Advanced Programming
Exam 2015

Nikolaj Dybdahl Rathcke (rfq695)

November 5, 2015

1 SubScript, Parser

The parser is built using the Parsec library. A program is parsed successfully when the input is well formed, based on the grammar, and we can build internal SubsAst representation. The source code for the implementation can be found E.1.

1.1 The Grammar

The grammar has been changed slightly. The `Stms` has been redefined to look like

```
Stms      ::= Stm ';' Stms
            | Stm ';'
```

This is equivalent to the handed out grammar except that the grammar can not be used for empty programs.
For `Stm`, though the grammar can not be expressed in the same way as it uses auxiliary functions, it should behave exactly the same.
For `Expr` we wanted to avoid left recursion problems, as in the following (from the handed out grammar):

```
Expr      ::= Expr ',' Expr
            | Expr1
```

This is dealt with by converting the first `Expr` to an `Expr1`. When the grammar is left recursive, we can't determine where to go by looking at the first symbol (in top-down parsing).
We also wanted `Expr1` to account for the fact that the arithmetic operators `*` and `%` binds stronger than `+` and `-` which in turn binds stronger than `<` and then `===`. This was handled in the following way:

```
expr1 :: Parser Expr
expr1 = do
      rv <- r
      botbot rv

botbot :: Expr -> ParsecT String () Data.Functor.Identity.Identity Expr
botbot inval = do
      void $ lexeme $ string "==="
      rv <- r
      botbot (Call "===" [inval, rv])
  <|> return inval

r :: ParsecT String () Data.Functor.Identity.Identity Expr
r = do
      sv <- s
      bot sv

bot :: Expr -> ParsecT String () Data.Functor.Identity.Identity Expr
bot inval = (do
      void $ lexeme $ char '<'
      sv <- s
      bot (Call "<" [inval, sv]))
  <|> return inval
```

This is only for one level of precedence (`===` binds weaker than `<`), but it applies the same principle to the other levels of precedence.
The rest of the grammar is all related to each other, so it is hard to distinctly explain a part of the grammar. However, the entire grammar is implemented. Note that it is not implemented in a straight

forward manner (that something is parsed in the function corresponding to the grammar), as there are many auxiliary functions that are called and some parts might be parsed a different place compared to the grammar: However, a parsed program should correspond to abstract syntax tree that is handed out without changes to it.

1.2 Parser Solution and Testing

The following commands are supported for parsing a program

```
SubsParser> parseFile "Path to file with a program"
```

or

```
SalsaParser> parseString "Program as string"
```

To test the parser, I have used an `HUnit` test suite, which executes a number of tests based on assertions. To run these tests, load the module `ParserTest.hs` (in `../src/subs` or appendix D.1) and use the following command

```
ParserTest> runTestTT tests
```

The main test is the program `intro.js`, which produces a `SubsAst` which is (almost) identical to the one in the handouts. The only difference is that sometimes there is a space after a comma in the handouts, but not in this parser. The `SubsAst`'s can be found in appendix A.1. This test covers most of the `for` and `if` statements as well as array and parentheses in the grammar.
Other tests (denoted `TestSuccX`, where X is a number) include small programs that succeed in precedence, associativity, `Ident` names and `FunName` . These parse correctly (this can also mean that it fails to parse when it should). These tests are about other things in the grammar that is not in `intro.js`

1.3 Assessment

First, an assessment of the code style. No warnings are produced, but there are 7 suggestions when using `hlint`. All but on of these are "reduce duplication", which I have chosen to ignore. The last suggestion seem to break the code.
The parser itself correctly parses most of the input. It successfully parses the `intro.js` program which uses a great part of the grammar. Precedence also works as intended.
Testing could have been more thorough. Right now, I am aware of one mistake - when there is a space before an "." (this does parse, but it should not). But an immediate fix (removing that it parses spaces before it) breaks another part of the code , so we will call this a feature for now. I am sure there are a lot more edge cases that parses incorrectly, but the 'essential' aspects of the parser is working - I could have used QuickCheck to find the edge cases that parses incorrectly, but time constraints has kept me from this as I was not familiar with it.

2 `SubScript`, Interpreter

The source code can be found in appendix E.2.

2.1 The `SubsM` Monad

The implemented SubsM monad is very similar to the state monad. To construct a SubsM instance of type a, we go from a *context* to either an error (which is a string) or some *(a, env)* where the environment, *env*, is mapping from variable names to values. A context consists of an environment and another mapping, *PEnv*, from function names and operands to the haskell functions implementing them.

Utility functions: Three auxillary functions has been implemented, namely `getEnv`, `getPEnv` and `putEnv`. We use these to update the environment, *env*, by getting the environment, inserting a key and value (overwriting if it exists), and putting the environment back in the context. We use them in the same manner to fetch values from the environment and getting function from the primitives environment, *PEnv*.

This function `evalExpr` implements the evaluation of expressions. The evaluation can raise errors when

operands are not of the same type (besides the special case with a string and integer in addition, which results in a string) or when a non-existent variable or function is referred to.
This function `stm` works in the same manner and the function `program` evaluates a statement and recursively calls itself until there are no more statements (end of program).

The last function, `runProg`, invokes the function `runSubsM` wrapped inside of the type constructor for SubsM. We construct an initial context that consists of an empty environment *env* and the primitives environment *PEnv* where the haskell functions have been implemented. Since there is no API for updating the primitives environment, this is the same throughout the entire program. Given the initial context and a statement, we get an error or a new environment. If it succeeds, the context is updated with the new environment. This is repeated until there are no more statements, in which the program terminates.

2.2 Missing Implementation

Array comprehension has not been implemented (which makes the utility function `modify` superfluous, so it is commented out). This obviously means that the implemented version of `SubScript` lacks a lot of functionality.

2.3 Interpreter Testing

The testing that has been done is "blackbox" testing. In appendix B.1 is the program `simpleProg.js` and the output from running the program. The program runs statements that should be legal and that they are evaluated correctly.
All statement should be legal, which is verified by the interpreter not generating any error. The most interesting cases are that

- Variable 'a' can be referred to after having been declared (line 2)
- Strings are compared by their lexicographic order (var 'e' and 'f' in line 6 – 7)
- Variable 'i', which is a string, can be concatenated with integers and the other way around for 'j' (line 11 – 12). Note that "2 * 3" is evaluated first.
- var 'l' is initialized with value 'undefined'

The other cases are just statements that are evaluated correctly.

In appendix B.2, the errors that the interpreter can generate is listed. These are simply "bad types" and "Key does not exist" errors. However they state in what kind of expression, the error occurred.

All other errors are generated by the parser which is tested in Section 1. Note that all tests associated with the interpreter has the prefix "intrp_".

2.4 Assessment

The testing is very simple, but rightfully so, as array comprehension has not been implemented which would be a big part of it.
However, leaving that aside, the supported functionality is working satisfactory. It generates errors when it should and evaluates legal statements correctly. I have not been able to produce a statement that is evaluated wrong.
Running programs with the implemented parser might be faulty, but I feel the interpreter is working as intended (in what functionality is implemented) and if there are any "mistakes", my first suspicious would be that the parser might have parsed something wrong (as all tests have been made using the command: `runhaskell Subs.hs _TestProgram`.

3 Generic Replicated Server Library

The module `gen_replicated` has been implemented and the source code can be found in appendix E.3.

3.1 Implementation

The implementation for *coordinator* loops with the argument *Pids*, a list of process ID's on all the replicas.
The implementation of *replica* loops with the arguments *State*, the state, and *Mod*, the callback module.

Implementation of the client interface functions:
`start(NumReplica, Mod)`: The function in the implementation works by calling the `init()` function in the given callback module. If it is successfull, then {`ok, State`} is returned from `init()`. It then creates a new process, the coordinator with an enmpty `Pids` list, then sends a message to the newly created coordinator to spawn replicas corresponding to `NumReplica` with the `Mod` as its callback module and returned {`ok, ServerRef`}. If `NumReplica` is less than 2, an error occurs as the implementation uses one replica as a dedicated writer (the first one). If some error should occur in `init()`, then {`error, Reason`} is returned.

`stop(Server)`: The function sends a message to the coordinator to shutdown all replica and itself. If succesfull, then {`ok, stopped`} is returned. It has not been implemented that all clients get the error message {`'ABORTED', server_stopped`} if they are waiting for a read or write.

`read(Server, Req)`: The function sends a message to the coordinator directly (non-blocking) with some read request `Req`. The implementation supports an answer which is a successfull read request, a stop request or if the callback function has made a throw exception.

`write(Server, Req)`: The function sends a message to the auxiliary function `Blocking` with some write request `Req`. The implementation supports an answer which is a successfull write request (both when an update is made to the state and when not), a stop request or if the callback function has made a throw exception.

Implementation of the coordinator and the replica
`coordinator(Pids)`: The coordinator is the one that keep track of what happens. It initiates tasks for the *replica*. In the implementation it can match on the following messages:

- `make_Replicas`, a call from `start` that creates the replicas with the auxiliary function `createReplica` which return a list of tuples which contain the process ID's and their status (free or busy, but this was used in the implementation).
- `getPids` which returns the list `Pids` - only used for testing purposes.
- `stop`, which makes the coordinator send a message to all replicas with the message stop. When this happens, the server does not call itself and therefore dies.
- `read`, which picks a random reader in the interval [2..NumReplica] of `Pids` and assigns that replica to the read request. The original purpose was the make it find a process which was "free" (status in the tuple) and assign the request to that, but as this was not implemented, it was made randomized instead, so we do not pick the same over and over again. Note that it cannot pick the first element in the list as this is the dedicated writer.
- `write`, which assigns the write request to the first process in `Pids`.
- `update`, should the callback function `handle_write` return with a new state, the replica sends this message to the coordinator, which then sends a message to all replica with the new state.

`replica(State, Mod)`: This function is the "worker", the one that actually performs the requests by calling the callback functions. It can match on the following messages:

- `stop`, if it gets this message, it will return `stopped, self()` and simply not loop and thus, dies.
- `read`, which calls the callback function `handle_read`. It then tries to match one of 2 cases: {`reply, Reply` or `stop`. If it gets the `reply` message, it send the term `Reply` back to the caller (the read function). If it gets the `stop` message, it sends this back and instruct the caller to call the stop function. If the callback function should throw an exception, it returns the value given by throw back.

- `write`, which calls the callback function `handle_write`. It then tries to match one of 3 cases: {`noUpdate, Reply`}, {`updated, Reply, NewState`} or `stop`. If it gets the `noUpdate` message, it simply returns the term `Reply` to the caller (as in read). If it gets the `updated` message, it will instruct the coordinator to update states (with the `update` message) in all replicas and returns the reply to the write function along with the new state. If it gets stop, it instructs the caller to call the stop function. It also matches a throw exception as read does.

3.2 Testing

To test the implementation, a module `callback_mod` which can be seen in appendix C.1 was made. The idea is that it is a simple callback module, which an array is used as the state. We can then test the implementation by invoke the returns as specified in the implementation to see if they behave as intended. The idea is our initial state is an array containing the number from 1 to 9. Our request, *Req*, is defined by number which is treated as an index.
As such, the `handle_read` callback reads the number on the index *Req* in the state and returns the state. Otherwise it can return `stop` (in which case {`ok, stopped`} is returned and all servers are shut down. It can also make the throw exception.
The `handle_write` works in the same manner with `stop` and throw exceptions. The request is also an index, but it replaces the value on that index with 99 (always). It can return {`noUpdate, Reply`} (even though an update is made) where the `Reply` is the new state. It can return {`updated, Reply, NewState`} in which case it returns the old state in `Reply` and the new state in `NewState`.

The tests and the output can be seen in appendix C.2, where each return from the callback module is handled correctly. There are descriptions to each test, though "Test 2" is probably the most interesting one as it shows we correctly spawn 5 replicas with the correct state and that a write request is updated on all other servers.

3.3 Assessment

I know the testing is a bit laid back. But the tests all provide correct answer for all the different returns, which is actually quite exstensive as a lot of the edge cases lie in how the request is handled (by the callback module) which is not in the scope for this question.
It does lack the functionality (and testing) for concurrent read and write requests. It did have the functionality to find a process that was not busy (the function `findAvailable` which is outcommented in the source code), but I did not get to implement the rest.
I think the solution lacks the last couple of features, but the implementation that was done is correct.

4 AlzheimerDB

This implementation does not work, but I will go through what I have done so far. It is using `gen_server`. The source code can be found in E.4

4.1 Implementation

`start`: This is simply a call to `start/3` from `gen_server`. This results in the callback function `init/1` which in the implementation returns {`ok, dict:new()`} where the dictionary is out `State`.

`query`: This is a call to `gen_server`'s `call/2` where the request is {`read, P`} where `P` is a function. This results in the callback function `handle_call/3`. The idea was that it calls the auxiliary function `filterWithError` which returns the list of {`Key, Value`} pairs in the dictionary (made into an array using `dict:to_list`) than returned true when `P` was applied on them. If an exception was thrown, we put that a {`error, Row`} in the front of the list that was returned, so we'd know whether to return the error or {`ok, Rows`}. This was also why I could not just use the filter function as it did not handle thrown exceptions.

`upsert`: This also called `gen_server`'s `call/2` with the request {`write, Id, F`} where `F` is a function. It would then look up the identifier `Id` in the dictionary using `dict:find`. If it existed, we call the function `F` on the argument {`existing`, {`Id, Value`}} - Where `Value` is returned from `dict:find`). If

the did not exists we call `F` on {`new, Id`}.
This function `F` can return {`modify, NewData`} (does not matter if there was a key or not) in which case we insert/replace the value under the `Id` with the value in `NewData`. This works with the function `dict:store`. This returns {`modify, NewData`}.
It could also return `ignore`, in which case we do nothing and return `ignore`.
In case `F` makes an exception, this is also what should be returned.

I could not finish the task due to time restraints.

A Parser

A.1 intro.js

The abstract syntax tree produced by the parser for `intro.js`

```
Prog [
    VarDecl "xs" (
        Just (Array [
            Number 0,Number 1,Number 2,
            Number 3,Number 4,Number 5,
            Number 6,Number 7,Number 8,Number 9])),
    VarDecl "squares" (
        Just (Compr ("x",Var "xs",Nothing)
            (Call "*" [Var "x",Var "x"]))),
    VarDecl "evens" (
        Just (Compr ("x",Var "xs",
            Just (ArrayIf (
                Call "===" [
                    Call "%" [Var "x",Number 2],Number 0]) Nothing))
                (Var "x"))),
    VarDecl "many_a" (
        Just (Compr ("x",Var "xs",
            Just (ArrayForCompr ("y",Var "xs",Nothing)))
                (String "a"))),
    VarDecl "hundred" (
        Just (Compr ("i",Array [Number 0],
            Just (ArrayForCompr ("x",Var "xs",
                Just (ArrayForCompr ("y",Var "xs",Nothing)))))
                    (Assign "i" (Call "+" [Var "i",Number 1]))))
]
```

The handed out abstract syntax tree for `intro.js`

```
Prog [
    VarDecl "xs" (
        Just (Array [
            Number 0, Number 1, Number 2,
            Number 3, Number 4, Number 5,
            Number 6, Number 7, Number 8, Number 9])),
    VarDecl "squares" (
        Just (Compr ("x",Var "xs", Nothing)
            (Call "*" [Var "x",Var "x"]))),
    VarDecl "evens" (
        Just (Compr ("x",Var "xs",
            Just (ArrayIf (
                Call "===" [
                    Call "%" [Var "x",Number 2], Number 0]) Nothing))
                (Var "x"))),
    VarDecl "many_a" (
        Just (Compr ("x",Var "xs",
            Just (ArrayForCompr ("y",Var "xs", Nothing)))
                (String "a"))),
    VarDecl "hundred" (
        Just (Compr ("i",Array [Number 0],
            Just (ArrayForCompr ("x",Var "xs",
                Just (ArrayForCompr ("y",Var "xs", Nothing)))))
                    (Assign "i" (Call "+" [Var "i", Number 1]))))
]
```

B Interpreter

B.1 simpleProg.js

The program `simlpeProg.js`:

```
var a = 3+3*2;
a = a-4%3;
b = a === 8;
c = ’c’ === ’d’;
var d = true === 2 < 4;
e = ’ab’ < ’ba’;
f = ’cab’ < ’ba’;
g = 4 % 2 < 3;
h = 4 < 3;
var i = ’i’;
i = i + 2 * 3;
var j = 2 + ’j’;
k = [3, 2, 2%2];
var l;
var m = ’a’ + ’b’;
```

This is the output of running the program `simpleProg.js` with the command:
”runhaskell Subs.hs tests/simpleProg.js” from the `./src` folder.

```
a = 8
b = true
c = false
d = true
e = true
f = false
g = true
h = false
i = "i6"
j = "2j"
k = [3, 2, 0]
l = undefined
m = "ab"
```

B.2 Errors

Errors than can be generated by the interpreter.

```
$ runhaskell Subs.hs tests/intrp_invalidVar
Subs.hs: Error "Key does not exist in ’getVar’"

$ runhaskell Subs.hs tests/intrp_isEqualBadTypes
Subs.hs: Error "Bad argument types in ’===’"

$ runhaskell Subs.hs tests/intrp_lessThanBadTypes
Subs.hs: Error "Bad argument types in ’<’"

$ runhaskell Subs.hs tests/intrp_minusBadTypes
Subs.hs: Error "Both operators must be in ’-’"

$ runhaskell Subs.hs tests/intrp_moduloBadTypes
Subs.hs: Error "Both operator must be integers in ’%’"

$ runhaskell Subs.hs tests/intrp_multiplyBadTypes
Subs.hs: Error "Both operators must be integers in ’*’"
```

```
$ runhaskell Subs.hs tests/intrp_plusBadTypes
Subs.hs: Error "Bad argument types in '+'"
```

C Generic Replicated Server Library

C.1

callback_mod

```
-module(callback_mod).
-export([init/0, handle_read/2, handle_write/2]).

init() ->
% {error, something_went_wrong}.
 {ok, [1,2,3,4,5,6,7,8,9]}.

handle_read(Req, State) ->
%  throw(something_happened).
%  stop.
   {reply, lists:nth(Req, State)}.

handle_write(Req, State) ->
% throw(something_happened).
%  stop.
% {noUpdate, lists:sublist(State, Req-1) ++ [99] ++ lists:nthtail(Req, State)}.
   {updated, State, lists:sublist(State, Req-1) ++ [99] ++ lists:nthtail(Req, State)}.
```

C.2 Testing

```
%%%%%%% TEST 1 %%%%%%%
% Test if the error in init is handled correctly

1> c(gen_replicated).
{ok,gen_replicated}
2> c(callback_mod).
{ok,callback_mod}
3> {_, P1} = gen_replicated:start(5, callback_mod).
{error,something_went_wrong}

%%%%%%% TEST 2 %%%%%%%
% Test to see if handle_write (which returns updated) actually updates on all servers.
% That means the read requests should return the initial state with the change.
% Furthermore we test that 5 replpicas are actually made.

1> c(gen_replicated).
{ok,gen_replicated}
2> c(callback_mod).
{ok,callback_mod}
3> {_, P1} = gen_replicated:start(5, callback_mod).
{ok,<0.47.0>}
4> gen_replicated:write(P1, 4).
{ok,[1,2,3,4,5,6,7,8,9],[1,2,3,99,5,6,7,8,9]}
5> gen_replicated:read(P1, 4).
{ok,99}
6> gen_replicated:read(P1, 5).
{ok,5}
7> gen_replicated:get_pids(P1).
{ok,[{<0.48.0>,free},
     {<0.49.0>,free},
     {<0.50.0>,free},
     {<0.51.0>,free},
     {<0.52.0>,free}]}

%%%%%%% TEST 3 %%%%%%%
% Test to see if handle_write (which now returns noUpdate) work.
% This is seen by the return from handle_read.

1> c(gen_replicated).
```

```
{ok,gen_replicated}
2> c(callback_mod).
{ok,callback_mod}
3> {_, P1} = gen_replicated:start(5, callback_mod).
{ok,<0.47.0>}
4> gen_replicated:write(P1, 4).
{ok,[1,2,3,99,5,6,7,8,9]}
5> gen_replicated:read(P1, 4).
{ok,4}

%%%%%%% TEST 4 %%%%%%%
% Test to see if the stop returned from handle_read stops the servers,
% but handle_write (which sends updated) still works.

1> c(gen_replicated).
{ok,gen_replicated}
2> c(callback_mod).
callback_mod.erl:8: Warning: variable 'Req' is unused
callback_mod.erl:8: Warning: variable 'State' is unused
{ok,callback_mod}
3> {_, P1} = gen_replicated:start(5, callback_mod).
{ok,<0.47.0>}
4> gen_replicated:write(P1, 4).
{ok,[1,2,3,4,5,6,7,8,9],[1,2,3,99,5,6,7,8,9]}
5> gen_replicated:write(P1, 3).
{ok,[1,2,3,99,5,6,7,8,9],[1,2,99,99,5,6,7,8,9]}
6> gen_replicated:read(P1, 3).
{ok,stopped}

%%%%%%% TEST 5 %%%%%%%
% Test to see if the throw returned by handle_read works,
% while the write requests still works.

1> c(gen_replicated).
{ok,gen_replicated}
2> c(callback_mod).
callback_mod.erl:8: Warning: variable 'Req' is unused
callback_mod.erl:8: Warning: variable 'State' is unused
{ok,callback_mod}
3> {_, P1} = gen_replicated:start(5, callback_mod).
{ok,<0.47.0>}
4> gen_replicated:write(P1, 4).
{ok,[1,2,3,4,5,6,7,8,9],[1,2,3,99,5,6,7,8,9]}
5> gen_replicated:re
read/2     replica/2
5> gen_replicated:read(P1, 3).
{'ABORTED',exception,something_happened}

%%%%%%% TEST 6 %%%%%%%
% Test to see if the stop returned from handle_write works,
% while the read requests still work.

1> c(gen_replicated).
{ok,gen_replicated}
2> c(callback_mod).
callback_mod.erl:13: Warning: variable 'Req' is unused
callback_mod.erl:13: Warning: variable 'State' is unused
{ok,callback_mod}
```

```
3> {_, P1} = gen_replicated:start(5, callback_mod).
{ok,<0.47.0>}
4> gen_replicated:read(P1, 4).
{ok,4}
5> gen_replicated:read(P1, 3).
{ok,3}
6> gen_replicated:write(P1, 3).
{ok,stopped}

%%%%%%% TEST 7 %%%%%%%
% Test to see if the throw returned from handle_write works,
% while the read requests still work.

1> c(gen_replicated).
{ok,gen_replicated}
2> c(callback_mod).
callback_mod.erl:13: Warning: variable 'Req' is unused
callback_mod.erl:13: Warning: variable 'State' is unused
{ok,callback_mod}
3> {_, P1} = gen_replicated:start(5, callback_mod).
{ok,<0.47.0>}
4> gen_replicated:read(P1, 4).
{ok,4}
5> gen_replicated:write(P1, 4).
{'Aborted',exception,something_happened}
```

D Unit Tests

D.1 ParserTest.hs

```
module ParserTest where

import Test.HUnit
import SubsParser

introAST :: String
introAST = "Right (Prog [VarDecl \"xs\" (Just (Array [Number 0,Number 1,Number 2,Number 3,↩
    Number 4,Number 5,Number 6,Number 7,Number 8,Number 9])),VarDecl \"squares\" (Just (Compr↩
    (\"x\",Var \"xs\",Nothing) (Call \"*\" [Var \"x\",Var \"x\"]))),VarDecl \"evens\" (Just ↩
    (Compr (\"x\",Var \"xs\",Just (ArrayIf (Call \"===\" [Call \"%\" [Var \"x\",Number 2],↩
    Number 0]) Nothing)) (Var \"x\"))),VarDecl \"many_a\" (Just (Compr (\"x\",Var \"xs\",Just↩
    (ArrayForCompr (\"y\",Var \"xs\",Nothing))) (String \"a\"))),VarDecl \"hundred\" (Just (↩
    Compr (\"i\",Array [Number 0],Just (ArrayForCompr (\"x\",Var \"xs\",Just (ArrayForCompr ↩
    (\"y\",Var \"xs\",Nothing))))) (Assign \"i\" (Call \"+\" [Var \"i\",Number 1]))))])"

funNameAST :: String
funNameAST = "Right (Prog [VarDecl \"k\" (Just (Call \"one.two.three\" [Number 1,Number 2,↩
    Number 3]))])"

precedence1AST :: String
precedence1AST = "Right (Prog [ExprAsStm (Call \"+\" [Call \"-\" [Call \"+\" [Number 2,Call ↩
    \"*\" [Number 3,Number 4]],Number 1],Call \"%\" [Number 2,Number 2]])])"


precedence2AST :: String
precedence2AST = "Right (Prog [ExprAsStm (Call \"===\" [Call \"+\" [Number 7,Number 42],Call ↩
    \"<\" [Number 12,Number 99]])])"

associativityAST :: String
associativityAST = "Right (Prog [VarDecl \"k\" (Just (Assign \"l\" (Var \"m\")))])"

allowedIdentsAST :: String
allowedIdentsAST = "Right (Prog [VarDecl \"k\" (Just (Var \"_k02__\"))])"

illegalIdents :: String
illegalIdents = "Left \"Parsing error\" (line 1, column 10):\nunexpected \"f\"\nexpecting ↩
    digit, \"*\", \"%\", \"+\", \"-\", \"<\", \"===\" or \";\""

reservedIdents :: String
reservedIdents = "Left \"Parsing error\" (line 1, column 8):\nunexpected illegal identifier"

dotError :: String
dotError = "Left \"Parsing error\" (line 1, column 15):\nunexpected \" \""

-- Tests
introTest :: Test
introTest = TestCase (do parsedFile <- parseFile "tests/intro.js"
                         assertEqual "intro test file"
                                     introAST $ show parsedFile)
testSucc1 :: Test
testSucc1 = TestCase (do parsedFile <- parseFile "tests/funName"
                         assertEqual "funName and array test"
                                     funNameAST $ show parsedFile)

testSucc2 :: Test
testSucc2 = TestCase (do parsedFile <- parseFile "tests/precedence1"
                         assertEqual "+ / - / * / % precedence test"
                                     precedence1AST $ show parsedFile)

testSucc3 :: Test
testSucc3 = TestCase (do parsedFile <- parseFile "tests/precedence2"
                         assertEqual "=== / < precedence test"
                                     precedence2AST $ show parsedFile)

testSucc4 :: Test
testSucc4 = TestCase (do parsedFile <- parseFile "tests/associativity"
                         assertEqual "'=' associativity test"
                                     associativityAST $ show parsedFile)

testSucc5 :: Test
testSucc5 = TestCase (do parsedFile <- parseFile "tests/allowedIdents"
                         assertEqual "allowed idents (using '_') test"
                                     allowedIdentsAST $ show parsedFile)

testSucc6 :: Test
testSucc6 = TestCase (do parsedFile <- parseFile "tests/illegalIdents"
                         assertEqual "illegal ident starting with digit"
                                     illegalIdents $ show parsedFile)

testSucc7 :: Test
testSucc7 = TestCase (do parsedFile <- parseFile "tests/reservedIdents"
                         assertEqual "reserved ident"
```

```
                                              reservedIdents $ show parsedFile)

testSucc8 :: Test
testSucc8 = TestCase (do parsedFile <- parseFile "tests/dotError"
                         assertEqual "spaces after dot error"
                                       dotError $ show parsedFile)

tests:: Test
tests = TestList [TestLabel "intro test file" introTest,
                  TestLabel "funName test file" testSucc1,
                  TestLabel "*/%/+/- precedence test file" testSucc2,
                  TestLabel "== / < precedence test file" testSucc3,
                  TestLabel "'=' associativity test file" testSucc4,
                  TestLabel "allowed Ident test file" testSucc5,
                  TestLabel "illegal Ident test file" testSucc6,
                  TestLabel "reserved Ident test file" testSucc7,
                  TestLabel "spaces after dot error" testSucc8]
```

E Source Code

E.1 SubsParser.hs

```
module SubsParser (
  parseString,
  parseFile
) where

import SubsAst as S
import Text.Parsec
import Text.Parsec.String
import Control.Monad
import Data.Functor.Identity

{-data ParseError = AmbiguousParse
                | NoParse
  deriving (Show, Eq)-}

whitespace :: Parser ()
whitespace = void $ many $ oneOf " \n\t"

lexeme :: Parser a -> Parser a
lexeme p = do
           x <- p
           whitespace
           return x

-- Reserved
reserved :: [String]
reserved = ["var", "true", "false", "undefined", "for", "of", "if"]

-- Ident implementation
ident :: Parser String
ident = do
      s1 <- letters
      s2 <- many letDigs
      return (s1:s2)
    where
      letters = oneOf $ ['a'..'z'] ++ ['A'..'Z'] ++ "_"
      letDigs = oneOf $ ['a'..'z'] ++ ['A'..'Z'] ++ "_" ++ ['0'..'9']

identParse :: Parser String
identParse = do n <- ident
                if n `elem` reserved then unexpected "illegal identifier"
                else return n

-- Grammar Implementation
prog :: Parser Program
prog = do
      p <- stms
      _ <- eof
      return (Prog p)

stms :: Parser [Stm]
stms = try stms1 <|> try stms2

stms1 :: Parser [Stm]
stms1 = do
      s1 <- lexeme stm
      void $ lexeme $ char ';'
      s2 <- lexeme stms
      return (s1:s2)

stms2 :: Parser [Stm]
stms2 = do
      s1 <- lexeme stm
      void  $ lexeme $ char ';'
      return [s1]

stm :: Parser Stm
stm =  try varDecl <|> try exprAsStm

varDecl :: Parser Stm
varDecl = try varDecl1 <|> try varDecl2

varDecl1 :: Parser Stm
varDecl1 = do
      void $ lexeme $ string "var"
      iden <- lexeme identParse
      a1   <- lexeme assignOpt
      return (VarDecl iden (Just a1))

varDecl2 :: Parser Stm
varDecl2 = do
      void $ lexeme $ string "var"
```

```
      iden <- lexeme identParse
      return (VarDecl iden Nothing)

assignOpt :: Parser Expr
assignOpt = do
      void $ lexeme $ char '='
      e1 <- lexeme expr
      return e1

exprAsStm :: Parser Stm
exprAsStm = do
      e1 <- lexeme expr
      return (ExprAsStm e1)

expr :: Parser Expr
expr = try exprExt <|> try exprEnd

exprExt :: Parser Expr
exprExt = do
      e1 <- lexeme expr1
      void $ lexeme $ char ','
      e2 <- lexeme expr
      return (Comma e1 e2)

exprEnd :: Parser Expr
exprEnd = lexeme expr1

-- Precedence for expressions
expr1 :: Parser Expr
expr1 = do
      rv <- r
      botbot rv

botbot :: Expr -> ParsecT String () Data.Functor.Identity.Identity Expr
botbot inval = do
      void $ lexeme $ string "==="
      rv <- r
      botbot (Call "===" [inval, rv])
  <|> return inval

r :: ParsecT String () Data.Functor.Identity.Identity Expr
r = do
      sv <- s
      bot sv

bot :: Expr -> ParsecT String () Data.Functor.Identity.Identity Expr
bot inval = (do
      void $ lexeme $ char '<'
      sv <- s
      bot (Call "<" [inval, sv]))
  <|> return inval

s :: ParsecT String () Data.Functor.Identity.Identity Expr
s = do
      tv <- t
      mid tv

mid :: Expr -> ParsecT String () Data.Functor.Identity.Identity Expr
mid inval = (do
      void $ lexeme $ char '+'
      tv <- t
      mid (Call "+" [inval, tv]))
  <|> (do
      void $ lexeme $ char '-'
      tv <- t
      mid (Call "-" [inval, tv]))
  <|> return inval

t :: ParsecT String () Data.Functor.Identity.Identity Expr
t = do
      pv <- prim
      top pv

top :: Expr -> ParsecT String () Data.Functor.Identity.Identity Expr
top inval = (do
      void $ lexeme $ char '*'
      pv <- prim
      top (Call "*" [inval, pv]))
  <|> (do
      void $ lexeme $ char '%'
      pv <- prim
      top (Call "%" [inval, pv]))
  <|> return inval

-- Try expressions
prim :: Parser Expr
prim = try number <|> try trueConst <|> try falseConst <|> try undef
```

```
      <|> try exprString <|> try array <|> try exprIdent <|> try parentheses

number :: Parser Expr
number = do
      d <- lexeme $ many1 digit
      return (Number (read d))

exprString :: Parser Expr
exprString = do
      spaces
      void $ char '\''
      ss <-  many1 letter
      void $ char '\''
      spaces
      return (String ss)

array :: Parser Expr
array = try array1 <|> try array2

array1 :: ParsecT String () Identity Expr
array1 = do
      void $ lexeme $ char '['
      e1 <- lexeme exprs
      void $ lexeme $ char ']'
      return (Array e1)

array2 :: Parser Expr
array2 = try array3 <|> try array4

array3 :: Parser Expr
array3 = do
      void $ lexeme $ char '['
      void $ lexeme $ string "for"
      void $ lexeme $ char '('
      iden1 <- lexeme identParse
      void $ lexeme $ string "of"
      e1   <- lexeme expr
      void $ lexeme $ char ')'
      a1   <- lexeme arrayCompr
      e2   <- lexeme expr
      void $ lexeme $ char ']'
      return (Compr (iden1, e1, Just a1) e2)

array4 :: Parser Expr
array4 = do
      void $ lexeme $ char '['
      void $ lexeme $ string "for"
      void $ lexeme $ char '('
      iden2 <- lexeme identParse
      void $ lexeme $ string "of"
      e1   <- lexeme expr
      void $ lexeme $ char ')'
      e2   <- lexeme expr
      void $ lexeme $ char ']'
      return (Compr (iden2, e1, Nothing) e2)

parentheses :: Parser Expr
parentheses = do
      void $ lexeme $ char '('
      e1 <- lexeme expr
      void $ lexeme $ char ')'
      return e1

undef :: Parser Expr
undef = do
      void $ lexeme $ string "undefined"
      return Undefined

trueConst :: Parser Expr
trueConst = do
      void $ lexeme $ string "true"
      return TrueConst

falseConst :: Parser Expr
falseConst = do
      void $ lexeme $ string "false"
      return FalseConst

exprIdent :: Parser Expr
exprIdent = do
      spaces
      iden <- lexeme identParse
      afterIdent iden

afterIdent :: Ident -> Parser Expr
afterIdent iden = try (afterIdent1 iden)
  <|> try (funCall iden)
```

```
  <|> return (Var iden)

afterIdent1 :: Ident -> Parser Expr
afterIdent1 iden =  do
      spaces
      void $ lexeme $ char '='
      e1 <- lexeme expr1
      return (Assign iden e1)

funCall :: FunName -> ParsecT String () Identity Expr
funCall iden = try (funCall1 iden) <|> try (funCall2 iden)

funCall1 :: FunName -> ParsecT String () Identity Expr
funCall1 iden = do
      void $ char '.'
      idAdd <- identParse
      let newId = iden ++ "." ++ idAdd
      funCall newId

funCall2 :: FunName -> ParsecT String () Identity Expr
funCall2 iden = do
      void $ lexeme $ char '('
      e1 <- lexeme exprs
      void $ lexeme $ char ')'
      return (Call iden e1)

exprs :: Parser [Expr]
exprs = expr1 `sepBy` lexeme (char ',')

arrayCompr :: Parser ArrayCompr
arrayCompr = try for1 <|> try for2 <|> try if1 <|> try if2

for1 :: Parser ArrayCompr
for1 = do
      void $ lexeme $ string "for"
      void $ lexeme $ char '('
      iden <- lexeme identParse
      void $ lexeme $ string "of"
      e1   <- lexeme expr
      void $ lexeme $ char ')'
      a1   <- lexeme arrayCompr
      return (ArrayForCompr (iden, e1, Just a1))

for2 :: Parser ArrayCompr
for2 = do
      void $ lexeme $ string "for"
      void $ lexeme $ char '('
      iden <- lexeme identParse
      void $ lexeme $ string "of"
      e1   <- lexeme expr
      void $ lexeme $ char ')'
      return (ArrayForCompr (iden, e1, Nothing))

if1 :: Parser ArrayCompr
if1 = do
      void $ lexeme $ string "if"
      void $ lexeme $ char '('
      e1 <- lexeme expr
      void $ lexeme $ char ')'
      a1 <- lexeme arrayCompr
      return (ArrayIf e1 (Just a1))

if2 :: Parser ArrayCompr
if2 = do
      void $ lexeme $ string "if"
      void $ lexeme $ char '('
      e1 <- lexeme expr
      void $ lexeme $ char ')'
      return (ArrayIf e1 Nothing)

parseString :: String -> Either ParseError Program
parseString = runParser prog () "Parsing error"

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

E.2 SubsInterpreter.hs

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

import SubsAst

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

-- | A value is either an integer, the special constant undefined,
--   true, false, a string, or an array of values.
-- Expressions are evaluated to values.

-- ^ Any runtime error.  You may add more constructors to this type
-- (or remove the existing ones) if you want.  Just make sure it is
-- still an instance of 'Show' and 'Eq'.
data Value = IntVal Int
           | UndefinedVal
           | TrueVal | FalseVal
           | StringVal String
           | ArrayVal [Value]
           deriving (Eq, Show)

data Error = Error String
  deriving (Show, Eq)

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

initialContext :: Context
initialContext = (Map.empty, initialPEnv)
  where initialPEnv =
          Map.fromList [ ("===", isEqual)
                       , ("<", lessThan)
                       , ("+", plus)
                       , ("*", multiply)
                       , ("-", minus)
                       , ("%", modulo)
                       , ("Array.new", arrayNew)
                       ]

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

instance Functor SubsM where
  fmap = Control.Monad.liftM

instance App.Applicative SubsM where
  pure = return
  (<*>) sf s = sf >>= \f -> fmap f s

instance Monad SubsM where
  return a = SubsM (\c -> Right (a, fst c))
  (>>=) m f = SubsM (\ (env, pEnv) -> case runSubsM m (env, pEnv) of
                      Left str        -> Left str
                      Right (a, env') -> runSubsM (f a) (env', pEnv))
  fail s = SubsM (\_ -> Left (Error s))

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

isEqual :: Primitive
isEqual [IntVal e1, IntVal e2]       = return (if e1 == e2 then TrueVal
                                       else FalseVal)
isEqual [StringVal e1, StringVal e2] = return (if e1 == e2 then TrueVal
                                       else FalseVal)
isEqual [TrueVal, FalseVal]          = return FalseVal
isEqual [FalseVal, TrueVal]          = return FalseVal
isEqual [TrueVal, TrueVal]           = return TrueVal
isEqual [FalseVal, FalseVal]         = return TrueVal
isEqual _                            = fail "Bad argument types in '==='"

lessThan :: Primitive
lessThan [IntVal e1, IntVal e2]       = return (if e1 < e2 then TrueVal
                                        else FalseVal)
lessThan [StringVal e1, StringVal e2] = return (if e1 < e2 then TrueVal
```

```
                                          else FalseVal)
lessThan _                               = fail "Bad argument types in '<'"

plus :: Primitive
plus [IntVal e1, IntVal e2]        = return $ IntVal (e1 + e2)
plus [StringVal e1, StringVal e2] = return $ StringVal (e1 ++ e2)
plus [StringVal e1, IntVal e2]     = return $ StringVal (e1 ++ show e2)
plus [IntVal e1, StringVal e2]     = return $ StringVal (show e1 ++ e2)
plus _                             = fail "Bad argument types in '+'"

multiply :: Primitive
multiply [IntVal e1, IntVal e2] = return $ IntVal (e1 * e2)
multiply _                      = fail "Both operators must be integers in '*'"

minus :: Primitive
minus [IntVal e1, IntVal e2] = return $ IntVal (e1 - e2)
minus _                      = fail "Both operators must be in '-'"

modulo :: Primitive
modulo [IntVal e1, IntVal e2] = return $ IntVal (e1 `mod` e2)
modulo _                      = fail "Both operator must be integers in '%'"

-- End initial Context

-- Utility Functions
getEnv :: SubsM Env
getEnv = SubsM (\(env, _) -> Right (env, env))

getPEnv :: SubsM PEnv
getPEnv = SubsM (\(env, penv) -> Right (penv, env))

putEnv :: Env -> SubsM ()
putEnv env = SubsM (\_ -> Right((), env))

{-
modify :: (Env -> Env) -> SubsM ()
modify f = do
  env <- getEnv
  putEnv (f env)
-}

updateEnv :: Ident -> Value -> SubsM ()
updateEnv name val = do
  env <- getEnv
  putEnv (Map.insert name val env)

getVar :: Ident -> SubsM Value
getVar name = do
  env <- getEnv
  case Map.lookup name env of
    Just a -> return a
    Nothing -> fail "Key does not exist in 'getVar'"

getFunction :: FunName -> SubsM Primitive
getFunction name = do
  penv <- getPEnv
  case Map.lookup name penv of
    Just a -> return a
    Nothing -> fail "Key does not exist in 'getFunction'"

-- End Utility Functions

-- Expr and Stms Evaluation
evalExpr :: Expr -> SubsM Value
evalExpr (Number n)            = return $ IntVal n
evalExpr (String s)            = return $ StringVal s
evalExpr (Array arr)           = do
  a <- arrEval arr
  return $ ArrayVal a
evalExpr (TrueConst)           = return TrueVal
evalExpr (FalseConst)          = return FalseVal
evalExpr (Undefined)           = return UndefinedVal
evalExpr (Var iden)            = getVar iden
evalExpr (Compr _ _)           = undefined
evalExpr (Call f arr)          = do
  fun <- getFunction f
  a <- arrEval arr
  fun a
evalExpr (Assign iden e1)      = do
  e <- evalExpr e1
  updateEnv iden e
  return e
evalExpr (Comma e1 e2)         = do
  _ <- evalExpr e1
  evalExpr e2

arrEval :: [Expr] -> SubsM [Value]
```

```
arrEval []         = return []
arrEval (a : arr) = do
  v  <- evalExpr a
  vs <- arrEval arr
  return (v : vs)

stm :: Stm -> SubsM ()
stm (VarDecl iden Nothing)    = updateEnv iden UndefinedVal
stm (VarDecl iden (Just e1)) = do
  e <- evalExpr e1
  updateEnv iden e
stm (ExprAsStm e1)           = do
  _ <- evalExpr e1
  return ()

program :: Program -> SubsM ()
program (Prog [])        = return ()
program (Prog (p:prog)) = do
  _  <- stm p
  program (Prog prog)

-- End Expr and Stms Evaluation

runProg :: Program -> Either Error Env
runProg prog =
  case runSubsM (program prog) initialContext of
    Left str         -> Left str
    Right (_, env)  -> Right env
```

E.3 gen_replicated.erl

```erlang
-module(gen_replicated).
-export([start/2, stop/1, read/2, write/2, coordinator/1, replica/2, get_pids/1]).

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

start(NumReplica, _) when NumReplica < 2 -> {error, "NumReplica < 2"};
start(NumReplica, Mod) ->
  case Mod:init() of
    {ok, _}         -> ServerRef = spawn_link(fun() -> coordinator([]) end),
                       ServerRef ! {make_replicas, NumReplica, Mod},
                       {ok, ServerRef};
    {error, Reason} -> {error, Reason}
  end.

stop(Server) ->
  Server ! {self(), stop},
  receive
    {stopped, _} -> {ok, stopped}
  end.

read(Server, Req) ->
  Server ! {self(), {read, Req}},
  receive
    {ok, Result} -> {ok, Result};
    stop         -> stop(Server);
    {throw, Val} -> {'ABORTED', exception, Val}
  end.

write(Server, Req) ->
  blocking(Server, Req).

get_pids(Server) ->
  Server ! {self(), getPids},
  receive
    {ok, Pids} -> {ok, Pids}
  end.

%Helper Functions
blocking(Server, Req) ->
  Server ! {self(), {write, Req}},
  receive
    {ok, Reply}           -> {ok, Reply};
    {ok, Reply, NewState} -> {ok, Reply, NewState};
    stop                  -> stop(Server);
    {throw, Val}          -> {'Aborted', exception, Val}
  end.

createReplica(NumReplica, _) when NumReplica == 0 -> [];
createReplica(NumReplica, Mod) ->
  {_, State} = Mod:init(),
  ReplicaRef = spawn_link(fun() -> replica(State, Mod) end),
  [{ReplicaRef, free} | createReplica(NumReplica-1, Mod)].

%findAvailable(N, _) when N == 1 -> fail;
%findAvailable(N, Pids) ->
%  case element(2, lists:nth(N, Pids)) of
%    free -> element(1, lists:nth(N, Pids));
%    busy -> findAvailable(N-1, Pids)
%  end.

%Coordinator and Replicas
coordinator(Pids) ->
  receive
    {From, getPids} ->
      From ! {ok, Pids},
      coordinator(Pids);
    {make_replicas, NumReplica, Mod} ->
      NewPids = createReplica(NumReplica, Mod),
      coordinator(NewPids);
    {From, {read, Req}} ->
      Reader = element(1, lists:nth(1+random:uniform(length(Pids)-1), Pids)),
      Reader ! {From, {read, Req}},
      coordinator(Pids);
    {From, {write, Req}} ->
      Reader = element(1, lists:nth(1, Pids)),
      Reader ! {self(), From, {write, Req}},
      coordinator(Pids);
    {update, NewState} ->
      Msg = {update, NewState},
      [element(1, Replica) ! Msg || Replica <- Pids],
      coordinator(Pids);
    {From, stop} ->
      Msg = {From, stop},
```

```
      [element(1, Replica) ! Msg || Replica <- Pids]
  end.

replica(State, Mod) ->
  receive
    {From, {read, Req}} ->
      try Mod:handle_read(Req, State) of
        {reply, Reply} -> From ! {ok, Reply},
                          replica(State, Mod);
        stop           -> From ! stop
      catch
        throw : Val    -> From ! {throw, Val}
      end;
    {Coordinator, From, {write, Req}} ->
      try Mod:handle_write(Req, State) of
        {noUpdate, Reply}           -> From ! {ok, Reply},
                                       replica(State, Mod);
        {updated, Reply, NewState} -> Coordinator ! {update, NewState},
                                       From ! {ok, Reply, NewState},
                                       replica(NewState, Mod);
        stop                        -> From ! stop
      catch
        throw : Val                 -> From ! {throw, Val}
      end;
    {update, UpdatedState} ->
      replica(UpdatedState, Mod);
    {From, stop} ->
      From ! {stopped, self()}
  end.
```

E.4 alzheimer

```
-module(alzheimer).
-behavior(gen_server).
-export([start/0, upsert/3, query/2, testFun/1]).
-export([init/1, handle_call/3]).

-define(SERVER, ?MODULE).
%%===================================================================
%% API
%%===================================================================


% Client Interface Functions
start() ->
  gen_server:start_link(?MODULE, [], []).

query(Aid, P) ->
  gen_server:call(Aid, {read, P}).

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

% Helper Function
filterWithError(_, List, Rows) when length(List) == 0 -> {ok, Rows};
filterWithError(P, [Elem | List], Rows) ->
  Elem = lists:nth(1, List),
  try P(Elem) of
    true  -> filterWithError(P, List, lists:append([Rows, [Elem]]));
    false -> filterWithError(P, List, Rows)
  catch
    _ : _ -> filterWithError(P, [], lists:append([[{error, element(2, Elem)}], Rows]))
  end.

% Callback Functions
init(_Args) ->
  {ok, dict:new()}.

handle_call(Req, From, State) ->
  case Req of
    {read, P} -> case filterWithError(P, dict:to_list(State), []) of
                   {ok, Rows} -> case element(1, Rows) of
                                   {error, Row} -> {error, Row};
                                   _            -> {ok, Rows}
                                 end
                 end;
    {write, Id, F} -> case dict:find(Id, State) of
                        {ok, Value} -> try F({existing, {Id, Value}}) of
                                         {modify, NewData} -> dict:store(Id, NewData, State),
                                                              {modify, NewData};
                                         ignore            -> ignore
                                       catch
                                         throw : Reason    -> throw(Reason)
                                       end;
                        error       -> try F({new, Id}) of
                                         {modify, NewData} -> dict:store(Id, NewData, State),
                                                              {modify, NewData};
                                         ignore            -> ignore
                                       catch
                                         throw : Reason    -> throw(Reason)
                                       end
                      end
  end.

% Test Functions
testFun(X) ->
  case X of
    {new, _}               -> {modify, 99};
    {existing, {_, _}} -> ignore
  end.
```