AP assignment 1: Boa Interpreter

sortraev, wlc376

2020-09-16

0 Design and Implementation

In this section, I present and explain key features of my implementation: `operate`, `apply`, and `eval`. The rest of the implementation is largely trivial, but can be viewed in appendix A.

The handed-in code in `code/part2/src` is supplied with simple code comments where appropriate.

0.1 Implementing `operate`

Implementation of `operate` is quite straightforward, and resembles that of `evalErr` from last weeks assignment.

Even cases such as `operate Eq` and `operate In` are trivial, since `Value` derives the `Eq` type class:

```
operate Eq a b = Right $ truthy' $ a == b              -- since Value implements Eq.
...
operate In a (ListVal xs) = Right $ truthy' $ elem a xs -- again, simple solution since Value implements Eq.
```

0.1.1 `operate` error handling

`operate` can produce two types of errors: zero divisor errors in division and modulo operations, and type errors when attempting arithmetic or less/greater than-comparison using non-integers; and when using `In` with a non-list as second argument. These errors are handled as such:

```
operate Div   (IntVal a) (IntVal b) = if b /= 0 then Right $ IntVal $ div a b
                                               else Left "Div error: zero divisor"
operate Mod   (IntVal a) (IntVal b) = if b /= 0 then Right $ IntVal $ mod a b
                                               else Left "Mod error: zero divisor"
...
operate op _ _       = Left $ "Type mismatch for " ++ show op
```

0.2 Implementing `apply`

For now, the implementation should only handle the functions `print` and `range`. As such, I let `apply` return an `abort (EBadFun fname)` for any other `fname`.

0.2.1 `apply "print"`

The `print` case uses the `output` operation for the `Comp` monad, which appends a string to the current output state of the program. I map each argument in the argument list to a print string using `showVal`, and concatenate this list of strings with the built-in `unwords` such that eg `print(True, None, 3+4)` produces the array `["True None 7"]` rather than `["True", "None", "7"]`.

```
apply "print" args = output (unwords $ map showVal args) -- unwords necessary in order to format correctly.
                       >> return NoneVal
  where showVal NoneVal = "None"
        ...
```

The `print` function produces no errors.

0.2.2 `apply "range"`

The `range` case is a little more tricky. Let's first see the entire piece of code before breaking it down. Below is a snippet of the `apply "range"` case:

```
1  apply "range" args = -- apply "range" is a mess, but it works.
2    case args of
3      [IntVal hi]                          -> boaRange 0  hi 1
4      [IntVal lo, IntVal hi]               -> boaRange lo hi 1
5      [IntVal lo, IntVal hi, IntVal step] -> boaRange lo hi step
6      args -> abort $ EBadArg $ argError $ length args
7
8      where argError num_args = "range expects " ++
9              if elem num_args [1..3] then "integer arguments only"
10             else "1-3 arguments, got " ++ show num_args
11           boaRange lo hi step =
12             if step == 0 then abort (EBadArg "range arg 3 must not be zero")
13             else return $ ListVal $ map IntVal $
14               [lo, lo+step..hi + (if step < 0 then 1 else -1)]
```

Building a Pythonic range

Python and Haskell ranges are exclusive and inclusive, respectively. This is accounted for by adding or subtracting one to the `hi` argument, depending on whether the `step` is negative or postivie, respectively. A range with a

`apply "range"` error handling

Wrt. number of arguments, I choose to not distinguish between zero and more than 3 arguments, showing an error with the message `range expects 1-3 arguments, got <n>`, where `<n>` is the number of arguments given (line 10).

A generic type error is thrown if the list contains the right number of arguments but did not match either of the three cases discussed in the previous paragraph (line 9).

If three integer arguments are given but the third is zero, an `abort EBadArg` is returned with an error message resembling that of Python's `range()` (line 12).

0.3 Implementing `eval`

As far as `eval` goes, the most interesting case is, of course, comprehensions. All other cases are largely trivial or self-explanatory.

0.3.1 `buildCompr` helper function

I tried to build the `Compr` case directly into `eval` but I could simply not get correct results with more than one list generator in the comprehension; for example, the comprehension:

```
[x + y | x <- [1, 2], y <- [3, 4]]
```

would evaluate to `[[4, 5], [5, 6]]`, and not the correct `[4, 5, 5, 6]`. I tried everything in my Haskell capabilities to concatenate the return values of the `mapM`, but found a solution in using a helper function with the return type `Comp [Value]`, whose result could be `fmap`'ed with `ListVal` to create the needed `Comp Value` (line 12 below). This helper function is seen in the below snippet:

```
1  buildCompr :: Comp Value -> [CClause] -> Comp [Value]
2  buildCompr out [] = (:[]) <$> out
3  buildCompr out (CCFor v e:cs) = eval e >>= \e_val ->
4    case e_val of   -- bind xs to v; evaluate rest of clauses; concat results.
5      ListVal xs -> concatMapM (\x -> withBinding v x (buildCompr out cs)) xs
6      _          -> abort $ EBadArg "Compr error: generator expects list expression"
7    where concatMapM f xs = concat <$> mapM f xs
8
9  buildCompr out (CCIf e:cs) = eval e >>= \cond ->
10   if truthy cond then buildCompr out cs else return [] -- stop generating!
11 ...
12 eval (Compr e cs) = Listval <$> buildCompr (eval e) cs
```

The only types of errors here are non-list values in generator expressions (line 6).

1 Testing

In this section, I discuss my validation test plan and report the results of testing.

1.1 Reproduction of tests

I use `Tasty` for testing. All of my tests can be viewed in `code/part2/tests`. To reproduce tests, navigate to `code/part2` and run `stack test`.

1.2 Test plan and execution

The goal of the test plan is to attain full edge case coverage with unit testing of each implemented function and helper function. I also want to test each possible type of error in the program.

After unit testing, I want to test the entire interpreter with a number of hard-coded syntax trees, and for example assert correct variable overshadowing, printing, and error propagation in a larger program with multiple errors.

For the arithmetic operators in `operate`, I would also like to test some simple, well-known identities (eg. $a \cdot 0 = 0 \cdot a = 0$ for all a).

However, due to deadline constraints, I have resorted to full code coverage of each case of `operate` and `apply`, but only the comprehension case of `eval`. In addition, I construct a number of ASTs in `part2/tests/progs` with which to test correct output.

I write a total of 70 test cases, which can be inspected in the print-out of `stack test` or in appendix B.

1.3 Validation testing results

All of my validation tests (as well as all the supplied onlineTA tests) are successful.

1.4 Evaluation

As explained, I unfortunately did not attain full coverage of the code or the possible edge cases. However, I feel satisfiably confident that my implementation is correct.

Appendix

A Code: Implementation

A.1 code/part2/src/BoaInterp.hs

```
-- Skeleton file for Boa Interpreter. Edit only definitions with 'undefined'

module BoaInterp
  (Env, RunError(..), Comp(..),
   abort, look, withBinding, output,
   truthy, operate, apply,
   eval, exec, execute)
  where

import BoaAST
import Control.Monad
import Data.List

type Env = [(VName, Value)]

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

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

instance Monad Comp where
  return a = Comp (\_ -> (return a, mempty))
  m >>= f =
    Comp (\env -> case runComp m env of
                    (Left re, s) -> (Left re, s)
                    (Right a, s) -> (a', mappend s s')
                      where (a', s') = runComp (f a) env)

-- You shouldn't need to modify these
instance Functor Comp where
  fmap = liftM
instance Applicative Comp where
  pure = return
  (<*>) = ap

-- Operations of the monad
abort :: RunError -> Comp a
abort err = Comp (\_ -> (Left err, mempty))


look :: VName -> Comp Value
look v = Comp $ \env -> let res = maybe (Left $ EBadVar v) Right (find v env)
                        in (res, mempty)
  where find :: VName -> Env -> Maybe Value
        find v ((w, val):xs) = if v == w then Just val else find v xs
        find _ [] = Nothing


withBinding :: VName -> Value -> Comp a -> Comp a
```

```haskell
withBinding v val m = Comp $ \env -> runComp m ((v, val):env)


output :: String -> Comp ()
output s = Comp $ \_ -> (Right (), [s])


truthy :: Value -> Bool
truthy NoneVal  = False
truthy FalseVal = False
truthy TrueVal  = True
truthy (IntVal n)    = n /= 0
truthy (StringVal s) = not $ null s
truthy (ListVal xs)  = not $ null xs


-- the reverse of truthy :)
truthy' :: Bool -> Value
truthy' True  = TrueVal
truthy' False = FalseVal


operate :: Op -> Value -> Value -> Either String Value
operate Plus  (IntVal a) (IntVal b) = Right $ IntVal $ a + b
operate Minus (IntVal a) (IntVal b) = Right $ IntVal $ a - b
operate Times (IntVal a) (IntVal b) = Right $ IntVal $ a * b

operate Div   (IntVal a) (IntVal b) = if b /= 0 then Right $ IntVal $ div a b
                                                else Left "Div error: zero divisor"
operate Mod   (IntVal a) (IntVal b) = if b /= 0 then Right $ IntVal $ mod a b
                                                else Left "Mod error: zero divisor"

operate Eq a b = Right $ truthy' $ a == b               -- a single case, since Value implements Eq.
operate Less    (IntVal a) (IntVal b) = Right $ truthy' $ a < b
operate Greater (IntVal a) (IntVal b) = Right $ truthy' $ a > b

operate In a (ListVal xs) = Right $ truthy' $ elem a xs -- again, simple solution since Value implements Eq.

operate op _ _      = Left $ "Type mismatch for " ++ show op


apply :: FName -> [Value] -> Comp Value
apply "print" args = output (unwords $ map showVal args) -- unwords necessary in order to format correctly.
                       >> return NoneVal
  where showVal NoneVal  = "None"
        showVal TrueVal  = "True"
        showVal FalseVal = "False"
        showVal (IntVal n) = show n
        showVal (StringVal s) = s
        showVal (ListVal xs) = "[" ++ intercalate ", " (map showVal xs) ++ "]"

apply "range" args = -- apply "range" is a mess, but it works.
  case args of
    [IntVal hi]                         -> boaRange 0  hi 1
    [IntVal lo, IntVal hi]              -> boaRange lo hi 1
    [IntVal lo, IntVal hi, IntVal step] -> boaRange lo hi step
    args -> abort $ EBadArg $ argError $ length args

    where argError num_args = "range expects " ++
            if elem num_args [1..3] then "integer arguments only"
            else "1-3 arguments, got " ++ show num_args
          boaRange lo hi step =
            if step == 0 then abort $ EBadArg "range arg 3 must not be zero"
            else return $ ListVal $ map IntVal $
              [lo, lo+step..hi + (if step < 0 then 1 else -1)]
```

```haskell
apply fname _ = abort $ EBadFun fname


buildCompr :: Comp Value -> [CClause] -> Comp [Value]
buildCompr out [] = (:[]) <$> out
buildCompr out (CCFor v e:cs) = eval e >>= \e_val ->
  case e_val of   -- bind xs to v; evaluate rest of clauses; concat results
    ListVal xs -> concatMapM (\x -> withBinding v x (buildCompr out cs)) xs
    _          -> abort $ EBadArg "Compr error: generator expects list expression"
  where concatMapM f xs = concat <$> mapM f xs

buildCompr out (CCIf e:cs) = eval e >>= \cond ->
  if truthy cond then buildCompr out cs else return [] -- stop generating!


eval :: Exp -> Comp Value
eval (Const val) = return val
eval (Var v)     = look v

eval (Oper op e1 e2) = eval e1 >>= \e1_val -> eval e2 >>= \e2_val ->
  either (abort . EBadArg) return (operate op e1_val e2_val)

eval (Not e)       = truthy' . not . truthy <$> eval e

eval (Call f args) = mapM eval args >>= apply f

eval (List es)    = ListVal <$> mapM eval es
eval (Compr e cs) = ListVal <$> buildCompr (eval e) cs


exec :: Program -> Comp ()
exec (SDef v e:stms) = eval e >>= \e_val -> withBinding v e_val $ exec stms
exec (SExp e:stms)   = eval e >> exec stms
exec []              = return ()


execute :: Program -> ([String], Maybe RunError)
execute program = let (res, output) = runComp (exec program) []
                      status        = either Just (\_ -> Nothing) res
                  in (output, status)
```

B Code: Testing

B.1 code/part2/tests/Test.hs

```haskell
import Test.Tasty
import Test.Tasty.HUnit

import OperateTests
import ApplyTests
import EvalTests
import AstTests

tests :: TestTree
tests = testGroup "All tests"
  [
   operateTests
  ,applyTests
  ,evalTests
  ,astTests
  ]

main :: IO ()
main = defaultMain $ localOption (mkTimeout 1000000) tests
```

B.2 code/part2/tests/OperateTests.hs

```
module OperateTests where

import Test.Tasty
import Test.Tasty.HUnit

import BoaAST
import BoaInterp
------------------------
----- operate tests -----
------------------------
a' = 42
b' = -1337

a = IntVal a'
b = IntVal b'
zero = IntVal 0
one = IntVal 1

plus1 = operate Plus a b
plus2 = operate Plus a zero
plus3 = operate Plus zero a

minus1 = operate Minus a b
minus2 = operate Minus a zero
minus3 = operate Minus zero a

times1 = operate Times a b
times2 = operate Times a zero
times3 = operate Times zero b
times4 = operate Times b one
times5 = operate Times one a

div1 = operate Div a b
div2 = operate Div a zero
div3 = operate Div zero b
div4 = operate Div b one
div5 = operate Div one a
div6 = operate Div one b
div7 = operate Div one one

mod1 = operate Mod a b
mod2 = operate Mod a zero
mod3 = operate Mod zero b
mod4 = operate Mod b one
mod5 = operate Mod one b

less1 = operate Less a b
less2 = operate Less b a
less3 = operate Less a a
greater1 = operate Greater a b
greater2 = operate Greater b a
greater3 = operate Greater a a

eq1 = operate Eq a a
eq2 = operate Eq a b
eq3 = operate Eq TrueVal FalseVal
eq4 = operate Eq NoneVal (ListVal [IntVal 3, StringVal "hej"])
eq5 = operate Eq (StringVal "abba") (StringVal "abbc")
eq6 = operate Eq (StringVal "123") (StringVal "123")

```

```
fooList1 = [TrueVal, IntVal 3, StringVal "hey", NoneVal, StringVal "hej", IntVal 4]
fooList2 = [TrueVal, IntVal 3, StringVal "hey!", NoneVal, StringVal "hej", IntVal 4]
fooList3 = (replicate 10000 (ListVal fooList1)) ++ [ListVal fooList2] ++
           (replicate 10000 (ListVal fooList1))

in1 = operate In (IntVal 3) (ListVal [])
in2 = operate In (StringVal "hej") (ListVal fooList1)
in3 = operate In TrueVal (ListVal [FalseVal, IntVal 3])
in4 = operate In (IntVal 1728329) (ListVal $ map IntVal [1..1728329*2])
in5 = operate In TrueVal (ListVal $ map IntVal [1..1728329*2])
in6 = operate In (ListVal fooList2) (ListVal fooList3)
in7 = operate In (ListVal fooList3) (ListVal fooList3)


operateTests :: TestTree
operateTests = testGroup ("operate tests - with (a, b) := (" ++
                          show a' ++ ", " ++ show b' ++ ")")
  [testCase "Plus1: a + b"       $ plus1  @?= (Right $ IntVal $ a' + b')
  ,testCase "Plus2: a + 0"       $ plus2  @?= (Right a)
  ,testCase "Plus3: 0 + a"       $ plus3  @?= (Right a)

  ,testCase "Minus1: a - 0"      $ minus2  @?= (Right a)
  ,testCase "Minus2: 0 - a"      $ minus3  @?= (Right $ IntVal (-a'))

  ,testCase "Minus3: a - b"      $ minus1  @?= (Right $ IntVal $ a' - b')
  ,testCase "Minus4: a - 0"      $ minus2  @?= (Right a)
  ,testCase "Minus5: 0 - a"      $ minus3  @?= (Right $ IntVal (-a'))

  ,testCase "Times1: a * b"      $ times1  @?= (Right $ IntVal $ a' * b')
  ,testCase "Times2: a * 0 == 0" $ times2  @?= (Right zero)
  ,testCase "Times3: 0 * b == 0" $ times3  @?= (Right zero)
  ,testCase "Times4: b * 1 == b" $ times4  @?= (Right b)
  ,testCase "Times5: 1 * a == a" $ times5  @?= (Right a)

  ,testCase "Div1: a / b"                 $ div1  @?= (Right $ IntVal $ div a' b')
  ,testCase "Div2: zero div error"        $ div2  @?= (Left "Div error: zero divisor")
  ,testCase "Div3: 0 / b == 0"            $ div3  @?= (Right zero)
  ,testCase "Div4: b / 1 == b"            $ div4  @?= (Right b)
  ,testCase "Div5: 1 / a == 0 for a > 1"  $ div5  @?= (Right zero)
  ,testCase "Div6: 1 / b == -1 for b < 0" $ div6  @?= (Right $ IntVal (-1))
  ,testCase "Div7: 1 / 1 == 1"            $ div7  @?= (Right one)

  ,testCase "Mod1: a % b"                 $ mod1  @?= (Right $ IntVal $ mod a' b')
  ,testCase "Mod2: zero div error"        $ mod2  @?= (Left "Mod error: zero divisor")
  ,testCase "Mod3: 0 % a == 0"            $ mod3  @?= (Right zero)
  ,testCase "Mod4: 1 % 1 == 0 for b < 0"  $ mod4  @?= (Right zero)
  ,testCase "Mod5: 1 % b == b+1 for b < 0" $ mod5  @?= (Right $ IntVal $ b' + 1)

  ,testCase "Less1: a < b"              $ less1 @?= (Right $ FalseVal)
  ,testCase "Less2: b < a"              $ less2 @?= (Right $ TrueVal)
  ,testCase "Less3: a < a"              $ less3 @?= (Right $ FalseVal)

  ,testCase "Greater1: a < b"           $ greater1 @?= (Right $ TrueVal)
  ,testCase "Greater2: b < a"           $ greater2 @?= (Right $ FalseVal)
  ,testCase "Greater3: a < a"           $ greater3 @?= (Right $ FalseVal)

  ,testCase "Eq1: 42 == 42"             $ eq1 @?= (Right $ TrueVal)
  ,testCase "Eq2: 42 == -1337"          $ eq2 @?= (Right $ FalseVal)
  ,testCase "Eq3: true == false"        $ eq3 @?= (Right $ FalseVal)
  ,testCase "Eq4: None == [3, \"hej\"]" $ eq4 @?= (Right $ FalseVal)
  ,testCase "Eq5: \"abba\" == \"abbc\"" $ eq5 @?= (Right $ FalseVal)
  ,testCase "Eq6: \"123\" == \"123\""   $ eq6 @?= (Right $ TrueVal)

  ,testCase "In1: empty list"                  $ in1  @?= (Right FalseVal)
```

```haskell
  ,testCase "In2: element in list"             $ in2  @?= (Right TrueVal)
  ,testCase "In3: not in list"                 $ in3  @?= (Right FalseVal)
  ,testCase "In4: element in large list"       $ in4  @?= (Right TrueVal)
  ,testCase "In5: element not in large list"   $ in5  @?= (Right FalseVal)
  ,testCase "In6: list in list of lists"       $ in6  @?= (Right TrueVal)
  ,testCase "In7: list of lists not in itself" $ in7  @?= (Right FalseVal)
  ,operateNegTests
  ]

operateTypeError1 = operate Greater FalseVal b
operateTypeError2 = operate In  (ListVal [b]) NoneVal
operateTypeError3 = operate Div (StringVal "hej") (IntVal 0)

operateNegTests :: TestTree
operateNegTests = testGroup "operate negative tests"
  [testCase "operate type mismatch test 1" $
    operateTypeError1 @?= (Left "Type mismatch for Greater")
  ,testCase "operate type mismatch test 2" $
    operateTypeError2 @?= (Left "Type mismatch for In")
  ,testCase "operate type mismatch test 3" $
    operateTypeError3 @?= (Left "Type mismatch for Div")
  ]
```

B.3 code/part2/tests/ApplyTests.hs

```haskell
module ApplyTests where

import Test.Tasty
import Test.Tasty.HUnit

import BoaAST
import BoaInterp

-----------------------
----- apply tests -----
-----------------------
--- print tests ---
emptyPrint = apply "print" []

printTests :: TestTree
printTests = testGroup "print tests"
  [testCase "print(), empty call" $ runComp emptyPrint [] @?= (Right NoneVal, [""])
  ]


--- range tests ---

rangeOneArg    = apply "range" [IntVal 219]
rangeTwoArgs   = apply "range" [IntVal (-427), IntVal 50]
rangeThreeArgs = apply "range" [IntVal 273, IntVal (-100), IntVal (-20)]
rangeEmpty     = apply "range" [IntVal 3, IntVal 4, IntVal (-1)]

rangeTests :: TestTree
rangeTests = testGroup "range() positive tests"
  [
   testCase "range: one arg"    $ runComp rangeOneArg [] @?=
     (Right (ListVal $ map IntVal [0..218]), [])
  ,testCase "range: two args"   $ runComp rangeTwoArgs [] @?=
     (Right (ListVal $ map IntVal [-427..49]), [])
  ,testCase "range: three args" $ runComp rangeThreeArgs [] @?=
     (Right (ListVal $ map IntVal [273, 273-20..(-100)]), [])
  ,testCase "range: hi > lo, step < 0" $ runComp rangeEmpty [] @?=
     (Right (ListVal []), [])
  ]

rangeZeroArgs = apply "range" []
rangeSeventeenArgs = apply "range" $ map IntVal [4..20]

rangeTypeErr1 = apply "range" [NoneVal]
rangeTypeErr2 = apply "range" [IntVal 1, TrueVal]
rangeTypeErr3 = apply "range" [IntVal 1, StringVal "hej", IntVal 3]

rangeNegTests :: TestTree
rangeNegTests = testGroup "range negative tests"
  [testCase "range: zero args" $ runComp rangeZeroArgs [] @?=
     (Left $ EBadArg "range expects 1-3 arguments, got 0", [])
  ,testCase "range: too many args" $ runComp rangeSeventeenArgs [] @?=
     (Left $ EBadArg "range expects 1-3 arguments, got 17", [])

  ,testCase "range: type error test 1" $ runComp rangeTypeErr1 [] @?=
     (Left $ EBadArg "range expects integer arguments only", [])
  ,testCase "range: type error test 2" $ runComp rangeTypeErr2 [] @?=
     (Left $ EBadArg "range expects integer arguments only", [])
  ,testCase "range: type error test 3" $ runComp rangeTypeErr3 [] @?=
     (Left $ EBadArg "range expects integer arguments only", [])
```

```
  ]


--- apply, unknown functions ---
unknownFun1 = apply "myFun" []
unknownFun2 = apply "main()" [IntVal 2, ListVal [StringVal "argv"]]

applyUnknownFunTests :: TestTree
applyUnknownFunTests = testGroup "Negative apply tests"
  [testCase "apply, unknown function 1" $ runComp unknownFun1 [] @?= (Left $ EBadFun "myFun", [])
  ,testCase "apply, unknown function 2" $ runComp unknownFun2 [] @?= (Left $ EBadFun "main()", [])
  ]

applyTests :: TestTree
applyTests = testGroup "apply tests"
  -- printTests ++ rangeTests ++ rangeNegTests ++ applyUnknownFunTests
  [
   printTests
  ,rangeTests
  ,rangeNegTests
  ,applyUnknownFunTests
  ]
```

B.4 tests/EvalTests.hs

```haskell
module EvalTests where

import Test.Tasty
import Test.Tasty.HUnit

import BoaAST
import BoaInterp

----------------------
----- eval tests -----
----------------------
range_x = Call "range" $ map (Const . IntVal) [-50, 51]
range_z = Call "range" $ map (Const . IntVal) [1, 501, 100]
from = Const $ IntVal 127
to   = Const $ IntVal $ -208
step = Const $ IntVal $ -24
range_y = Call "range" $ map (Const . IntVal) [127, -208, -24]

compr1 = Compr (Var "z") [CCFor "z" range_z, CCIf (Oper Greater (Var "z") (Const $ IntVal 500))]
compr1_expected = ListVal []

compr2 = Compr (Var "z") [CCIf (Const TrueVal), CCFor "z" range_z]
compr2_expected = ListVal $ map IntVal [1, 101..500]

compr3 = Compr (Var "x") [CCFor "x" range_x, CCIf (Not (Oper Mod (Var "x") (Const $ IntVal 7)))]
compr3_expected = ListVal $ map IntVal [x | x <- [-50..50], mod x 7 == 0]


compr4_cond = Oper Greater (Oper Mod (Oper Plus (Var "x") (Var "y")) (Const $ IntVal 27))
                    (Oper Plus (Var "y") (Const (IntVal 13)))
compr4 = Compr (Oper Div (Oper Times (Var "x") (Var "y")) (Var "z"))
               [CCFor "x" range_x, CCFor "y" range_y, CCIf compr4_cond, CCFor "z" range_z]
compr4_expected = ListVal $ map IntVal [div (x * y) z | x <- [-50..50],
                                                        y <- [127, 103..(-208)],
                                                        mod (x + y) 27 > y + 13,
                                                        z <- [1, 101..500]]

comprTests :: TestTree
comprTests = testGroup "Compr tests"
  [
   testCase "Compr1: empty result" $
     runComp (eval compr1) [] @?= (Right compr1_expected, [])

  ,testCase "Compr2: first clause an if" $
     runComp (eval compr2) [] @?= (Right compr2_expected, [])

  ,testCase "Compr3: simple comprehension" $
     runComp (eval compr3) [] @?= (Right compr3_expected, [])

  ,testCase "Compr4: contrived comprehension" $
     runComp (eval compr4) [] @?= (Right compr4_expected, [])
  ]

compr5 = Compr (Var "x") [CCFor "x" (Const $ IntVal 4)]

comprNegTests :: TestTree
comprNegTests = testGroup "Compr negative tests"
  [
   testCase "Compr5: non-list in generator" $ runComp (eval compr5) []
     @?= (Left $ EBadArg "Compr error: generator expects list expression", [])
```

```
  ]


evalTests :: TestTree
evalTests = testGroup "eval tests"
  [
  comprTests
 ,comprNegTests
  ]
```

B.5 code/part2/tests/AstTests.hs

```haskell
module AstTests where

import Test.Tasty
import Test.Tasty.HUnit

import BoaAST
import BoaInterp

------------------------------
----- programs from ASTs -----
------------------------------
astTests :: TestTree
astTests = testGroup "Programs in ASTs from files"
  [
   testProgFromFile "misc" Nothing
  ,testProgFromFile "contrived_comprehension" Nothing
  ,testProgFromFile "variable_overshadowing" Nothing
  ,testProgFromFile "print_bomb" Nothing
  ,testProgFromFile "name_error"     (Just (EBadVar "composites"))
  ,testProgFromFile "zero_div_error" (Just (EBadArg "Div error: zero divisor"))
  -- TODO: add more negative tests
  ]


baseTestDir = "tests/progs/"

testProgFromFile :: String -> Maybe RunError -> TestTree
testProgFromFile testName maybeError =
  let ast_file  = baseTestDir ++ testName ++ ".ast"
      out_file  = baseTestDir ++ testName ++ ".out"
      test_name = "Running AST from " ++ ast_file

  in testCase test_name $
       read <$> readFile ast_file
         >>= \ast_in -> readFile out_file
         >>= \expected_out -> execute ast_in @?= (lines expected_out, maybeError)
```

C Code: Warmup exercise

C.1 code/part1/Warmup.hs

```haskell
-- Edit all the definitions with "undefined"
module Warmup where

import Control.Monad

type ReadData  = Int
type WriteData = String  -- must be an instance of Monoid
type StateData = Double

-- Plain version of RWS monad
newtype RWSP a = RWSP {runRWSP :: ReadData -> StateData -> (a, WriteData, StateData)}

-- complete the definitions
instance Monad RWSP where
  return a = RWSP $ \_ s -> (a, mempty, s)
  m >>= f  = RWSP $ \r s -> let (a, w, s')    = runRWSP m     r s
                                (a', w', s'') = runRWSP (f a) r s'
                            in  (a', mappend w w', s'')


-- No need to touch these
instance Functor RWSP where
  fmap = liftM
instance Applicative RWSP where
  pure  = return
  (<*>) = ap

-- returns current read data
askP :: RWSP ReadData
askP = RWSP (\r s -> (r, mempty, s))  -- freebie

-- runs computation with new read data
withP :: ReadData -> RWSP a -> RWSP a
withP r' m = RWSP (\_r s -> runRWSP m r' s)

-- adds some write data to accumulator
tellP :: WriteData -> RWSP ()
tellP w = RWSP (\_ s -> ((), w, s))

-- returns current state data
getP :: RWSP StateData
getP = RWSP (\_ s -> (s, mempty, s))

-- overwrites the state data
putP :: StateData -> RWSP ()
putP s' = RWSP (\_ _ -> ((), mempty, s'))

-- sample computation using all features
type Answer = String
sampleP :: RWSP Answer
sampleP =
  do r1 <- askP
     r2 <- withP 5 askP
     tellP "Hello, "
     s1 <- getP
     putP (s1 + 1.0)
```

```haskell
    tellP "world!"
    return $ "r1 = " ++ show r1 ++ ", r2 = " ++ show r2 ++ ", s1 = " ++ show s1

type Result = (Answer, WriteData, StateData)

expected :: Result
expected = ("r1 = 4, r2 = 5, s1 = 3.5", "Hello, world!", 4.5)

testP = runRWSP sampleP 4 3.5 == expected

-- Version of RWS monad with errors
type ErrorData = String
newtype RWSE a = RWSE {runRWSE :: ReadData -> StateData ->
                                    Either ErrorData (a, WriteData, StateData)}

-- Hint: here you may want to exploit that "Either ErrorData" is itself a monad
instance Monad RWSE where
  return a = RWSE $ \_ s -> Right (a, mempty, s)
  m >>= f = RWSE $ \r s -> runRWSE m r s >>=
         \(a, w, s')    -> runRWSE (f a) r s' >>=
         \(a', w', s'') -> Right (a', mappend w w', s'')


instance Functor RWSE where
  fmap = liftM
instance Applicative RWSE where
  pure = return
  (<*>) = ap

askE :: RWSE ReadData
askE = RWSE (\r s -> Right (r, mempty, s))

withE :: ReadData -> RWSE a -> RWSE a
withE r' m = RWSE (\_ s -> runRWSE m r' s)

tellE :: WriteData -> RWSE ()
tellE w = RWSE (\_ s -> Right ((), w, s))

getE :: RWSE StateData
getE = RWSE (\_ s -> Right (s, mempty, s))

putE :: StateData -> RWSE ()
putE s' = RWSE (\_ _ -> Right ((), mempty, s'))

throwE :: ErrorData -> RWSE a
throwE e = RWSE (\_ _ -> Left e)

sampleE :: RWSE Answer
sampleE =
  do r1 <- askE
     r2 <- withE 5 askE
     tellE "Hello, "
     s1 <- getE
     putE (s1 + 1.0)
     tellE "world!"
     return $ "r1 = " ++ show r1 ++ ", r2 = " ++ show r2 ++ ", s1 = " ++ show s1

-- sample computation that may throw an error
sampleE2 :: RWSE Answer
sampleE2 =
  do r1 <- askE
     x <- if r1 > 3 then throw "oops" else return 6
     tellE "Blah"
     return $ "r1 = " ++ show r1 ++ ", x = " ++ show x
```

```haskell
testE = runRWSE sampleE 4 3.5 == Right expected
testE2 = runRWSE sampleE2 4 3.5 == Left "oops"

-- Generic formulations (nothing further to add/modify)

-- The class of monads that support the core RWS operations
class Monad rws => RWSMonad rws where
  ask :: rws ReadData
  with :: ReadData -> rws a -> rws a
  tell :: WriteData -> rws ()
  get :: rws StateData
  put :: StateData -> rws ()

-- And those that additionally support throwing errors
class RWSMonad rwse => RWSEMonad rwse where
  throw :: ErrorData -> rwse a

-- RWSP is an RWS monad
instance RWSMonad RWSP where
  ask = askP
  with = withP
  tell = tellP
  get = getP
  put = putP

-- So is RWSE
instance RWSMonad RWSE where
  ask = askE
  with = withE
  tell = tellE
  get = getE
  put = putE

-- But RWSE also supports errors
instance RWSEMonad RWSE where
  throw = throwE

-- Generic sample computation, works in any RWS monad
sample :: RWSMonad rws => rws Answer
sample =
  do r1 <- ask
     r2 <- with 5 ask
     tell "Hello, "
     s1 <- get
     put (s1 + 1.0)
     tell "world!"
     return $ "r1 = " ++ show r1 ++ ", r2 = " ++ show r2 ++ ", s1 = " ++ show s1

-- Generic sample computation, works in any RWS monad supporting errors
sample2 :: RWSEMonad rwse => rwse Answer
sample2 =
  do r1 <- ask
     x <- if r1 > 3 then throw "oops" else return 6
     tell "Blah"
     return $ "r1 = " ++ show r1 ++ ", x = " ++ show x

testP' = runRWSP sample 4 3.5 == expected
testE' = runRWSE sample 4 3.5 == Right expected
testE2' = runRWSE sample2 4 3.5 == Left "oops"

allTests = [testP, testP', testE, testE2, testE', testE2']
```