
  placeShapeHelper id_ (x,y) con
placeShapeInActiveViews _ _ = error "Trying to place something that is not a shape in the active vie

placeShapeHelper :: Ident -> Position -> Context -> Context
placeShapeHelper id_ pos con =
  let (Context (env,active,fr) state) = con
      positions = map (\view_id -> (view_id,pos)) active
      newState = M.insert id_ positions state
  in
   Context (env,active,fr) newState

------------------ Working on Animation ----------------------

goToNextFrame :: Animation -> Animation
goToNextFrame (views, frames) = (views,frames++[[]])

addInstructions :: [GpxInstr] -> Animation -> Animation
addInstructions new_instr (views, frames) =
  let (rest,[curframe]) = getLast frames
  in
   (views, rest++[curframe++new_instr])

placeShapeInCurrentFrame :: Definition -> [ViewName] -> Animation -> Animation
placeShapeInCurrentFrame (Rectangle _ (Const x) (Const y) (Const w) (Const h) colour) active anim =
  placeShapeFrameHelper (DrawRect x y w h) colour active anim
placeShapeInCurrentFrame (Circle _ (Const x) (Const y) (Const r) colour) active anim =
  placeShapeFrameHelper (DrawCirc x y r) colour active anim
placeShapeInCurrentFrame _ _ _ = error "Trying to place something that is not a shape in the current

placeShapeFrameHelper :: (a -> ColourName -> GpxInstr)
                         -> Colour -> [a] -> Animation -> Animation
placeShapeFrameHelper shape colour active anim =
  let col = evalColour colour
      new_instr = map f active
      f view = shape view col
  in
   addInstructions new_instr anim

addViewToAnimation :: (ViewName,Integer,Integer) -> Animation -> Animation
addViewToAnimation v (views, frames) = (views++[v],frames)

--------------------------------------------------------------
------------------------- Monads -----------------------------
--------------------------------------------------------------

--------------------------------------------------------------
```

```haskell
------------------------- Monads Types ---------------------------


------------------------- SalsaCommand ---------------------------

newtype SalsaCommand a = SalsaCommand {runSC :: Context -> (a,State)}

instance Monad SalsaCommand where
  return k = SalsaCommand $ \(Context _ state) -> (k,state)
  m >>= f = SalsaCommand $ \c ->
    let (Context e _) = c
        (a,state1) = runSC m c
        m1 = f a
    in
     runSC m1 (Context e state1)

---------------------------- Salsa -------------------------------

data Salsa a = Salsa ((Context,Animation) -> (a,(Context,Animation)))

instance Monad Salsa where
  return k = Salsa $ \state -> (k,state)
  (Salsa a1) >>= f = Salsa $ \state0 -> let (r,state1) = a1 state0
                                            (Salsa a2) = f r
                                        in
                                         a2 state1

------------------------------------------------------------------
------------------------- Accessors ------------------------------

------------------------- SalsaCommand ---------------------------

askCmd :: SalsaCommand Context
askCmd = SalsaCommand $ \con@(Context _ s) -> (con,s)

updateState :: (Context -> State) -> SalsaCommand ()
updateState f = SalsaCommand $ \con -> ((), f con)

---------------------------- Salsa -------------------------------

askCont :: Salsa Context
askCont = Salsa $ \s@(con,_) -> (con,s)

runSalsa :: Context -> Salsa a -> (a,(Context,Animation))
runSalsa con (Salsa m) = m (con,([],[[]]))

updateContext :: (Context -> Context) -> Salsa ()
updateContext f = Salsa $ \(con,anim) -> ((), (f con,anim))

updateAnimation :: (Animation -> Animation) -> Salsa ()
updateAnimation f = Salsa $ \(con,anim) -> ((), (con, f anim))

------------------------------------------------------------------
------------------ Interpret Functions ---------------------------
------------------------------------------------------------------

------------------------- SalsaCommand ---------------------------

command :: Command -> SalsaCommand ()
command (Move ids point) = do
  con <- askCmd
  let (Context (_,active,_) state) = con
  (h:_) <- mapM (\x ->
                  do
                    let list = lookupKey x state
                    newlist <- mapM (setNextPosition active point) list
                    updateState (addToState (bindCommand x newlist))
                ) ids
  return h -- Dummy return
command (At cmd id_) = do
  (Context (env,active,_) state) <- askCmd
  let vs = lookupViews id_ env
  (tmp_state,mapping) <- setTmpActiveViews state active vs
  updateState $ const tmp_state
  command cmd
  (Context (_,_,_) state1) <- askCmd
  next_state <- revertActiveViews state1 mapping
  updateState $ const next_state
command (Par cmd1 cmd2) = do
  command cmd1
  command cmd2

------------------------ Salsa -----------------------------

defCom :: DefCom -> Salsa ()
defCom (Def def) = definition def
defCom (Com cmd) = do
  con <- askCont
  let (_,state) = runSC (command cmd) con
      (Context e s) = con
      newcon = Context e state
  instr <- compareStates s state
  updateContext $ const newcon
  updateAnimation $ addInstructions instr
  updateAnimation goToNextFrame

definition :: Definition -> Salsa ()
definition (Viewdef id_ x y) = do
  valx <- evalExpr x askCont
  valy <- evalExpr y askCont
  if valx <  0 || valy < 0 then
    error "A view cannot be defined with a negative width or height!"
   else do
    updateContext (addToEnvironment (bindDefinition id_ $ Viewdef id_ (Const valx) (Const valy)))
    updateContext (updateActiveViews [id_])
    updateAnimation (addViewToAnimation (id_,valx,valy))
definition (View id_) = do
  (Context (env,_,_) _) <- askCont
  let views = lookupViews id_ env
  updateContext (updateActiveViews views)
definition (Group id_ views) = do
  updateContext (addToEnvironment (bindDefinition id_ $ Group id_ views))
  updateContext (updateActiveViews views)
definition (Rectangle id_ x y w h colour) = do
  valx <- evalExpr x askCont
  valy <- evalExpr y askCont
  valw <- evalExpr w askCont
  valh <- evalExpr h askCont
  if valw <  0 || valh < 0 then
    error "A rectangle cannot be defined with a negative width or height!"
   else
    let newRect = Rectangle id_ (Const valx) (Const valy) (Const valw) (Const valh) colour
    in do
      updateContext (addToEnvironment (bindDefinition id_ newRect))
      updateContext (placeShapeInActiveViews newRect)
      (Context (_,active,_) _) <- askCont
      updateAnimation (placeShapeInCurrentFrame newRect active)
definition (Circle id_ x y r colour) = do
  valx <- evalExpr x askCont
  valy <- evalExpr y askCont
  valr <- evalExpr r askCont
  if valr <  0 then
    error "A circle cannot be defined with a negative radius!"
   else
     let newCirc = Circle id_ (Const valx) (Const valy) (Const valr) colour
     in do
       updateContext (addToEnvironment (bindDefinition id_ newCirc))
       updateContext (placeShapeInActiveViews newCirc)
       (Context (_,active,_) _) <- askCont
       updateAnimation (placeShapeInCurrentFrame newCirc active)

-------------------- Interpret Helpers ---------------------
------------------------------------------------------------

--------------------- SalsaCommand -------------------------

evalNextPoint :: Pos -> Position -> SalsaCommand Position
evalNextPoint (Abs exp1 exp2) _ = do
  x <- evalExpr exp1 askCmd
  y <- evalExpr exp2 askCmd
  return (x,y)
evalNextPoint (Rel exp1 exp2) (x1,y1) = do
  x <- evalExpr exp1 askCmd
  y <- evalExpr exp2 askCmd
  return (x1+x,y1+y)

setTmpActiveViews :: State -> [Ident] -> [Ident] -> SalsaCommand (State,[((Ident,Ident),Ident)])
setTmpActiveViews state active tmp_active =
  let list = M.toList state
      (a:_) = active
      act = removeDouble active tmp_active
      (new_list, mappings) = foldl (f tmp_active a) ([],[]) list
      (new_list2,mappings2) = foldl (f act $ '_':a) ([],[]) new_list
      f from to (acc_def,acc_m) (id1,positions) = let
        (next_def,next_m) = foldl (g from to id1) ([],[]) positions
        in
         (acc_def++[(id1,next_def)],acc_m++next_m)
      g from to id2 (acc_xs,acc_ms) (view,pos) = if view `elem` from
                                                 then (acc_xs++[(to,pos)],acc_ms++[((id2, to), view)
                                                 else (acc_xs++[(view,pos)],acc_ms)
    in
   return (M.fromList new_list2, mappings++mappings2)



revertActiveViews :: State -> [((Ident,Ident),Ident)] -> SalsaCommand State
revertActiveViews state mappings = do
  let list = M.toList state
      new_state = map (\(id_,views) ->
                        (id_,map (\(viewName,pos) ->
                                   case lookup (id_,viewName) mappings of
                                     Just previous ->
                                       (previous,pos)
                                     Nothing ->
                                       (viewName,pos)
                                 ) views)) list
  return $ M.fromList new_state

setNextPosition :: Eq t => [t] -> Pos ->
                   (t, Position) -> SalsaCommand (t, Position)
setNextPosition active point (view,pos) =
  if view `elem` active
  then do
    next <- evalNextPoint point pos
    return (view,next)
  else
    return (view,pos)

getLowestPosition :: Position -> (ViewName,Position) -> Position
getLowestPosition (l_x,l_y) (_,(x,y)) = let
  n_x = if x < l_x then x else l_x
  n_y = if y < l_y then y else l_y
  in
   (n_x,n_y)

------------------------- Salsa ------------------------------

-- This function assumes that the set of keys in old is the same as in new
compareStates :: State -> State -> Salsa [GpxInstr]
compareStates old_s new_s = let
  s1 = M.toList old_s
  s2 = M.toList new_s
  in
   do
     l <- mapM (fg s1) s2
     return $ concat l
   where
     fg old (ident,new_positions) =
       case lookup ident old of
         Just old_positions ->
           do
             (Context (env,_,framerate) _) <- askCont
             l <- generateInstructions (lookupKey ident env) framerate
                  old_positions new_positions
             return $ concat l
         Nothing ->
           error $ ident++" did not exist in the new state"

-- It is assumed that every viewname in old_p.. must also appear in new_p..
generateInstructions :: Definition -> Integer -> [(ViewName, Position)] -> [(ViewName, Position)] ->
generateInstructions s fr oldPos =
  mapM (genInstrHelper s fr oldPos)
  where
    genInstrHelper shape framerate old_positions (viewName, new_pos) =
      case lookup viewName old_positions of
        Just old_pos ->
          if old_pos == new_pos
          then return []
          else mapM (positionToInstr shape viewName) (interpolate framerate old_pos new_pos)
        Nothing ->
          error $ viewName++" did not exist in the list of old_positions"

positionToInstr :: Definition -> ViewName -> Position -> Salsa GpxInstr
positionToInstr (Rectangle _ _ _ expw exph expcol) viewName (x,y) = do
  w <- evalExpr expw askCont
  h <- evalExpr exph askCont
  return $ DrawRect x y w h viewName (evalColour expcol)
positionToInstr (Circle _ _ _ expr expcol) viewName (x,y) = do
```

```
  r <- evalExpr expr askCont
  return $ DrawCirc x y r viewName (evalColour expcol)
positionToInstr _ _ _ = error "Trying to create instructions from something that is not a shape"

evalExpr :: Monad m => Expr -> m Context -> m Integer
evalExpr (Const int) _ = return int
evalExpr (Plus exp1 exp2) askf = do
  x <- evalExpr exp1 askf
  y <- evalExpr exp2 askf
  return $ x+y
evalExpr (Minus exp1 exp2) askf = do
  x <- evalExpr exp1 askf
  y <- evalExpr exp2 askf
  return $ x-y
evalExpr (Xproj ident) askf = do
  (Context _ state) <- askf
  let max_i = toInteger(maxBound :: Int)
      (x,_) = foldl getLowestPosition (max_i,max_i) $ lookupKey ident state
  return x
evalExpr (Yproj ident) askf = do
  (Context _ state) <- askf
  let max_i = toInteger(maxBound :: Int)
      (_,y) = foldl getLowestPosition (max_i,max_i) $ lookupKey ident state
  return y

------------------------------------------------------------------
----------------- Helper Functions --------------------------------
------------------------------------------------------------------

lookupKey :: Ord a => a -> M.Map a b -> b
lookupKey key env =
  Mb.fromMaybe (error "Tried to look up unknown key ")
  (M.lookup key env)


removeDouble :: Eq a => [a] -> [a] -> [a]
removeDouble [] _ = []
removeDouble _ [] = []
removeDouble (x:xs) ys =
  if x `elem` ys
  then removeDouble xs ys
  else x:removeDouble xs ys

evalColour :: Colour -> ColourName
evalColour Blue = "blue"
evalColour Plum = "plum"
evalColour Red = "red"
evalColour Green = "green"
evalColour Orange = "orange"

getLast :: [a] -> ([a],[a])
getLast [] = ([],[])
getLast (x:[]) = ([],[x])
getLast (x:xs) = let (rest,last_) = getLast xs
                 in
                  (x:rest,last_)
```

## 9.7 SalsaParser.hs

```
----------------------------------------------------------------------------
```

```haskell
-- Student name: Arni Asgeirsson
-- Student KU-id: lwf986
-----------------------------------------------------------------------------
module SalsaParser (parseString, parseFile, Error(..)) where

import SalsaAst
import Text.ParserCombinators.ReadP

----------------------------------------------------------------
------------------- The interface ------------------------------
----------------------------------------------------------------

data Error = NoParsePossible String
           | AmbiguousGrammar [(Program, String)]
           | UnexpectedRemainder Program String
           deriving (Eq, Show)

parseString :: String -> Either Error Program
parseString = parse runParser

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

----------------------------------------------------------------
-------------------- The parser functions ----------------------
----------------------------------------------------------------

------------------------ Definitions ---------------------------

identarr :: String
identarr = ['A'..'Z']++['a'..'z']++['0'..'9']++"_"

reservedWords :: [String]
reservedWords = ["viewdef","rectangle", "circle", "group", "view"]

colourNames :: [String]
colourNames = ["blue", "plum", "red", "green", "orange"]

---------------------- Top-Level parsers -----------------------

parse :: ReadP Program -> String -> Either Error Program
parse parser s =
  case readP_to_S parser s of
    [(result, "")] -> Right result
    [(result, unparsed)] -> Left $ UnexpectedRemainder result unparsed
    [] -> Left $ NoParsePossible s
    results -> Left $ AmbiguousGrammar results

runParser :: ReadP Program
runParser = do
  skipSpaces
  p <- pProgram
  skipSpaces
  eof
  return p

---------------------- Grammar Parsers -------------------------

pProgram :: ReadP Program
pProgram = pDefComs
```

```

pDefComs :: ReadP [DefCom]
pDefComs = do
  dc <- pDefCom
  dcs <- pDefComs'
  return $ dc:dcs

pDefComs' :: ReadP [DefCom]
pDefComs' =  pDefComs +++ return []

pDefCom :: ReadP DefCom
pDefCom = do
  cmd <- pCommand
  return $ Com cmd
  +++
  do
    def <- pDefinition
    return $ Def def

pDefinition :: ReadP Definition
pDefinition = hViewdef +++ hRectangle +++ hCircle +++ hView +++ hGroup

hViewdef :: ReadP Definition
hViewdef = do
  stringT "viewdef"
  vid <- pVIdent
  exp1 <- pExpr
  exp2 <- pExpr
  return $ Viewdef vid exp1 exp2

hRectangle :: ReadP Definition
hRectangle = do
  stringT "rectangle"
  sid <- pSIdent
  exp1 <- pExpr
  exp2 <- pExpr
  exp3 <- pExpr
  exp4 <- pExpr
  col <- pColour
  return $ Rectangle sid exp1 exp2 exp3 exp4 col

hCircle :: ReadP Definition
hCircle = do
  stringT "circle"
  sid <- pSIdent
  exp1 <- pExpr
  exp2 <- pExpr
  exp3 <- pExpr
  col <- pColour
  return $ Circle sid exp1 exp2 exp3 col

hView :: ReadP Definition
hView = do
  stringT "view"
  vid <- pVIdent
  return $ View vid

hGroup :: ReadP Definition
hGroup = do
  stringT "group"
  vid <- pVIdent
  vids <- bracks '[' pVIdents ']'
```

```haskell
  return $ Group vid vids

pCommand :: ReadP Command
pCommand = do
  cmd2 <- pCommand2
  pCommand' cmd2

pCommand' :: Command -> ReadP Command
pCommand' iV = do
  stringT "||"
  cmd2 <- pCommand2
  pCommand' $ Par iV cmd2
  +++
  return iV

pCommand2 :: ReadP Command
pCommand2 = do
  cmd3 <- pCommand3
  pCommand2' cmd3

pCommand2' :: Command -> ReadP Command
pCommand2' iV = do
  charT '@'
  vid <- pVIdent
  pCommand2' $ At iV vid
  +++
  return iV

pCommand3 :: ReadP Command
pCommand3 = do
  sids <- pSIdents
  stringT "->"
  pos <- pPos
  return $ Move sids pos
  +++
  bracks '{' pCommand '}'

pVIdents :: ReadP [Ident]
pVIdents = do
  vid <- pVIdent
  vids' <- pVIdents'
  return $ vid:vids'

pVIdents' :: ReadP [Ident]
pVIdents' = pVIdents +++ return []

pSIdents :: ReadP [Ident]
pSIdents = do
  sid <- pSIdent
  sids' <- pSIdents'
  return $ sid:sids'

pSIdents' :: ReadP [Ident]
pSIdents' = pSIdents +++ return []

pPos :: ReadP Pos
pPos =
  bracks '(' (hMiddle Abs) ')'
  +++
  do
  charT '+'
  bracks '(' (hMiddle Rel) ')'
```

```haskell

hMiddle :: (Expr -> Expr -> b) -> ReadP b
hMiddle c = do
  exp1 <- pExpr
  charT ','
  exp2 <- pExpr
  return $ c exp1 exp2

pExpr :: ReadP Expr
pExpr = do
  prim <- pPrim
  pExpr' prim

pExpr' :: Expr -> ReadP Expr
pExpr' iV = do
  exp_ <- pOp iV
  pExpr' exp_
  +++
  return iV

pOp :: Expr -> ReadP Expr
pOp iV = do
  charT '+'
  prim <- pPrim
  return $ Plus iV prim
  +++
  do
  charT '-'
  prim <- pPrim
  return $ Minus iV prim

pPrim :: ReadP Expr
pPrim = do
  int <- pInteger
  return $ Const int
  +++
  bracks '(' pExpr ')'
  +++
  do
  sid <- pSIdent
  charT '.'
  pProj sid

pProj :: Ident -> ReadP Expr
pProj iV = do
  charT 'x'
  return $ Xproj iV
  +++
  do
  charT 'y'
  return $ Yproj iV

pColour :: ReadP Colour
pColour = do
  stringT "blue"
  return Blue
  +++
  do
  stringT "plum"
  return Plum
  +++
  do
  stringT "red"
  return Red
  +++
  do
  stringT "green"
  return Green
  +++
  do
  stringT "orange"
  return Orange

-------------------------- Extra parsers --------------------------

pVIdent :: ReadP Ident
pVIdent = do
  skipSpaces
  h <- satisfy (`elem` ['A'..'Z'])
  rest <- munch (`elem` identarr)
  skipSpaces
  return $ h:rest

pSIdent :: ReadP Ident
pSIdent =  do
  skipSpaces
  h <- satisfy (`elem` ['a'..'z'])
  rest <- munch (`elem` identarr)
  skipSpaces
  let ident = h:rest
  if ident `elem` (reservedWords++colourNames)
    then pfail
    else return ident

pInteger :: ReadP Integer
pInteger = do
  skipSpaces
  n <- munch1 (`elem` ['0'..'9'])
  skipSpaces
  -- Note: read is a partial function
  return (read n::Integer)

------------------------- Helper parsers --------------------------

bracks ::  Char -> ReadP b -> Char -> ReadP b
bracks lb a rb = do
  charT lb
  b <- a
  charT rb
  return b

stringT :: String -> ReadP ()
stringT s = do
  skipSpaces
  _ <- string s
  skipSpaces

charT :: Char -> ReadP ()
charT c = do
  skipSpaces
  _ <- char c
  skipSpaces
```

## 9.8 Test_Interp.hs

```haskell
---------------------------------------------------------------------------
--- Student name: Arni Asgeirsson
--- Student KU-id: lwf986
---------------------------------------------------------------------------
module Test_Interp
       (runAllTestsI,runAllTestsIWith)
       where

import SalsaAst
import Gpx
import Test.QuickCheck
import qualified Test.QuickCheck as QC
import Control.Monad
import SalsaInterp
import qualified Data.Map as M
import Data.Char
import Data.List
import qualified Data.Maybe as Mb
import Test.HUnit

----------------------------------------------------------------
------------------- Interface to run tests -------------------
----------------------------------------------------------------

runAllTestsI :: IO ()
runAllTestsI = runAllTestsIWith 100

runAllTestsIWith :: Int -> IO ()
runAllTestsIWith n = do
  putStrLn "------------------------------------------------------------------"
  putStrLn "------------ Running tests for the SalsaInterp module ------------"
  putStrLn "------------------------------------------------------------------\n"
  putStrLn "-------------------- Running QuickCheck tests --------------------"
  putStrLn $ "1. Testing if the interpreter inteprets the expected outputs from\n"
    ++"      random valid Salsa Program."
  putStrLn "Might take a few seconds ...\n"
  runQCTest n
  putStrLn "\n---------------------- Running HUnit tests -------------------------"
  putStrLn "2. Testing if the Par command works as intended\n"
  _ <- runTestTT parCases
  putStrLn "\n3. Testing if the At command and Group definition works together\n"
  _ <- runTestTT atgroupCases
  putStrLn "\n4. Testing if the Xproj and Yproj expressions work\n"
  _ <- runTestTT projCases
  putStrLn "\n5. Testing if the interpolate function works as intended\n"
  _ <- runTestTT interpolateCases
  return () -- Dummy return

----------------------------------------------------------------
---------------------- QuickCheck Tests ----------------------
------------------ Test valid input Programs ------------------
----------------------------------------------------------------

----------------------- QC test runner -----------------------

-- TODO allow it to be user defined how many tests it must run
runQCTest :: Int -> IO ()
runQCTest n = QC.quickCheckWith QC.stdArgs{maxSuccess = n } prop_runProg

```

```
---------------------------- Property ----------------------------

prop_runProg :: TestAnimation -> Bool
prop_runProg (TestAnimation ((i,n),o)) = compareAnimations (runProg n i) o

---------------------------- Test type ----------------------------

newtype TestAnimation = TestAnimation ((Program,Integer), Animation)
                        deriving (Show, Eq)

instance QC.Arbitrary TestAnimation where
  arbitrary = do
    defcoms <- QC.listOf1 $ QC.elements $ definitions++commands
    (input,output_) <- genManyDefcom ("viewdef":defcoms)
    return $ TestAnimation (input, output_)

---------------------------- Definitions ----------------------------

identarr :: String
identarr = ['A'..'Z']++['a'..'z']++['0'..'9']++"_"

definitions :: [String]
definitions = ["viewdef","rectangle", "circle", "view","group"]

commands :: [String]
commands = ["move",  "at",--"par",
            "move","move","move"]

colours :: [(String,Colour)]
colours = [("blue",Blue), ("plum",Plum), ("red",Red),
           ("green",Green), ("orange",Orange)]

numbers :: String
numbers = ['0'..'9']

exprList :: [String]
exprList = ["const","plus", "minus", --"xproj", "yproj",
            "const","const","const","const","const"]

posList :: [String]
posList = ["abs","rel"]

---------------------------- Generators ----------------------------


-- The framerate is intentionaly keept low to avoid very very big data sets

genManyDefcom :: [String] -> QC.Gen (([DefCom],Integer),Animation)
genManyDefcom [] = error "Cannot parse an empty list of definitions or commands"
genManyDefcom words_ = do
  n <- QC.elements ['1'..'9']
  let framerate = read [n]::Integer
      init_ = (createEmptyContext framerate,[],([],[[]]))
  (_,all_defcoms,all_anim) <- foldM f init_ words_
  return ((all_defcoms,framerate),all_anim)
  where
    f (acontext,acci,anim) word = do
      (context,defcoms,new_anim) <- genDefcom word acontext anim
      if word `elem` commands && defcoms /= []
        then
          let (view,frames) = new_anim
              next = frames++[[]]
          in
           return (context,acci++defcoms,(view,next))
        else
          return (context,acci++defcoms,new_anim)
genDefcom :: String -> Context -> Animation -> QC.Gen (Context,[DefCom],Animation)
genDefcom "viewdef" c@(Context (env,_,n) state) a@(views,frames) = do
  vident <- genVident
  if isInEnvironment vident env
    then
      return (c,[],a)
    else do
      expw_ <- genExpr env
      exph_ <- genExpr env
      expw <- forcePositive expw_ state
      exph <- forcePositive exph_ state
      w <- evalExprQC expw state
      h <- evalExprQC exph state
      let def = Viewdef vident expw exph
          env' = M.insert vident def env
      return (Context (env',[vident],n) state,[Def def], (views++[(vident,w,h)],frames))
genDefcom "rectangle" c@(Context (env,active,n) state) a@(views,frames) = do
  sident <- genSident
  if isInEnvironment sident env
    then
      return (c,[],a)
    else do
      expx <- genExpr env
      expy <- genExpr env
      expw_ <- genExpr env
      expw <- forcePositive expw_ state
      exph_ <- genExpr env
      exph <- forcePositive exph_ state
      x <- evalExprQC expx state
      y <- evalExprQC expy state
      w <- evalExprQC expw state
      h <- evalExprQC exph state
      (col,col_type) <- genColour
      let (rest,[last_]) = getLast frames
          instrs = map (\viewName -> DrawRect x y w h viewName col) active
          def = Rectangle sident expx expy expw exph col_type
          env' = M.insert sident def env
          list = map (\viewName -> (viewName,(x,y))) active
          state' = M.insert sident list state
      return (Context (env',active,n) state',[Def def], (views,rest++[last_++instrs]))
genDefcom "circle" c@(Context (env,active,n) state) a@(views,frames) = do
  sident <- genSident
  if isInEnvironment sident env
    then
      return (c,[],a)
    else do
      expx <- genExpr env
      expy <- genExpr env
      expr_ <- genExpr env
      expr <- forcePositive expr_ state
      x <- evalExprQC expx state
      y <- evalExprQC expy state
      r <- evalExprQC expr state
      (col,col_type) <- genColour
      let (rest,[last_]) = getLast frames
          instrs = map (\viewName -> DrawCirc x y r viewName col) active
          def = Circle sident expx expy expr col_type
          env' = M.insert sident def env
          list = map (\viewName -> (viewName,(x,y))) active
          state' = M.insert sident list state
      return (Context (env',active,n) state',[Def def], (views,rest++[last_++instrs]))
genDefcom "view" c@(Context (env,_,n) state) a@anim =
  let list = M.toList env
      flist = filter (\(_,def) -> case def of
                         (Group _ _) -> True
                         (View _) -> True
                         _ -> False
                         ) list
  in
   if null flist
   then return (c,[], a)
   else do
     (id_,some_def) <- QC.elements flist
     let new_active = case some_def of
           (Group _ g) -> g
           (View v) -> [v]
           _ -> error "Shouldn't be possible due to filtering above"
     return (Context (env,new_active,n) state, [Def $ View id_],anim)
genDefcom "group" c@(Context (_,[],_) _) a =
  return (c,[],a)
genDefcom "group" c@(Context (env,active,n) state) a = do
  vident <- genVident
  if isInEnvironment vident env
    then
      return (c,[],a)
    else do
      new_actives_ <- QC.listOf1 $ QC.elements active
      let new_actives = removeDuplex new_actives_
          def = Group vident new_actives
          env' = M.insert vident def env
      return (Context (env',new_actives,n) state, [Def def], a)
genDefcom "move" c@(Context (env,active,n) state) a@(views,frames) =
  let list = M.toList env
      flist = filter (\(x:_,_) -> isLower x) list
  in
   if null flist
   then return (c,[], a)
   else do
     ids <- QC.listOf1 $ QC.elements flist
     let ids2 = removeDuplex ids
     expPos <- genPos env
     (all_instr,new_state) <- foldM (f_ active expPos n state) ([],state) ids2
     let (rest,[last_]) = getLast frames
         next1 = rest++[last_++all_instr]
         ids_ = map fst ids2
         def = Move ids_ expPos
     return (Context (env,active,n) new_state,[Com def],(views,next1))
genDefcom "at" c@(Context (env,active,n) state) a =
  let list = M.toList env
      flist = filter (\(_,def) -> case def of
                   --      (Group _ _) -> True -- Uncomment to see doom and destruction!
                         (View _) -> True
                         _ -> False
                         ) list
  in
   if null flist
   then return (c,[], a)
   else do
     (id_,some_def) <- QC.elements flist
     let tmp_active = case some_def of
```

```
            (Group _ g) -> g
            (View v) -> [v]
            _ -> error "Shouldn't be possible due to the filtering above"
      middle_cmd <- QC.elements commands
      middle <- genDefcom middle_cmd (Context (env,tmp_active,n) state) a
      let (Context (env',_,_) state', a_com, anim') = middle
      case a_com of
        [] ->
          return (c,[],a)
        [Com some] ->
          return (Context (env',active,n) state', [Com $ At some id_], anim')
        _ -> error "Shouldn't be possible"
genDefcom "par" c a = do
  cmd1 <- QC.elements commands
  cmd2 <- QC.elements commands
  res1 <- genDefcom cmd1 c a
  let (con1, defcom1, anim1) = res1
  res2 <- genDefcom cmd2 con1 anim1
  let (con2, defcom2, anim2) = res2
  return (case (defcom1,defcom2) of
              ([],_) ->
                (c, [], a)
              (_,[]) ->
                (c, [], a)
              ([Com com1],[Com com2]) ->
                (con2, [Com $ Par com1 com2], anim2)
              _ -> error "Shouldm't be possible")
genDefcom s _ _ = error $ "Cannot parse "++s++" into a DefCom"

forcePositive :: Expr -> State_ -> Gen Expr
forcePositive exp_ s = do
  val <- evalExprQC exp_ s
  return (if val >= 0
          then
            exp_
          else
            Plus exp_ (Const (val*(-2))))

-- COM assumes that id is in state
-- TODO RENAME
f_ :: Ord t => [ViewName]  -> Pos -> Integer -> State_ ->
      ([GpxInstr], M.Map t [(ViewName, (Integer, Integer))]) ->
      (t, Definition) -> Gen ([GpxInstr], M.Map t [(ViewName, (Integer, Integer))])
f_ active pos n s acc (id_,def) =
  foldM (\(acc_instrs, acc_state) a ->
           let positions = lookupKey id_ acc_state
           in
            case lookup a positions of
              Just old_pos -> do
                next_pos <- evalPos_ old_pos pos s
                let next_state = M.insert id_ (map (\(vn,p) -> if vn == a
                                                               then (vn,next_pos)
                                                               else (vn,p)
                                                    ) positions) acc_state
                instr <- genInstrs old_pos next_pos def n a s
                return (acc_instrs++instr,next_state)
              Nothing ->
                return (acc_instrs,acc_state)) acc active

genInstrs :: (Integer, Integer)
                   -> (Integer, Integer)
                   -> Definition
                   -> Integer
                   -> ViewName
                   -> State_
                   -> Gen [GpxInstr]
genInstrs opos npos (Rectangle _ _ _ ew eh ecol) n view s = do
  w <- evalExprQC ew s
  h <- evalExprQC eh s
  col <- evalColourQc ecol
  let positions = interpolate n opos npos
  return $ map (\(x,y) -> DrawRect x y w h view col) positions
genInstrs opos npos (Circle _ _ _ er ecol) n view s = do
  r <- evalExprQC er s
  col <- evalColourQc ecol
  let positions = interpolate n opos npos
  return $ map (\(x,y) -> DrawCirc x y r view col) positions
genInstrs _ _ _ _ _ _ _ = error "Cannot generate instructions if not given a shape"

evalPos_ :: (Integer, Integer)
                  -> Pos -> State_ -> Gen (Integer, Integer)
evalPos_ _ (Abs expx expy) s = do
  x2 <- evalExprQC expx s
  y2 <- evalExprQC expy s
  return (x2,y2)
evalPos_ (x,y) (Rel expx expy) s = do
  x2 <- evalExprQC expx s
  y2 <- evalExprQC expy s
  return (x+x2,y+y2)

genPos :: Environment -> QC.Gen Pos
genPos env = do
  pos <- QC.elements posList
  _genPos pos env

_genPos :: String -> Environment -> QC.Gen Pos
_genPos "abs" env = do
  expx <- genExpr env
  expy <- genExpr env
  return (Abs expx expy)
_genPos "rel" env = do
  expx <- genExpr env
  expy <- genExpr env
  return (Rel expx expy)
_genPos s _ = error $ "Cannot parse "++s++" into an Pos"

evalColourQc :: Colour -> QC.Gen String
evalColourQc Blue = return "blue"
evalColourQc Plum = return "plum"
evalColourQc Red = return "red"
evalColourQc Green = return "green"
evalColourQc Orange = return "orange"

evalExprQC :: Expr -> State_ -> QC.Gen Integer
evalExprQC (Const n) _ = return n
evalExprQC (Plus e1 e2) s = do
  n1 <- evalExprQC e1 s
  n2 <- evalExprQC e2 s
  return $ n1 + n2
evalExprQC (Minus e1 e2) s = do
  n1 <- evalExprQC e1 s
  n2 <- evalExprQC e2 s
  return $ n1 - n2
evalExprQC (Xproj id_) s = do
```

```haskell
  let max_i = toInteger(maxBound :: Int)
      (x,_) = foldl getLowestPosition (max_i,max_i) $ lookupKey id_ s
  return x
evalExprQC (Yproj id_) s = do
  let max_i = toInteger(maxBound :: Int)
      (_,y) = foldl getLowestPosition (max_i,max_i) $ lookupKey id_ s
  return y

genExpr :: Environment -> QC.Gen Expr
genExpr env = do
  expr <- QC.elements exprList
  _genExpr expr env

-- If I have to create a xproj or yproj, but no shape definition
-- has been made yet, a Const will be returned instead.
_genExpr :: String -> Environment -> QC.Gen Expr
_genExpr "plus" env = do
  (exp1) <- genExpr env
  (exp2) <- genExpr env
  return (Plus exp1 exp2)
_genExpr "minus" env = do
  (exp1) <- genExpr env
  (exp2) <- genExpr env
  return (Minus exp1 exp2)
_genExpr "const" _ = do
  n <- genNumber
  return (Const (read n::Integer))
_genExpr "xproj" env = projHelper Xproj env
_genExpr "yproj" env = projHelper Yproj env
_genExpr s _ = error $ "Cannot parse "++s++" into an Expr"

projHelper :: (String -> Expr) -> Environment -> QC.Gen Expr
projHelper a env =
  let list  = M.toList env
      flist = filter (\(x:_,_) -> isLower x) list
    in
   if null flist
   then
     _genExpr "const" env
   else do
     (sid,_) <- QC.elements flist
     return $ a sid

genColour :: QC.Gen (String,Colour)
genColour = QC.elements colours

genVident :: QC.Gen String
genVident = do
  h <- QC.elements ['A'..'Z']
  rest <- QC.listOf $ QC.elements identarr
  return $ h:rest

genSident :: QC.Gen String
genSident = do
  h <- QC.elements ['a'..'z']
  rest <- QC.listOf $ QC.elements identarr
  return $ h:rest

genNumber :: QC.Gen String
genNumber = QC.listOf1 $ QC.elements numbers

------------------------------------------------------------
```

```haskell
---------------------- Helper Functions ----------------------
-----------------------------------------------------------------

compareAnimations :: Animation -> Animation -> Bool
compareAnimations (views1,frames1) (views2,frames2) =
  let (b1,_) = g views1 views2
      b2 = f frames1 frames2
      f [] [] = True
      f [] _ = False
      f _ [] = False
      f (x:xs) (y:ys) = let (b,_) =g x y
                        in
                         b && f xs ys
      g x1 x2 = foldl (\(a,v2) v -> if a &&  v `elem` v2
                                   then (True,delete v v2)
                                   else (False,[])
                     ) (True,x2) x1
  in
   b1 && b2

isInEnvironment :: Ord k => k -> M.Map k a -> Bool
isInEnvironment id_ env = case M.lookup id_ env of
  Just _ -> True
  Nothing -> False

removeDuplex :: Eq a => [a] -> [a]
removeDuplex [] = []
removeDuplex (x:xs) = if x `elem` xs
                      then removeDuplex xs
                      else x:removeDuplex xs

------------ Copied from the SalsaInterp.hs file ------------

data Context = Context ConEnvironment State_
             deriving (Show)

type ConEnvironment = (Environment, [Ident], Integer)
type Environment = M.Map Ident Definition
type State_ = M.Map Ident [(Ident,Position)]


createEmptyContext :: Integer -> Context
createEmptyContext n = Context (M.empty, [], n) M.empty

lookupKey :: Ord a => a -> M.Map a b -> b
lookupKey key env =
  Mb.fromMaybe (error "Tried to look up unknown key ")
  (M.lookup key env)

getLast :: [a] -> ([a],[a])
getLast [] = ([],[])
getLast (x:[]) = ([],[x])
getLast (x:xs) = let (rest,last_) = getLast xs
                 in
                  (x:rest,last_)

getLowestPosition :: Position -> (ViewName,Position) -> Position
getLowestPosition (l_x,l_y) (_,(x,y)) = let
  n_x = if x < l_x then x else l_x
  n_y = if y < l_y then y else l_y
  in
   (n_x,n_y)
```

```


------------------------------------------------------------
----------------------- HUnit tests ------------------------
------------------------ Par tests -------------------------
------------------------------------------------------------

parCases :: Test
parCases = TestLabel "Test cases for the Par command"
           $ TestList [testP1]

-- Test that par command works in a simple example
testP1 :: Test
testP1 = let s = [ Def (Viewdef "A" (Const 200) (Const 12))
                 , Def (Circle "b" (Const 10) (Const 10) (Const 5) Green)
                 , Def (Circle "c" (Const 10) (Const 10) (Const 5) Blue)
                 , Com (Par (Move ["b"] (Abs (Const 20) (Const 50))) (Move ["c"] (Abs (Const 100) (C
             a = ([("A",200,12)],[[DrawCirc 10 10 5 "A" "green",DrawCirc 10 10 5 "A" "blue"
                                  ,DrawCirc 12 18 5 "A" "green",DrawCirc 14 26 5 "A" "green"
                                  ,DrawCirc 16 34 5 "A" "green",DrawCirc 18 42 5 "A" "green"
                                  ,DrawCirc 20 50 5 "A" "green",DrawCirc 28 18 5 "A" "blue"
                                  ,DrawCirc 46 26 5 "A" "blue",DrawCirc 64 34 5 "A" "blue"
                                  ,DrawCirc 82 42 5 "A" "blue",DrawCirc 100 50 5 "A" "blue"],[]])
         in TestCase $ assertEqual "" a (runProg 5 s)



------------------------------------------------------------
---------------------- At/Group tests ----------------------
------------------------------------------------------------

atgroupCases :: Test
atgroupCases = TestLabel "Test cases for testing At and Group"
               $ TestList [testAg1]

-- Test that the At command and Group definition works together

testAg1 :: Test
testAg1 = let s = [ Def (Viewdef "A" (Const 200) (Const 12))
                  , Def (Circle "b" (Const 200) (Const 20) (Const 5) Green)
                  , Def (Viewdef "B" (Const 10) (Const 500))
                  , Def (Group "C" ["A","B"])
                  , Com (At (Move ["b"] (Rel (Const 20) (Const 50))) "C")]
              a = ([("A",200,12),("B",10,500)],
                   [[DrawCirc 200 20 5 "A" "green"
                    ,DrawCirc 210 45 5 "A" "green",DrawCirc 220 70 5 "A" "green"],[]])
          in TestCase $ assertEqual "" a (runProg 2 s)

------------------------------------------------------------
----------------------- Proj tests -------------------------
------------------------------------------------------------

projCases :: Test
projCases = TestLabel "Test cases for the [XY]proj expressions"
            $ TestList [testPr1]

-- Test that Xproj and Yproj works in a simple example
testPr1 :: Test
testPr1 = let s = [ Def (Viewdef "A" (Const 200) (Const 12))
                  , Def (Circle "b" (Const 200) (Const 20) (Const 5) Green)
                  , Def (Circle "c" (Const 10) (Const 100) (Const 5) Blue)
                  , Com (Move ["b"] (Abs (Yproj "c") (Xproj "b")))]
              a = ([("A",200,12)],[[DrawCirc 200 20 5 "A" "green",DrawCirc 10 100 5 "A" "blue"
                                  ,DrawCirc 150 110 5 "A" "green",DrawCirc 100 200 5 "A" "green"],
          in TestCase $ assertEqual "" a (runProg 2 s)

------------------------------------------------------------
-------------------- interpolate tests ---------------------
------------------------------------------------------------

interpolateCases :: Test
interpolateCases = TestLabel "Test cases for the interpolate function"
                   $ TestList [testIp1,testIp2,testIp3,testIp4,testIp5]

-- Test with 0 frame rate
testIp1 :: Test
testIp1 = let s = interpolate 0 (0,0) (10,10)
              a = []
              d = "0 frame rate should return []"
          in TestCase $ assertEqual d a s

-- Test with 1 frame rate
testIp2 :: Test
testIp2 = let s = interpolate 1 (0,0) (10,10)
              a = [(10,10)]
              d = "1 frame rate should return the end point"
          in TestCase $ assertEqual d a s

-- Test with 5 frame rate
testIp3 :: Test
testIp3 = let s = interpolate 5 (0,0) (10,10)
              a = [(2,2),(4,4),(6,6),(8,8),(10,10)]
              d = "5 frame rate should return a list of 5 points"
          in TestCase $ assertEqual d a s

-- Test with 5 and negative direction
testIp4 :: Test
testIp4 = let s = interpolate 5 (10,10) (0,0)
              a = [(8,8),(6,6),(4,4),(2,2),(0,0)]
              d = "5 frame rate should return a list of 5 points"
          in TestCase $ assertEqual d a s


-- Test with same points
testIp5 :: Test
testIp5 = let s = interpolate 3 (5,5) (5,5)
              a = [(5,5),(5,5),(5,5)]
              d = "interpolate from point a to a should only a"
          in TestCase $ assertEqual d a s
```

## 9.9 Test_Parser.hs

```
----------------------------------------------------------------------
--- Student name: Arni Asgeirsson
--- Student KU-id: lwf986
----------------------------------------------------------------------
module Test_Parser
       (runAllTests,runAllTestsWith)
       where

import Test.HUnit
import SalsaAst
import SalsaParser
import Test.QuickCheck
import qualified Test.QuickCheck as QC
import Control.Monad
import Control.Exception

----------------------------------------------------------------
------------------- Interface to run tests ---------------------
----------------------------------------------------------------

runAllTests :: IO Bool
runAllTests = runAllTestsWith 100

runAllTestsWith :: Int -> IO Bool
runAllTestsWith n = do
  putStrLn "--------------------------------------------------------------------"
  putStrLn "------------- Running tests for the SalsaParser module -------------"
  putStrLn "--------------------------------------------------------------------\n"
  putStrLn "---------------------- Running QuickCheck tests --------------------"
  putStrLn $ "1. Testing if the parser parses the expected outputs from random"
    ++" valid input strings"
  putStrLn "Might take a few seconds ...\n"
  runQCTest n
  putStrLn "\n--------------------- Running HUnit tests ------------------------"
  putStrLn "2. Testing if the precedence and associativity works as intended\n"
  _ <- runTestTT precedenceCases
  putStrLn $ "\n3. Testing if the expected errors occur with specific invalid"
    ++" input strings\n"
  _ <- runTestTT errorCases
  putStrLn "\n--------------------- Running Unit tests -------------------------"
  putStrLn "4. Testing if an empty file parses as expected\n"
  b1 <- testF1
  print b1
  putStrLn "\n5. Testing if an simple Salsa file parses as expected\n"
  b2 <- testF2
  print b2
  putStrLn "\n6. Testing if an more advanced Salsa file parses as expected\n"
  b3 <- testF3
  print b3
  putStrLn "\n7. Testing if an invalid Salsa program from file parses as expected\n"
  b4 <- testF4
  print b4
  putStrLn "\n8. Testing if a non-existing file raises the expected error\n"
  testF5

----------------------------------------------------------------
---------------------- QuickCheck Tests ------------------------
------------------- Test valid input strings -------------------
----------------------------------------------------------------

------------------------ Definitions ---------------------------

identarr :: String
identarr = ['A'..'Z']++['a'..'z']++['0'..'9']++"_"

definitions :: [String]
definitions = ["viewdef","rectangle", "circle", "view", "group"]

commands :: [String]
commands = ["move","at","par",
            "move","move","move"]

```

```haskell
colours :: [(String,Colour)]
colours = [("blue",Blue), ("plum",Plum), ("red",Red),
           ("green",Green), ("orange",Orange)]

whiteSpaces :: [String]
whiteSpaces = [" ","\n"]

numbers :: String
numbers = ['0'..'9']

exprList :: [String]
exprList = ["plus", "minus", "const", "xproj", "yproj",
            "const","const","const","const"]

posList :: [String]
posList = ["abs","rel"]

------------------------- Test type -------------------------

newtype TestProgram = TestProgram (String, Either Error Program)
                      deriving (Show, Eq)

instance QC.Arbitrary TestProgram where
  arbitrary = do
    defcoms <- QC.listOf1 $ QC.elements $ definitions++commands
    (input,output_) <- genManyDefcom defcoms
    return $ TestProgram (input, Right output_)

-------------------------- Property --------------------------

prop_pProgram :: TestProgram -> Bool
prop_pProgram (TestProgram (i,o)) = parseString i == o

------------------------ QC test runner ------------------------

runQCTest :: Int -> IO ()
runQCTest n = QC.quickCheckWith QC.stdArgs{maxSuccess = n } prop_pProgram

------------------------- Generators -------------------------

genManyDefcom :: [String] -> QC.Gen (String,[DefCom])
genManyDefcom [] = error "Cannot parse an empty list of definitions or commands"
genManyDefcom words_ = do
  result <- mapM genDefcom words_
  foldM f ("",[]) result
  where
    f (acci,acco) (i,o) = return (acci++i,acco++o)

genDefcom :: String -> QC.Gen (String, [DefCom])
genDefcom "viewdef" = do
  vident <- genVident
  (w,expw) <- genExpr
  (h,exph) <- genExpr
  input <- insertWhiteSpaces1 ["viewdef",vident,w,h]
  return (input, [Def $ Viewdef vident expw exph])
genDefcom "rectangle" = do
  sident <- genSident
  (x,expx) <- genExpr
  (y,expy) <- genExpr
  (w,expw) <- genExpr
  (h,exph) <- genExpr
  (col,col_type) <- genColour
  input <- insertWhiteSpaces1 ["rectangle",sident,x,y,w,h,col]
  return (input, [Def $ Rectangle sident expx expy expw exph col_type])
genDefcom "circle" = do
  sident <- genSident
  (x,expx) <- genExpr
  (y,expy) <- genExpr
  (r,expr) <- genExpr
  (col,col_type) <- genColour
  input <- insertWhiteSpaces1 ["circle",sident,x,y,r,col]
  return (input, [Def $ Circle sident expx expy expr col_type])
genDefcom "view" = do
  vident <- genVident
  input <- insertWhiteSpaces1 ["view",vident]
  return (input, [Def $ View vident])
genDefcom "group" = do
  vident <- genVident
  vidents <- QC.listOf1 genVident
  input <- insertWhiteSpaces1 $ ["group",vident,"["]++vidents++["]"]
  return (input, [Def $ Group vident vidents])
genDefcom "move" = do
  sidents <- QC.listOf1 genSident
  (pos,expPos) <- genPos
  input <- insertWhiteSpaces1 $ ["{"]++sidents++["->",pos,"}"]
  return (input, [Com $ Move sidents expPos])
genDefcom "at" = do
  word <- QC.elements commands
  (cmd,Com cmdexp:[]) <- genDefcom word
  vident <- genVident
  input <- insertWhiteSpaces1 ["{",cmd,"@",vident,"}"]
  return (input, [Com $ At cmdexp vident])
genDefcom "par" = do
  word1 <- QC.elements commands
  word2 <- QC.elements commands
  (cmd1,Com cmdexp1:[]) <- genDefcom word1
  (cmd2,Com cmdexp2:[]) <- genDefcom word2
  input <- insertWhiteSpaces ["{",cmd1,"||",cmd2,"}"]
  return (input, [Com $ Par cmdexp1 cmdexp2])
genDefcom s = error $ "Cannot parse "++s++" into a DefCom"

genPos :: QC.Gen (String, Pos)
genPos = do
  pos <- QC.elements posList
  _genPos pos

_genPos :: String -> QC.Gen (String, Pos)
_genPos "abs" = do
  (x,expx) <- genExpr
  (y,expy) <- genExpr
  input <- insertWhiteSpaces ["(",x,",",y,")"]
  return (input, Abs expx expy)
_genPos "rel" = do
  (x,expx) <- genExpr
  (y,expy) <- genExpr
  input <- insertWhiteSpaces ["+","(",x,",",y,")"]
  return (input, Rel expx expy)
_genPos s = error $ "Cannot parse "++s++" into an Pos"

genExpr :: QC.Gen (String, Expr)
genExpr = do
  expr <- QC.elements exprList
  _genExpr expr
```

```haskell
_genExpr :: String -> QC.Gen (String, Expr)
_genExpr "plus" = do
  (e1,exp1) <- genExpr
  (e2,exp2) <- genExpr
  input <- insertWhiteSpaces ["(",e1,"+",e2,")"]
  return (input, Plus exp1 exp2)
_genExpr "minus" = do
  (e1,exp1) <- genExpr
  (e2,exp2) <- genExpr
  input <- insertWhiteSpaces ["(",e1,"-",e2,")"]
  return (input, Minus exp1 exp2)
_genExpr "const" = do
  n <- genNumber
  input <- insertWhiteSpaces ["(",n,")"]
  return (input, Const (read n::Integer))
_genExpr "xproj" = do
  sident <- genSident
  input <- insertWhiteSpaces ["(",sident,".","x",")"]
  return (input, Xproj sident)
_genExpr "yproj" = do
  sident <- genSident
  input <- insertWhiteSpaces ["(",sident,".","y",")"]
  return (input, Yproj sident)
_genExpr s = error $ "Cannot parse "++s++" into an Expr"

genColour :: QC.Gen (String,Colour)
genColour = QC.elements colours

genVident :: QC.Gen String
genVident = do
  h <- QC.elements ['A'..'Z']
  rest <- QC.listOf $ QC.elements identarr
  return $ h:rest

genSident :: QC.Gen String
genSident = do
  h <- QC.elements ['a'..'z']
  rest <- QC.listOf $ QC.elements identarr
  return $ h:rest

genNumber :: QC.Gen String
genNumber = QC.listOf1 $ QC.elements numbers

insertWhiteSpaces :: [String] -> QC.Gen String
insertWhiteSpaces = wsHelper QC.listOf

insertWhiteSpaces1 :: [String] -> QC.Gen String
insertWhiteSpaces1 = wsHelper QC.listOf1

wsHelper :: (QC.Gen String -> QC.Gen [String]) -> [String] -> QC.Gen String
wsHelper m words_ = do
  initWhite <- m $ QC.elements whiteSpaces
  foldM f (concat initWhite) words_
  where
    f acc word = do
      whitespaces <- QC.listOf1 $ QC.elements whiteSpaces
      return $ acc++word++concat whitespaces

------------------------------------------------------------------------------
--------------------------- HUnit tests --------------------------------------
------------------------- Precedence tests -----------------------------------
------------------------------------------------------------------------------
```

```

precedenceCases :: Test
precedenceCases = TestLabel "Test cases for precedence"
                  $ TestList [testP1,testP2,testP3,testP4,testP5,
                              testP6,testP7,testP8,testP9,testP10,
                              testP11]

-- Show that @ and || are both left associative
testP1 :: Test
testP1 = let s = "a -> (0,0) @ A @ B @ C"
             a = "{{{ a -> (0,0) @ A } @ B } @ C }"
             d = "@ must be left associative"
         in TestCase $ assertEqual d (parseString a) (parseString s)
testP2 :: Test
testP2 = let s = "a -> (0,0) || b->(0,0) || c->(0,0)"
             a = "{{{a -> (0,0)} || b->(0,0)} || c->(0,0)}"
             d = "|| must be left associative"
         in TestCase $ assertEqual d (parseString a) (parseString s)

-- Left associativity of + and -
testP3 :: Test
testP3 = let s = "viewdef A 1 1+5+2"
             a = "viewdef A 1 (1+5)+2"
             d = "+ must be left associative"
         in TestCase $ assertEqual d (parseString a) (parseString s)
testP4 :: Test
testP4 = let s = "viewdef A 1 1-5-2"
             a = "viewdef A 1 (1-5)-2"
             d = "- must be left associative"
         in TestCase $ assertEqual d (parseString a) (parseString s)
testP5 :: Test
testP5 = let s = "viewdef A 1 1+5-2-5"
             a = "viewdef A 1 ((1+5)-2)-5"
             d = "+ & - must be left associative"
         in TestCase $ assertEqual d (parseString a) (parseString s)

-- Show that @ has higher precedence than ||
testP6 :: Test
testP6 = let s = "a -> (0,0) || b->(0,0) @ A"
             a = "a -> (0,0) || {b->(0,0) @ A}"
             d = "@ should have higher precedence that ||"
         in TestCase $ assertEqual d (parseString a) (parseString s)
testP7 :: Test
testP7 = let s = "a -> (0,0) || b->(0,0) @ A || c->(0,0) @ B @ C"
             a = "a -> (0,0) || {b->(0,0) @ A} || {{c->(0,0) @ B} @ C}"
             d = "@ should have higher precedence that ||"
         in TestCase $ assertEqual d (parseString a) (parseString s)

-- Show that + and - has the same precedence (ie it maintains its order)
testP8 :: Test
testP8 = let s = "viewdef A 1 1+5-4"
             a = "viewdef A 1 ((1+5)-4)"
             d = "+ & - should have the same precedence"
         in TestCase $ assertEqual d (parseString a) (parseString s)
testP9 :: Test
testP9 = let s = "viewdef A 1 1-5+4"
             a = "viewdef A 1 ((1-5)+4)"
             d = "+ & - should have the same precedence"
         in TestCase $ assertEqual d (parseString a) (parseString s)

-- Show that . has higher precedence than + and -
testP10 :: Test
```

```haskell
testP10 = let s = "viewdef A 1 r.x + c.y"
              a = "viewdef A 1 (r.x) + (c.y)"
              d = ". should have higher precedence than +"
          in TestCase $ assertEqual d (parseString a) (parseString s)
testP11 :: Test
testP11 = let s = "viewdef A 1 r.x - c.y"
              a = "viewdef A 1 (r.x) - (c.y)"
              d = ". should have higher precedence than -"
          in TestCase $ assertEqual d (parseString a) (parseString s)

------------------------------------------------------------
----------------------- Error tests -----------------------
------------------------------------------------------------
errorCases :: Test
errorCases = TestLabel "Test cases for invalid input strings"
             $ TestList [testE1, testE2, testE3, testE4, testE5,
                         testE6, testE7, testE8, testE9, testE10,
                         testE11,testE12,testE13,testE14, testE15,
                         testE16,testE17,testE18,testE19,testE20,
                         testE21,testE22,testE23,testE24,testE25,
                         testE26,testE27,testE28,testE29,testE30,
                         testE31,testE32,testE33,testE34,testE35,
                         testE36,testE37,testE38,testE39,testE40,
                         testE41]

-- Parse empty string
testE1 :: Test
testE1 = let s = ""
         in TestCase $ assertEqual "" (Left $ NoParsePossible s) (parseString s)

-- Using sident in place of vident
testE2 :: Test
testE2 = let s = "viewdef sident 4 5"
         in TestCase $ assertEqual "" (Left $ NoParsePossible s) (parseString s)
testE3 :: Test
testE3 = let s = "view sident"
         in TestCase $ assertEqual "" (Left $ NoParsePossible s) (parseString s)
testE4 :: Test
testE4 = let s = "group sident [A,B,C]"
         in TestCase $ assertEqual "" (Left $ NoParsePossible s) (parseString s)
testE5 :: Test
testE5 = let s = "group Vident [A,B,sident]"
         in TestCase $ assertEqual "" (Left $ NoParsePossible s) (parseString s)
testE6 :: Test
testE6 = let s = "{ a -> (1,1) } @ sident"
         in TestCase $ assertEqual "" (Left $ NoParsePossible s) (parseString s)

-- Using vident in place of sident
testE7 :: Test
testE7 = let s = "rectangle Vident 1 2 3 4 green"
         in TestCase $ assertEqual "" (Left $ NoParsePossible s) (parseString s)
testE8 :: Test
testE8 = let s = "circle Vident 1 2 3 red"
         in TestCase $ assertEqual "" (Left $ NoParsePossible s) (parseString s)
testE9 :: Test
testE9 = let s = "{ Vident -> (1,1) } @ A"
         in TestCase $ assertEqual "" (Left $ NoParsePossible s) (parseString s)
testE10 :: Test
testE10 = let s = "{ a b c Vident -> (1,1) } @ A"
          in TestCase $ assertEqual "" (Left $ NoParsePossible s) (parseString s)
testE11 :: Test
testE11 = let s = "viewdef A 1 (1 + Vident . x)"
```

```
          in TestCase $ assertEqual "" (Left $ NoParsePossible s) (parseString s)

-- Using wrong types of characters
testE12 :: Test
testE12 = let s = "viewdef _A 1 2"
          in TestCase $ assertEqual "" (Left $ NoParsePossible s) (parseString s)
testE13 :: Test
testE13 = let s = "viewdef A+a 1 2"
          in TestCase $ assertEqual "" (Left $ NoParsePossible s) (parseString s)
testE14 :: Test
testE14 = let s = "view Bæ"
          in TestCase $ assertEqual "" (Left $ NoParsePossible s) (parseString s)
testE15 :: Test
testE15 = let s = "circle a 1 2 (a . z) red"
          in TestCase $ assertEqual "" (Left $ NoParsePossible s) (parseString s)

-- Using numbers instead of letters, (same as using wrong characters?)
testE16 :: Test
testE16 = let s = "5 -> (0,0)"
          in TestCase $ assertEqual "" (Left $ NoParsePossible s) (parseString s)
testE17 :: Test
testE17 = let s = "group 2 [ A ]"
          in TestCase $ assertEqual "" (Left $ NoParsePossible s) (parseString s)

-- Using letters instead of numbers
testE18 :: Test
testE18 = let s = "viewdef A b 4"
          in TestCase $ assertEqual "" (Left $ NoParsePossible s) (parseString s)
testE19 :: Test
testE19 = let s = "viewdef A 4 (a+2)"
          in TestCase $ assertEqual "" (Left $ NoParsePossible s) (parseString s)

-- TODO does it make sense to test for this?
-- Using wrong parenthesis
testE20 :: Test
testE20 = let s = "( a -> (0,0) )"
          in TestCase $ assertEqual "" (Left $ NoParsePossible s) (parseString s)
testE21 :: Test
testE21 = let s = "group A { B }"
          in TestCase $ assertEqual "" (Left $ NoParsePossible s) (parseString s)
testE22 :: Test
testE22 = let s = "{ a -> [0,0] }"
          in TestCase $ assertEqual "" (Left $ NoParsePossible s) (parseString s)

-- TODO does it make sense to test for this?
-- Using non-existing operator
testE23 :: Test
testE23 = let s = "a -> - (1,2)"
          in TestCase $ assertEqual "" (Left $ NoParsePossible s) (parseString s)
testE24 :: Test
testE24 = let s = "viewdef A 1 (2 * 4)"
          in TestCase $ assertEqual "" (Left $ NoParsePossible s) (parseString s)
testE25 :: Test
testE25 = let s = "circle a 3 1 ( a , x) red"
          in TestCase $ assertEqual "" (Left $ NoParsePossible s) (parseString s)

-- Using a reserved word as sident (res+colour)
testE26 :: Test
testE26 = let s = "rectangle viewdef 1 2 3 4 green"
           in TestCase $ assertEqual "" (Left $ NoParsePossible s) (parseString s)
testE27 :: Test
testE27 = let s = "rectangle rectangle 1 2 3 4 green"
```

```
          in TestCase $ assertEqual "" (Left $ NoParsePossible s) (parseString s)
testE28 :: Test
testE28 = let s = "circle a 1 2 (1 + group . x)"
          in TestCase $ assertEqual "" (Left $ NoParsePossible s) (parseString s)
testE29 :: Test
testE29 = let s = "green -> (6,66)"
          in TestCase $ assertEqual "" (Left $ NoParsePossible s) (parseString s)

-- TODO does it make sense to test for this?
-- Using invalid colour name
testE30 :: Test
testE30 = let s = "circle a 1 2 3 purple"
          in TestCase $ assertEqual "" (Left $ NoParsePossible s) (parseString s)

-- Wrong whitespace inbetween stuff
testE31 :: Test
testE31 = let s = "circle a 31 ( a , x) red"
          in TestCase $ assertEqual "" (Left $ NoParsePossible s) (parseString s)
testE32 :: Test
testE32 = let s = "circlea 3 1 ( a , x) red"
          in TestCase $ assertEqual "" (Left $ NoParsePossible s) (parseString s)
testE33 :: Test
testE33 = let s = "viewdefA 1 2"
          in TestCase $ assertEqual ""
             (Right [Def $ Viewdef "A" (Const 1) (Const 2)])
             (parseString s)
testE34 :: Test
testE34 = let s = "group A [BCD]"
          in TestCase $ assertEqual ""
             (Right [Def $ Group "A" ["BCD"]])
             (parseString s)

-- Grouping zero vidents
testE35 :: Test
testE35 = let s = "group A [ ]"
          in TestCase $ assertEqual "" (Left $ NoParsePossible s) (parseString s)

-- Show that the salsa is case sensitive
testE36 :: Test
testE36 = let s = "Viewdef A 1 2"
          in TestCase $ assertEqual "" (Left $ NoParsePossible s) (parseString s)
testE37 :: Test
testE37 = let s = "Group A [ B ]"
          in TestCase $ assertEqual "" (Left $ NoParsePossible s) (parseString s)
testE38 :: Test
testE38 = let s = "rectangle a 1 2 3 4 Orange"
          in TestCase $ assertEqual "" (Left $ NoParsePossible s) (parseString s)

-- Use negative values
testE39 :: Test
testE39 = let s = "viewdef A -2 5"
          in TestCase $ assertEqual "" (Left $ NoParsePossible s) (parseString s)
testE40 :: Test
testE40 = let s = "viewdef A (-2) 5"
          in TestCase $ assertEqual "" (Left $ NoParsePossible s) (parseString s)

-- Show that simple expression can create negative values to by pass the
-- integer restriction
testE41 :: Test
testE41 = let s = "viewdef A (0-2) 5"
          in TestCase $ assertEqual ""
             (Right [Def $ Viewdef "A" (Minus (Const 0) (Const 2)) (Const 5)])
```

```
              (parseString s)

-- Show the UnexpectedRemainder error
-- Umm how?

------------------------------------------------------------------
-------------------------- Unit tests --------------------------
----------------------- Parse file tests -----------------------
------------------------------------------------------------------

-- Parse empty file
testF1 :: IO Bool
testF1 = testFile "test_files/empty.salsa"

-- Parse simple salsa
testF2 :: IO Bool
testF2 = testFile "test_files/simple.salsa"

-- Parse multi salsa
testF3 :: IO Bool
testF3 = testFile "test_files/multi.salsa"

-- Parse invalid salsa
testF4 :: IO Bool
testF4 = testFile "test_files/invalid.salsa"

-- Parse non-existing file
-- Note that if the file does exist then it returns the parse from there
-- So this does only test the intended if the file indeed does not exist.
testF5 :: IO Bool
testF5 = catch (testFile "test_files/doesNotExist.salsa")
         (\e -> do let _ = e::IOException
                   return True)

-- A helper function to compare the results from parseString and parseFile
testFile :: String -> IO Bool
testFile path_ = do
  content <- readFile path_
  output_ <- parseFile path_
  return $ parseString content == output_
```

## 9.10 multi.salsa

```
viewdef One 500 500
viewdef Two 400 400
group Both [One Two]
view Both
rectangle larry 10 350 20 20 blue
rectangle fawn 300 350 15 25 plum

view Two
larry -> (300, 350) || fawn -> (10,350)

view Both
larry fawn -> +(0, 0 - 300)
```

## 9.11 simple.salsa

```
viewdef Default 400 400
rectangle box 10 400 20 20 green
box -> (10, 200)
box -> +(100, 0)
box -> (110,400)
box -> +(0-100, 0)
```

## 9.12 empty.salsa

```
```

## 9.13 invalid.salsa

```
viewdef Default 400 a
rectangle Box 10 assda 20 20 green 10 23
```