AP Assignment 1:
Interpreting Arithmetic Expressions

Nikolaj Gade (qhp695), Sebastian Larsen Prehn (vpz655), Alba Dekens (dmb904)

1 Introduction

The implemented code compiles and passes all the tests which we wrote for it. Our testing is not exhaustive, but tests the general functionality of all the functions implemented.

1.1 Running tests

The tests were run using `cabal 3.6.2.0`, `cabal 3.10.3.0`, and `cabal 3.12.1.0`, successfully passing all the tests on all three versions. In order to run the tests, `cabal test` can be run in the terminal in the same folder as `a1.cabal`.

2 Functions

The `Lambda` evaluation is relatively simple, as there is no actual evaluation happening. It merely returns the data it is given in the `ValFun` structure.

```
82  eval env (Lambda var e) = Right $ ValFun env var e
```

The `Apply` evaluation first runs `eval` on the function expression and the argument expression. We then use pattern matching to make sure they are of the correct types. If they are, we extend the environment and evaluate the function body with the new environment. If not, a error is returned.

```
83  eval env (Apply e1 e2) =
84    case (eval env e1, eval env e2) of
85      (Left err, _) → Left err
86      (_, Left err) → Left err
87      (Right (ValFun env2 var e3), Right x)   → eval (envExtend var x env2) e3
88      (_, _)                                  → Left "Applying non-function"
```

3 Try-catch

The `TryCatch` evaluation evaluates the first expression and uses pattern matching in order to differentiate between a success and a failure. If it succeeds, the result is returned. If not, the result of the second expression is returned.

```
89  eval env (TryCatch e1 e2) =
90    case (eval env e1) of
91      (Right x) → Right x
92      (Left _) → eval env e2
```

4 Pretty-printer

We've implemented `printExp` similarly to `eval`, where each possible `Exp` value is pattern matched. The function is then called recursively for sub-expressions.

```haskell
printExp :: Exp -> String
printExp (CstBool b)      = show b
printExp (CstInt i)       = show i
printExp (Add e1 e2)      = "("++(printExp e1)++" + "++(printExp e2)++")"
printExp (Mul e1 e2)      = "("++(printExp e1)++" * "++(printExp e2)++")"
printExp (Sub e1 e2)      = "("++(printExp e1)++" - "++(printExp e2)++")"
printExp (Div e1 e2)      = "("++(printExp e1)++" / "++(printExp e2)++")"
printExp (Pow e1 e2)      = "("++(printExp e1)++" ** "++(printExp e2)++")"
printExp (If e1 e2 e3)    = "(if "++(printExp e1)++" then "++(printExp e2)++" else "++(printExp
    e3)++")"
printExp (Var var)        = var
printExp (Let var e1 e2)  = "(let "++var++" = "++(printExp e1)++" in "++(printExp e2)++")"
printExp (Lambda var e)   = "(\\"++var++" -> "++(printExp e)++")"
printExp (Apply e1 e2)    = "("++(printExp e1)++" "++(printExp e2)++")"
printExp (TryCatch e1 e2) = "(try "++(printExp e1)++" catch "++(printExp e2)++")"
```

5 Questions

1. `Apply` evaluates both expressions at the same time. In case of errors, the first ones caught are those expressions'. Then the case of the function expression not being a `ValFun` is handled. The opposite would be a better implementation, as it would cause `Apply` to fail sooner in the case where both the function expression was not a `ValFun`, and the argument expression required much computation time to be evaluated.

2. The order of evaluation for `TryCatch` does not matter to the final result. However, for the sake of runtime, evaluating the first expression first would be faster, since the second might not need to be evaluated.

3. Using a Y combinator, it is possible for `eval` to loop indefinitely. For example, applying the Y combinator to a lambda returning itself would result into a infinite loop.