Take-home Exam in Advanced Programming

Deadline: Thursday, November 5, 16:00

Version 1.3

Preamble

This is the exam set for the individual, written take-home exam on the course Advanced Programming, B1-2015. This document consists of 16 pages; make sure you have them all. Please read the entire preamble carefully.

The exam set consists of 4 questions. Your solution will be graded as a whole, on the 7-point grading scale, with an external examiner.

In the event of errors or ambiguities in the exam set, you are expected to state your assumptions as to the intended meaning in your report. You may ask for clarifications in the discussion forum on Piazza, but do not expect an immediate reply. If there is no time to resolve a case, you should proceed according to your chosen (documented) interpretation.

What To Hand In

To pass this exam you must hand in both a report and your source code:

- *The report* should be around 5–10 pages, not counting appendices, presenting (at least) your solutions, reflections, and assumptions, if any. The report should contain all your source code in appendices. The report must be a PDF document.
- *The source code* should be in a .ZIP file, archiving one directory called `src` (which may contain further subdirectories).

Make sure that you follow the format specifications (PDF and .ZIP). If you don't, the hand in **will not be assessed** and treated as a blank hand in. The hand in is done via the course web page on Absalon.

Learning Objectives

To get a passing grade you must demonstrate that you are both able to program a solution using the techniques taught in the course *and* write up your reflections and assessments of your own work.

- For each question your report should give an overview of your solution, **including an assessment** of how good you think your solution is and on which grounds you base your assessment. Likewise, it is important to document all *relevant* design decisions you have made.
- In your programming solutions emphasis should be on correctness, on demonstrating that your have understood the principles taught in the course, and on clear separation of concerns.
- It is important that you implement the required API, as your programs might be subjected to automated testing as part of the grading. Failure to implement the correct API may influence your grade.
- To get a passing grade, you *must* have some working code in both Haskell and Erlang.

Exam Fraud

The exam is an individual exam, thus you are **not** allowed to discuss any part of the exam with anyone on, or outside the course.

You are only allowed to ask, *not answer*, how a question is to be interpreted on the course discussion forum on Piazza. If you are afraid that your question contains too many technical details, then use the private message feature on Piazza.

Specifically, but not exclusively, you are **not** allowed to discuss any part of the exam with any other student nor to copy parts of other students' programs. Submitting answers you have not written yourself, or *sharing your answers with others*, is considered exam fraud.

This is an open-book exam, and so you are welcome to make use of any reading material from the course, or elsewhere. Make sure to use proper academic citation for the material you draw considerable inspiration from (including what you may find on the Internet, for example, snippets of code). Also note, that these rules mean that it is not allowed to copy any part of the exam set (or supplementary skeleton files) and publish it on forums other than the course discussion forum (e.g., StackOverflow, IRC, exam banks, chatrooms, or suchlike).

During the exam period, students are not allowed to answer questions, *only teachers and teaching assistants are allowed to answer questions* on the discussion forum.

Breaches of the above policy will be handled in accordance with the Faculty of Science's disciplinary procedures.

Emergency Webpage

There is an emergency web page at

`http://www.diku.dk/~kflarsen/ap-e2015/`

in case Absalon becomes unstable. The page will describe what to do if Absalon becomes unreachable during the exam, especially what to do at the hand-in deadline.

SubScript

JavaScript is perhaps the most widely available programming language in the world: nearly every personal computer and handheld device, connected to the Internet, has some sort of JavaScript engine installed, as part of the web browser engine, or otherwise.

Perhaps to make JavaScript more comprehensible, a recurring proposal is to add array comprehensions to JavaScript. One implementation is the one found in Firefox version 30, or greater[1].

The following two questions is about implementing a conservative subset of Mozilla's JavaScript implementation called SubScript.

Hint: Use the Firefox Web Console Developer Tool[2] to get to a Mozilla JavaScript prompt. This is a simple way to play around with JavaScript array comprehensions until you have SubScript at your disposal.

For instance, consider an array of numbers:

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

You can get the array of the squares of the numbers as follows:


```
[ for (x of xs) x * x ];
```

You can also filter the array according to a predicate and get a, perhaps, smaller array. For instance, to get all the even numbers in an array of numbers:


```
[ for (x of xs) if (x % 2 === 0) x ];
```


You can also perform nested iterations, and generate larger arrays. For instance, to repeat an element (in this case) 100 times:

```
[ for (x of xs) for (y of xs) 'a' ];
```

Or, to generate (in this case) 100 consecutive integers, starting at 1: Ver. 1.2

```
[ for (i of [0]) for (x of xs) for (y of xs) i = i + 1 ];
```

[1] For more details on the Firefox implementation of JavaScript array comprehensions, see https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Array_comprehensions.

[2] https://developer.mozilla.org/en/docs/Tools/Web_Console

Question 1: SubScript, Parser

To ease development, we will omit many núances of JavaScript from SubScript. In the grammar below you will notice that many constructions valid in JavaScript are not valid in SubScript.

Grammar

Program	::=	*Stms*	
Stms	::=	ϵ	
	|	*Stm* `';'` *Stms*	
Stm	::=	`'var'` *Ident* *AssignOpt*	
	|	*Expr*	
AssignOpt	::=	ϵ	
	|	`'='` *Expr1*	Ver. 1.3
Expr	::=	*Expr* `','` *Expr*	
	|	*Expr1*	
Expr1	::=	*Number*	
	|	*String*	
	|	`'true'`	
	|	`'false'`	
	|	`'undefined'`	
	|	*Expr1* `'+'` *Expr1*	Ver. 1.1
	|	*Expr1* `'-'` *Expr1*	Ver. 1.1
	|	*Expr1* `'*'` *Expr1*	Ver. 1.1
	|	*Expr1* `'%'` *Expr1*	Ver. 1.1
	|	*Expr1* `'<'` *Expr1*	Ver. 1.1
	|	*Expr1* `'==='` *Expr1*	Ver. 1.1
	|	*Ident* *AfterIdent*	
	|	`'['` *Exprs* `']'`	
	|	`'['` `'for'` `'('` *Ident* `'of'` *Expr1* `')'` *ArrayCompr* *Expr1* `']'`	Ver. 1.3
	|	`'('` *Expr* `')'`	
AfterIdent	::=	ϵ	
	|	`'='` *Expr1*	Ver. 1.1
	|	*FunCall*	
FunCall	::=	`'.'` *Ident* *FunCall*	
	|	`'('` *Exprs* `')'`	
Exprs	::=	ϵ	
	|	*Expr1* *CommaExprs*	
CommaExprs	::=	ϵ	
	|	`','` *Expr1* *CommaExprs*	
ArrayCompr	::=	ϵ	
	|	`'if'` `'('` *Expr1* `')'` *ArrayCompr*	Ver. 1.3
	|	`'for'` `'('` *Ident* `'of'` *Expr1* `')'` *ArrayCompr*	Ver. 1.3

JavaScript identifiers have a very liberal syntax. For simplicity, we stay a bit more conser-

vative: a SubScript *Ident* begins with either an alphabetic ASCII character or _, and is followed by zero or more *alphanumeric* ASCII characters or _. For instance, both `x0`, `_x0`, `x_0`, and even _ and `___`, are all legal identifiers in SubScript; while such legal JavaScript identifiers as `$$$` and `\u17708` are illegal.

Last, but not least, an identifier cannot be one of the keywords otherwise used in SubScript, i.e. not `var`, `true`, `false`, `undefined`, `for`, `of`, or `if`. Using other JavaScript keywords is allowed in SubScript, but this might hamper the testability of your scripts.

All numbers in JavaScript are IEEE-754 double-precision floating point numbers (doubles). Dealing in doubles is beyond the scope of SubScript: A *Number* is a non-empty sequence of at most 8 digits, optionally preceded by a minus sign, indicating a negative number[3].

A SubScript *String* is a sequence of non-single-quote characters surrounded by single quotes. In real JavaScript, strings can both be surrounded by double quotes, and have escape sequences. This is great fun to parse at your leisure, but keeps array comprehensions at bay, so we took it out of SubScript.

The operator `'%'` is the (usual) integer modulo operator, and in particular, *not* the IEEE-754 double-precision floating point modulo operator. (This is in lieu with JavaScript.)

Table 1 presents the precedences and associativity of the operators in SubScript. Note, due to the restrictive syntax of SubScript, the precedence of `'.'` is irrelevant.

Precedence	Operator(s)	Associativity
-	`'.'`	left
5	`'*'` `'%'`	left
4	`'+'` `'-'`	left
3	`'<'`	left
2	`'==='`	left
1	`'='`	right
0	`','`	left

Table 1: Operator precedence and associativity in SubScript.

All tokens, except `'.'` and `'var'` are separated by arbitrary whitespace. `'.'` must be surrounded by non-whitespace characters. `'var'` must be immediately followed by a whitespace character.

What to implement

You should implement a module `SubsParser` with the following interface.

A function `parseString` for parsing a SubScript program given as a string:

```
parseString :: String -> Either ParseError Program
```

[3]Every integer in the range $[-\underbrace{99999999}_{8\text{ digits}};\underbrace{99999999}_{8\text{ digits}}]$ is both exactly representable as an IEEE-754 double-precision floating point number, and fits in a Haskell `Int`.

Where you decide and specify what the type `ParseError` should be, the only requirement is that it must be an instance of `Show` and `Eq`. The type `ParseError` must also be exported from the module. The handed-out skeleton code already has the exports set up correctly.

Likewise, you should implement a function `parseFile` for parsing a SubScript program given in a file located at a given path:

```
parseFile :: FilePath -> IO (Either ParseError Program)
```

Where `ParseError` is the same type as for `parseString`.

The type `Program` is defined in the handed out `SubsAst.hs`. We list this module below for quick reference. You should not change the types for the abstract syntax tree unless there is an update on Absalon telling you explicitly that you can do so.

```
module SubsAst where

data Program = Prog [Stm]
             deriving (Show, Eq)

data Stm = VarDecl Ident (Maybe Expr)
         | ExprStm Expr
         deriving (Show, Eq)

data Expr = Number Int
          | String String
          | Array [Expr]
          | Undefined
          | TrueConst
          | FalseConst
          | Var Ident
          | Compr ArrayFor Expr
          | Call FunName [Expr]
          | Assign Ident Expr
          | Comma Expr Expr
          deriving (Show, Eq)

type ArrayFor = (Ident, Expr, Maybe ArrayCompr)

data ArrayCompr = ArrayForCompr ArrayFor
                | ArrayIf Expr (Maybe ArrayCompr)
                deriving (Eq, Show)

type Ident = String
type FunName = String
```


As you can see, there are no elements of the abstract syntax tree for representing the arithmetical operators `'+'`, `'-'`, `'*'`, or `'%'`, nor is there any dedicated way of

representing the relational operators `'<'` or `'==='`. These should be translated into calls to functions built into your interpreter. For instance, should the expression `38 + 4` be translated to `Call "+" [Number 38, Number 4]`.

You must use one of the three monadic parser libraries, `SimpleParse.hs`, `ReadP` or Parsec to implement your parser. You will find Haskell skeletons for the parser and abstract syntax tree on Absalon.

If you use Parsec, then only plain Parsec is allowed, namely the following submodules of `Text.Parsec`: `Prim`, `Char`, `Error`, `String`, and `Combinator` (or the compatibility modules in `Text.ParserCombinators`), in particular you are *disallowed* to use `Text.Parsec.Token`, `Text.Parsec.Language`, and `Text.Parsec.Expr`.

Together with your parser you must also hand in a test-suite to show that your parser works (or where it does not work). That is, that it correctly parses valid programs and rejects invalid programs.

Question 2: SubScript, Interpreter

This question is about writing an interpreter for the SubScript subset of JavaScript. The intention is that the semantics of a syntactically correct SubScript program is mostly the same as if it would be as interpreted by a standard JavaScript interpreter up to a few simplifications with respect to type coercions, which is detailed in the following.

We recommend that you read through the whole question before you start implementing anything, and read the examples in the introduction and in Appendix A, which also contains abstract syntax trees if your parser is not completely working.

Semantics of SubScript

The semantics of most of SubScript should be straightforward, below some of the more murky points are elaborated.

- Variables can only be referred to after they have been declared (thus there are no recursive declarations). Ver. 1.1
- Variables can be redeclared and will shadow earlier declarations.
- The value of an assignment expression, `x = e`, is the value of the right-hand side, `e`.
- The comma operator, `e1, e2`, evaluates each of its operands (from left to right) and returns the value of the last operand, `e2`.
- In contrast to JavaScript there are only limited type coercions in SubScript. Thus, it is illegal to, say, multiply an integer and an boolean (which is legal in JavaScript).

  Both arguments to arithmetic operators must be integers. The only exception to this rule is for addition, where it is possible to add two strings or a string and a number, in the later case the number should be converted to a string, addition of strings means string concatenation. Ver. 1.3 Ver. 1.3

  Likewise, both arguments to the `===` operator should have the same type, and the two arguments to the `<` operator should either both be integers or both be strings, strings are compared using the usual lexicographic order on strings. Ver. 1.3

  As SubScript computations are dynamically typed there are several ways type errors can occur. From the description of the syntax it is possible to infer what these type error conditions are, generally the only type coercion allowed is the one described for addition everything else is an error. If you are in doubt document your interpretation succinctly in your report. Ver. 1.3
- The build-in function `Array.new(n)` is used for generating an array with `n` elements all with the special value `undefined`. The argument `n` should be strictly larger than zero, otherwise it is considered an error.
- An array comprehension

  `[for (`n `of` e_1`)` *iter* e_2`]`

consists of at least one `for` clause and zero or more `for` or `if` clauses and finally a single expression. The result of an array comprehension is an array where the elements of the array are produced by evaluating e_2 in different environments with n bound to each element in e_1 (hence, e_1 must be either an array or a string) plus the bindings stemming from *iter*. In the case that e_1 is a string, the binding for the variable n is bound to a one character string for each character in e_1.

Similar to the Haskell expression

$$[e_2 \mid n \texttt{<-} e_1, \mathit{iter}_H]$$

where each `if` clause in *iter* is a boolean expression in iter_H and each `for` clause in *iter* is a binding in iter_H.

Note that, the bindings from `for` clauses are only in scope in the array comprehension nested left to right. For example, the binding of n is available in `iter` (and in e_2), but the binding from *iter* is not available in e_1 (but they are in e_2), and bindings from nested `for` clauses can shadow outer bindings.

- The result of evaluation a SubScript expression is a value which is either an integer, a boolean, the special value `undefined`, a string, or an array of values. We represent values by the following Haskell data type:


```
data Value = IntVal Int
           | UndefinedVal
           | TrueVal | FalseVal
           | StringVal String
           | ArrayVal [Value]
           deriving (Eq, Show)
```


which should be declared in `SubsInterpreter.hs`.

If your interpreter encounters an error, it should terminate with an well-defined error type. That is, **not** by calling the built-in Haskell function `error`.

Your Task

The main objective of this question is that you should demonstrate that you know how to write an interpreter using monads for structuring your code. Thus you should structure your solution along the following lines, where you most likely also need a few extra helper functions:

(a) Define a module `SubsInterpreter` that exports a function `runProg`, the type `Value` and a type `Error`.

See the handed-out `SubsInterpreter.hs` for a *strongly recommended* skeleton for your solution. The handed-out skeleton code already has the exports set up correctly.

(b) During the interpretation of a SubScript program we need to keep track of a context for the statements and expressions to be executed in. The context consists of two

parts: (1) a variable environment mapping variable names to values; and (2) a read-only primitives environment mapping names of build-in functions and operators (primitives) to Haskell functions implementing their semantics. That is, we use the following types:

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

where `Map` is from the `Data.Map` library and the `SubsM` type is described in the following. These types are already declared in `SubsInterpreter.hs`.

(c) We use the type `SubsM` for structuring our interpreter:

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

Make the `SubsM` type a `Monad` instance (and a `Functor` and `Applicative` instances as well).

You decide what the `Error` type should be, just make sure that it is an instance of `Show` and `Eq`.

(d) In the initial context, `initialContext`, we have an empty variable environment, and a primitives environment with seven entries. Finish the following implementation:


```
initialContext :: Context
initialContext = (Map.empty, initialPEnv)
  where initialPEnv =
          Map.fromList [ ("===", undefined)
                       , ("<", undefined)
                       , (">", undefined)
                       , ("*", undefined)
                       , ("-", undefined)
                       , ("%", undefined)
                       , ("Array.new", arrayNew)
                       ]
```

Where the primitive `arrayNew`, for example, is implemented by the following function:

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

(e) Implement the following utility functions for working with the context:

```
modify      :: (Env -> Env) -> SubsM ()
updateEnv   :: Ident -> Value -> SubsM ()
getVar      :: Ident -> SubsM Value
getFunction :: FunName -> SubsM Primitive
```

(f) Implement a functions for evaluating expressions and statements:

```
evalExpr :: Expr -> SubsM Value
stm :: Stm -> SubsM ()
```

(g) Define a function `runProg`

```
runProg :: Program -> Either Error Env
```

`runProg` p runs program p in the initial context, yielding either a runtime error, or the result of the program. The result of evaluating a JavaScript program is the environment mapping variable names to values.

Putting it all together

Once you have implemented the parser and interpreter, the file `Subs.hs` can be used to run SubScript programs, as follows.

```
$ runhaskell Subs.hs program.js
```

You should *not* need to modify `Subs.hs`.

Advice for your solution

Getting array comprehension right is the most difficult part of this question. Thus, they alone weight with 25% of this question. Hence, if you have difficulties making your interpreter work for the full SubScript language, then start by making it work for the subset of the language with array comprehensions left out.

Then proceed by, for instance, allowing only one `for` clause in array comprehensions, or disallowing `if` clauses in array comprehensions, and so on.

If you make such restrictions make sure to clearly documenting them in your assessment, and explain why the disallowed language constructs cause you problems.

Also, make sure that you have tested your solution and that your testing is automated, so that we can run your tests and verify your results.

Question 3: Generic Replicated Server Library

This question is about making a library, `gen_replicated`, that handles the generic parts of implementing replicated servers that can handle multiple concurrent readers and a single writer. Similar to how `gen_server` is used to write generic servers following the OTP guidelines.

A generic replicated server consists of a *coordinator* and a number of *replica*. The coordinator takes care of starting and stopping replica, routes read operations to replica, coordinates that there is only one concurrent write operation, and that all replica are brought up to date after a write operation. The actual implementation of the read and write operations are handled by a so-called behaviour or callback module, similar to `gen_server`.

The `gen_replicated` module should export the following API:

- `start(NumReplica, Mod)` for starting a replicated server, with `NumReplica` of replica processes, and callback module `Mod`. Returns `{ok, ServerRef}` on success or `{error, Reason}` if some error occurred.
- `stop(ServerRef)` for stopping the coordinator and all the replica. Clients waiting for a read or write request should get the value `{'ABORTED', server_stopped}` returned from the `read` and `write` functions. VER. 1.3
- `read(ServerRef, Request)` for sending a read request to a replicated server. The coordinator will forward the request to one of the replica. The return value is `{ok, Result}` where `Result` is the result from calling the `Mod:handle_read` function. If the `Mod:handle_read` call raises a `throw` exception with value `Val`, then this function should return `{'ABORTED', exception, Val}`.
- `write(ServerRef, Request)` for sending a write request to a replicated server. The coordinator will wait until there are no ongoing read nor write requests, and then forward the request to one of the replica. The return value is `{ok, Result}` where `Result` is reply from the `Mod:handle_write` function. If the write request resulted in a new state then all replica should be updated to the updated state. If the `Mod:handle_write` call raises a `throw` exception with value `Val`, then this function should return `{'ABORTED', exception, Val}`.

The callback module should define the following callbacks:

- `init()` for computing the initial state, should return `{ok, State :: term()}`.
- `handle_read(Request :: term(), State :: term())` for handling the read request `Request` in the state `State`. Should return `{reply, Reply :: term()}` or return `stop` if the server should be stopped, returning `stop` should have the same effect as if `stop/1` had been called while the read request was running. VER. 1.3
- `handle_write(Request :: term(), State :: term())` for handling the write request `Request` in the state `State`. Should return one of the following three kinds of values:

- `{noupdate, Reply :: term()}` if the operation does not result in a updated state,
- `{updated, Reply :: term(), NewState :: term()}` if the operation results in the updated state `NewState`,
- `stop` if the server should be stopped, returning `stop` should have the same effect as if `stop/1` had been called while the write request was running. VER. 1.3

Document which properties your module provides (and under which assumptions). For instance, is it possible, or not, for readers to starve out a writer, or the other way around? Remember to detail in your report how you have tested these properties. In general, as always, remember to test your solution and include your test in the hand-in.

Question 4: AlzheimerDB

This question is about making a simple in-memory database called AlzheimerDB. Because AlzheimerDB is in-memory only, it will forget everything if it is turned off, hence the name[4].

You can implement this database using the `gen_replicated` module from Question 3, but you do not have to. Document and explain your implementation strategy.

Implement a module `alzheimer` with the following API (if you haven't implemented `gen_replicated` then you may use `gen_server` instead, to demonstrate that you know how to implement a behaviour). In the following `Aid` is a process ID for an AlzheimerDB server:

- `start()` for starting a new AlzheimerDB server. Returns `{ok, Aid}`.
- `query(Aid, P)` for applying the function `P` to each row in the database. Calls `P({Id, Data})` for each row in the database, where `Data` is the row data for `Id`. Returns `{ok, Rows}` where `Rows` is a list of all the rows for which `P` returned `true`; or returns `{error, Row}` if `P` raised an exception for `Row`. VER. 1.3

  As query operations are read-only it is possible to efficiently run several concurrently. Explain what you have done to take advantage of this, and how you have tested it. In particular, you should discuss advantages and limitations of your approach.
- `upsert(Aid, Id, F)` for inserting or updating the row with identifier `Id`. If there is no row with identifier `Id` in the database, then `F` is called with the argument `{new, Id}`; otherwise `F` is called with the argument `{existing, {Id, Data}}`, where `Data` is the row data for `Id`. If `F` returns `{modify, NewData}`, then the row for `Id` is set to `NewData`, otherwise if `F` returns `ignore` then the database is not updated. Returns the return value of the `F` call, and if `F` raises a normal (`throw`) exception, then this function should raise the same exception.

Demonstrate that your solution works by implementing a sample program that uses your `alzheimer` module.

[4] Apologies to the Mnesia developers for reusing their pun.

Appendix A: Example SubScript programs

Appendix A.1: Source code for `intro.js`

The examples from the introduction for SubScript.

```
var xs = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
var squares = [ for (x of xs) x * x ];
var evens = [ for (x of xs) if (x % 2 === 0) x ];
var many_a = [ for (x of xs) for (y of xs) 'a' ];
var hundred = [ for (i of [0])
                for (x of xs)
                for (y of xs) i = i + 1 ];
```

Appendix A.2: 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])))))
]
```

Appendix A.3: Source code for `scope.js`

Simple program that demonstrates that variables bound in array comprehensions can shadow those declared before, in this example the variable `x`, and that those binding are restored afterwords. Thus, in this example we end up with both variables `x` and `z` bound to the value 42, and the variable `y` bound to an array with three elements, each a one-character string.

```
var x = 42;
var y = [for (x of 'abc') x];
var z = x;
```

Appendix A.4: Abstract syntax tree for `scope.js`

```
Prog [
    VarDecl "x" (Just (Number 42)),
    VarDecl "y" (Just (Compr ("x", String "abc", Nothing) (Var "x"))),
    VarDecl "z" (Just (Var "x"))
  ]
```