Assignment 1 (main part): A Boa Interpreter

Version 1.0

Due: Wednesday, September 16 at 20:00

The objective of this assignment is to gain practical experience with using monads to structure simple interpreters. Please read through the *entire* assignment text before you start working on it.

Note This assignment also includes a collection of simple warm-up exercises, as described on Absalon. For those, you are *only* asked to submit your working code, but not a separate design/implementation document, assessment, or evidence of testing. If you want to communicate anything extra about your solutions to the warm-up exercises, place your remarks as comments in the source code.

1 The Boa language

The language Boa is a tiny subset of Python 3, with the intent that valid Boa programs should normally give the same result as when run in Python. In this assignment, we will only work with the abstract syntax of Boa; next week, we will consider concrete syntax and parsing.

Boa works with a type of structured *values*, given by the following algebraic datatype:

```
data Value =
    NoneVal
  | TrueVal | FalseVal
  | IntVal Int
  | StringVal String
  | ListVal [Value]
```

That is, a Boa value is one of the three special atoms `None`, `True`, or `False`, an integer, a character string, or a (possibly empty) list of values.

A Boa expression has one of the following forms:

```
data Exp =
    Const Value
  | Var VName
  | Oper Op Exp Exp
  | Not Exp
  | Call FName [Exp]
  | List [Exp]
  | Compr Exp [CClause]

type VName = String
type FName = String
```

```
data Op = Plus | Minus | Times | Div | Mod | Eq | Less | Greater | In

data CClause =
    CCFor VName Exp
  | CCIf Exp
```

The intended semantics of expressions should be unsurprising, with the following notes:

- `Const` v evaluates to the value v.
- `Var` x evaluates to the value currently bound to the variable x. If x has no current binding, its evaluation signals an error.
- `Oper` o e_1 e_2 applies the operator o to the values of e_1 and e_2 (which must be evaluated in that order). The meanings of the arithmetic operators (`Plus`, ..., `Mod`) are the obvious ones, with both arguments required to be integers (otherwise an error is signaled). An attempted division or modulo by 0 also signals an error.

  The comparison operators (`Eq`, `Less`, `Greater`) return (the Boa atoms) `True` or `False`, depending on whether the corresponding relation holds between the arguments or not. `Eq` can be used to compare arbitrary values for structural equality, while the other two can only compare integers. Finally, the operator `In` requires its second argument value to be a list, and checks whether the first one occurs (as determined by an `Eq`-test) anywhere in that list.
- `Not e` returns the logical negation of the value of its argument. For this purpose, the atoms `None` and `False`, the integer 0, the empty string, and the empty list are considered to represent falsehood (and so their negation returns `True`), while all other values are considered to represent truth (and so return `False` when negated).
- `Call` f `[`e_1`,...,`e_n`]` (where $n \geq 0$) evaluates the argument tuple $(e_1, ..., e_n)$ (from left to right) and then calls the built-in function f on the values. Currently, there are only two such functions in Boa:
  - `range` may be called on between 1 and 3 integer arguments. The most general form, with arguments (n_1, n_2, n_3), generates the list of integers from n_1 to (but not including) n_2, stepping by n_3. For example, taking $n_1 = 3$, $n_2 = 10$ (or 11), and $n_3 = 2$ would generate the list $[3, 5, 7, 9]$. n_3 may be positive or negative, but not 0. If $n_1 \geq n_2$ (when $n_3 > 0$) or $n_1 \leq n_2$ (when $n_3 < 0$), the resulting list is empty.

    If only two arguments are supplied, they are used for n_1 and n_2, with n_3 taken to be 1; and a single argument is used as the value of n_2, with $n_1 = 0$ and $n_3 = 1$.
  - `print` may be called on any number of arguments, of any types. It prints all the arguments on a single line, separated by single spaces. Simple values (atoms and integers) are printed in the natural way. Strings are printed directly, without any outer quotes.[1] Lists are printed between "`[`" and "`]`", with the elements separated by "`, `" (comma and space). The result of the call to `print` is always just the special atom `None`.

[1] Note that, in Python, when a string is not an *immediate* argument to `print`, but occurs deeper within a list or other data structure being printed, it is formatted in a style that can be easily parsed back in. Specifically, such strings are surrounded by single quotes ("`'`"), and any quote, backslash, or newline characters in the string are rendered as the two-character sequences "`\'`", "`\\`", and "`\n`", respectively. You are allowed, but *not* required, to implement this refined behavior in Boa.

For example, a call to `print` where the arguments evaluate to:

```
[IntVal 42, StringVal "foo", ListVal [TrueVal, ListVal []], IntVal (-1)]
```

should result in the following line of output (with no leading or trailing spaces):

```
42 foo [True, []] -1
```

Attempting to call any other function than the above two signals an error.

- `List` [e_1,...,e_n] (where $n \geq 0$) simply evaluates the expressions $e_1, ..., e_n$ (from left to right) to values, and packages those up as a single list-value.
- `Compr` e_0 [cc_1,...,cc_n] (where $n \geq 0$) is a *list comprehension*, essentially analogous to Haskell's [e_0 | q_1, ..., q_n]. Here, a Boa `for`-clause `CCFor` x e corresponds to a Haskell generator `x <-` e. (If e does not evaluate to a list value, an error is signaled.) The binding for x may shadow any previous bindings for x, and is visible in all the following clauses, as well as in the body expression e_0.

  The Boa `if`-clause `CCIf` e corresponds to a boolean guard in Haskell, where the value of e is interpreted as a truth value in the same way as for `Not`-expressions above, so that, e.g., `CCIf (List [])` would be considered a failing guard, while `CCIf (ConstVal (IntVal 7))` would succeed.

  Note that, in the concrete syntax of Boa, we will later require that $n \geq 1$, and that cc_1 is a `CCFor`-clause (not `CCIf`). However, the semantics of the Boa abstract syntax, like Haskell, imposes no such restrictions.

  Also, remember that all Boa expressions (including, but not limited to, e_0) may print output as a side effect of evaluation.

Finally, a Boa program consists of a sequence of *statements*:

```
type Program = [Stmt]

data Stmt =
    SDef VName Exp
  | SExp Exp
  deriving (Eq, Show, Read)
```

A definition statement `SDef` x e evaluates e to a value, and binds x to that value for the remaining statements in the sequence. An expression statement `SExp` e just evaluates e and discards the result value. (But any printing and/or errors arising from evaluating e still take effect.)

Whenever an error is signaled, evaluation or execution stops immediately with an error message, and all current variable bindings are discarded. However, any lines of output successfully generated before the error occurred are still printed.

An example of a larger Boa program, in both concrete and abstract syntax, as well as the expected output, can be found in appendix A.

2 A Boa interpreter

The above definitions (with the usual `deriving`-clauses for the datatype declarations) can be found in module `BoaAST`. The interpreter itself lives in the module `BoaInterp`, and introduces the following types:

```
type Env = [(VName, Value)]

data RunError = EBadVar VName | EBadFun FName | EBadArg String
  deriving (Eq, Show)
```

We have chosen to represent environments as simple *association lists* (i.e., lists of the form `[(`x_1`,`v_1`),...,(`x_n`,`v_n`)]`, where the *first* binding (if any) for a variable x in the list is taken as the current one.

We distinguish between three kinds of runtime errors: `EBadVar` x represents an attempt to access the unbound variable x; and correspondingly, `EBadFun` f means that we attempted to call the undefined function f. All other possible errors have to do with passing an invalid argument or argument list to a Boa construct or function, and are represented as `EBadArg` s, where s is some informative, human-readable error message. All errors in interpreted Boa programs should be reported through `RunError`, and never by calling the Haskell `error` function. (Actual errors in the *interpreter itself*, such as unhandled Boa features, or somehow reaching "impossible" cases, may – if relevant – still use `error`.)

The main type constructor for Boa computations is then the following:

```
newtype Comp a = Comp {runComp :: Env -> (Either RunError a, [String])}
```

That is, computations have (read-only) access to the environment, and return either a runtime error or a result of the expected type, together with (in either case) a possibly empty list of output lines produced by the computation.

Make `Comp` an instance of `Monad` (and `Functor` and `Applicative` as well, but you can use the usual boilerplate code for that), and define the following associated operations:

```
abort :: RunError -> Comp a
look :: VName -> Comp Value
withBinding :: VName -> Value -> Comp a -> Comp a
output :: String -> Comp ()
```

Here, `abort` *re* is used for signaling the runtime error *re*. `look` x returns the current binding of the variable x (or signals an `EBadVar` x error if x is unbound). Conversely, the operation `withBinding` x v m runs the computation m with x bound to v, in addition to any other current bindings. Finally, `output` s appends the line s to the output list. s should *not* include a trailing newline character (unless the line arises from printing a string value that itself contains an embedded newline).

The rest of your code (except where explicitly indicated below) should *not* depend on the exact definition of the type `Comp a`; that is, it should neither directly use the value constructor `Comp` nor the projection `runComp`, but always go through one of the above functions.

Next, define the following helper functions:

```
truthy :: Value -> Bool
operate :: Op -> Value -> Value -> Either String Value
apply :: FName -> [Value] -> Comp Value
```

`truthy` v simply determines whether the value v represents truth or falsehood, as previously specified. `operate` o v_1 v_2 applies the operator o to the arguments v_1 and v_2, returning either the resulting value, or an error message if one or both arguments are inappropriate for the operation. Similarly, `apply` f `[`v_1`,...,`v_n`]` applies the built-in function f to the (already evaluated) argument tuple v_1, ..., v_n, possibly signaling an error if f is not a valid function name (`EBadFun`), or if the arguments are not valid for the function (`EBadArg`).

Finally, define the main interpreter functions,

```
eval :: Exp -> Comp Value
exec :: Program -> Comp ()
execute :: Program -> ([String], Maybe RunError)
```

`eval` e is the computation that evaluates the expression e in the current environment and returns its value. Likewise, `exec` p is the computation arising from executing the program (or program fragment) p, with no nominal return value, but with any side effects in p still taking place in the computation. Finally, `execute` p explicitly returns the list of output lines, and the error message (if relevant) resulting from executing p in the initial environment, which contains no variable bindings. (For implementing `execute` (only), you are allowed to use the `runComp` projection of the monad type.)

Hint: when implementing `eval` for comprehensions, start by covering the cases with *exactly one* clause in the list, then try to incrementally generalize your code to other clause lists.

Driver program

For your convenience, we have provided a simple stand-alone wrapper for your interpreter: `stack build` will create the executable `boa`, which you can then run as `stack run boa -- -i` *`program`*`.ast`, where *`program`*`.ast` is the abstract syntax tree of the program. (Some examples are provided in the directory `examples/`). In particular, this will show you the printed output of the interpreted program in the form a Boa user would see it.

3 What to hand in

3.1 Code

Form To facilitate both human and automated feedback, it is very important that you closely follow the code-packaging instructions in this section. We provide skeleton/stub files for all the requested functionality in both the warm-up and the main part. These stub files are packaged in the handed-out `code.zip`. It contains a single directory `code/`, with a couple of subdirectories organized as Stack projects. You should edit the provided stub files as directed, and leave everything else unchanged.

It is crucial that you not change the provided types of any exported functions, as this will make your code incompatible with our testing framework. Also, *do not* remove the bindings for any functions you do not implement; just leave them as `undefined`.

When submitting the assignment, package your code up again as a single `code.zip` (*not* `.rar`, `.tar.gz`, or similar), with exactly the same structure as the original one. When rebuilding `code.zip`, please take care to include only files that constitute your actual submission: your source code and supporting files (build configuration, tests, etc.), but *not* obsolete/experimental versions, backups, editor autosave files, revision-control metadata, `.stack-work` directories, and the like. If your final `code.zip` is *substantially* larger than the handed-out version, you probably included something that you shouldn't have.

For the warm-up part, just put your function definitions in `code/part1/src/Warmup.hs`, where indicated.

For the main part, your code must be placed in the file `code/part2/src/BoaInterp.hs`. It should only export the requested functionality. Any tests or examples should be put in a separate module under `code/part2/tests/`. For inspiration, we have provided a very minimalistic (and far from adequate) test suite in `code/part2/tests/Test.hs`. If you are using Stack (and why wouldn't you be?), you can build and run the suite by `stack test` from the directory `code/part2/`.

The definitions for this assignment (e.g., type `Exp`) are available in file `.../src/BoaAST.hs`. You should only import this module, and not directly copy its contents into `BoaInterp`. And of course, *do not* modify anything in `BoaAST`.

Content As always, your code should be appropriately commented. In particular, try to give brief informal specifications for any auxiliary "helper" functions you define, whether locally or globally. On the other hand, avoid trivial comments that just rephrase in English what the code is already saying in Haskell. Try to use a consistent indentation style, and avoid lines of over 80 characters.

You may (but shouldn't need to, for this assignment) import additional functionality from (only) the core GHC libraries: your solution code should compile with a `stack build` issued from the directory `code/part`*n*`/`, using the provided `package.yaml`. (For your *testing*, you may use additional relevant packages from the course-mandated version of the Stack LTS distribution. We recommend `Tasty`.)

In your test suite, remember to also include any relevant *negative* test cases, i.e., tests verifying that your code correctly detects and reports error conditions. Also, if some functionality is known to be missing or wrong in your code, the corresponding test cases should still compare against the *correct* expected output for the given input (i.e., the test should *fail*), not against whatever incorrect result your code currently returns.

Your code should ideally give no warnings when compiled with `ghc(i) -W`; otherwise, add a comment explaining why any such warning is harmless or irrelevant in each particular instance. If some problem in your code prevents the whole file from compiling at all, be sure to comment out the offending part before submitting, or all the automated tests will fail.

3.2 Report

In addition to the code, you must submit a short (normally 2–3 pages) report, covering the following two points, for the main (not warm-up) part only:

- Document any (relevant) *design* and *implementation* choices you made. This includes, but is not limited to, answering any questions explicitly asked in the assignment text. Focus on high-level aspects and ideas, and explain *why* you did something non-obvious, not only *what* you did. It is rarely appropriate to do a detailed function-by-function code walk-through in the report; technical remarks about how the functions work belong in the code as comments.

- Give an honest, justified *assessment* of the quality of your submitted code, and the degree to which it fulfills the requirements of the assignment (to the best of your understanding and knowledge), as well as – where relevant – general issues of efficiency, robustness, maintainability, etc. Be sure to clearly explain any known or suspected deficiencies.

  It is very important that you document on what your assessment is based (e.g., wishful thinking, scattered examples, systematic tests, correctness proofs?). Include any automated tests you wrote with your source submission, make it clear how to run them, and *summarize* the results in the report. If there were some aspects or properties of your code that you couldn't easily test in an automated way, explain why.

Your report submission should be a single PDF file named `report.pdf`, uploaded along with (not inside!) `code.zip`. The report should include a listing of your code and tests (but not the already provided auxiliary files) as an appendix.

3.3 General

Detailed upload instructions, in particular regarding the logistics of group submissions, can be found on the Absalon submission page.

We also *expect* to provide an automated system to give you preliminary feedback on your planned code submission, including matters of form, style, correctness, etc. You are **strongly advised** to take advantage of this opportunity to validate your submission, and – if necessary – fix or otherwise address (e.g., by documenting as known flaws) any legitimate problems it uncovers.

Note, however, that passing the automated tests is *not* a substitute for doing *and documenting* your own testing. Your assessment must be able to stand alone, without leaning on the output from our tool.

A A sample Boa program

Here is a larger example of a Boa program, in concrete syntax for readability (available in the handout as file `code/part2/examples/misc.boa`):

```
squares = [x*x for x in range(10)];
print([123, [squares, print(321)]]);
print('Odd squares:', [x for x in squares if x % 2 == 1]);
n = 5;
composites = [j for i in range(2, n) for j in range(i*2, n*n, i)];
print('Printing all primes below', n*n);
[print(x) for x in range(2,n*n) if x not in composites]
```

And here is the corresponding abstract syntax, as a Haskell value of type `Program` (file `.../misc.ast`):

```
[SDef "squares"
      (Compr (Oper Times (Var "x") (Var "x"))
             [CCFor "x" (Call "range" [Const (IntVal 10)])]),
 SExp (Call "print" [List [Const (IntVal 123),
                           List [Var "squares",
                                 Call "print" [Const (IntVal 321)]]]]),
 SExp (Call "print" [Const (StringVal "Odd squares:"),
                     Compr (Var "x") [CCFor "x" (Var "squares"),
                                      CCIf (Oper Eq (Oper Mod (Var "x")
                                                              (Const (IntVal 2)))
                                                    (Const (IntVal 1)))]]),
 SDef "n" (Const (IntVal 5)),
 SDef "composites"
      (Compr (Var "j") [CCFor "i" (Call "range" [Const (IntVal 2),Var "n"]),
                        CCFor "j" (Call "range" [Oper Times (Var "i")
                                                            (Const (IntVal 2)),
                                                 Oper Times (Var "n") (Var "n"),
                                                 Var "i"])]),
 SExp (Call "print" [Const (StringVal "Printing all primes below"),
                     Oper Times (Var "n") (Var "n")]),
 SExp (Compr (Call "print" [Var "x"])
             [CCFor "x" (Call "range" [Const (IntVal 2),
                                       Oper Times (Var "n") (Var "n")]),
              CCIf (Not (Oper In (Var "x") (Var "composites")))])]
```

When run, this program should print the following lines, all without leading or trailing spaces (file `.../misc.out`):

```
321
[123, [[0, 1, 4, 9, 16, 25, 36, 49, 64, 81], None]]
Odd squares: [1, 9, 25, 49, 81]
Printing all primes below 25
2
3
5
7
11
13
17
19
23
```