Assignment 0:

Philip Rajani Lassen, Nicolas Ringsmose Larsen

September 2018

1 Simple Arithmetic Expressions

1.1 Printing Expressions

The first function printing expressions was fairly easy to write with a recursive implementation. We elected to not handle the redundant brackets. The only bug we ran into was representing negative numbers, however this was easily fixed with an extra set of parenthesis.

1.2 Evaluating Expressions

The simpleEval came fairly quickly. We did get a bug when we ran unit tests on the Power evaluation. Due to Haskell's lazy evaluation when the following expression was evaluated

Pow (Pow (Cst 2) (Cst (-1))) (Cst 0) = $(2^{(-1)})^0$

The reason that our original implementation failed that we had not considered the order of operations. Our initial implementation ignored Haskell's lazy implementation and we therefor got the result $((2^{(-1)})^0 = 1$ instead of getting an error for taking 2^{-1} as the specification specified. Once we realized the problem we made an incredibly hacky solution

```
evalSimple (Pow exp1 exp2) = let x = evalSimple exp1 in
                             if x == 0
                             then x ^ evalSimple exp2
                             else x ^ evalSimple exp2
```

This disturbing implementation forced x to be evaluated by Haskell and then regardless of whether x evaluated to 0 or not, proceeded to enter an if then else branch that did the exact same thing. Luckily cooler heads prevailed and we realized that their must be an idiomatic and less idoitc approach to solving the problem of overly lazy evaluation.

Instead we used the Seq datatype to introduce strictness to the power functions. This way we could force the base number to be evaluated first rather than just returning 1 on any n^0 and this solved the issue.

2 Extended arithmetic expressions

One of the difficulties we had was understanding how a function could be used to represent the environment. This difficulty stemmed from our background in imperative programming, where we felt that a map or list would be used to represent the environment. The other difficulty was figuring out how to create a function that took a function as an argument and returned a new function. Once we wrapped our minds around how this worked and could be implemented the rest of code fell into place.

3 Returning explicit errors

3.1 Initial Implementation

Our first attempt at returning an Either ArithErr Integer went relatively smoothly. The hardest part was changing our implementation of our helper function for evaluating summation expression. The difficulty came from the fact that we needed to add Right Integers instead of just Integers, while also handling Left ArithErr returned in any of the calls to the body of the summation appropriately.

3.2 Re-factoring

Our initial code had a fair deal of redundant code. Thus it made sense to try and re-factor the code. We were able to do this for 5 different expression types, which were (Plus, Mul, Sub, Div, Pow). There was a clear way to factor the code for Plus, Mul, and Sub as the code was ostensibly the same just with a different operator. We could simply create a helper function that took in a function which would be either (+), (*), or (-). However there were some clear patterns between Plus, Mul, Sub and Div, Pow. We decided to make our helper function also applicatble to those functions. This increased the complexity of our code. The type of the helper function became

```
Exp -> Exp -> Env -> (Integer -> Integer ->
Either ArithError Integer) -> Either ArithError Integer
```

From the type its quite clear that the code became relatively clunky. The helper function now takes a more complicated function as an argument. This function takes two Integers as arguments and returns an Either ArithErr Integer. The updated implementation reduces the total number of lines of the whole program but in some ways increases the complexity of the code. It is debatable whether the factoring is worth the increase in complexity.

4 Testing

The way we tested our code was by running tests in the Haskell Read Eval Print Loop (REPL) after we were finished writing a function. We then wrote Tests into Arrays and a function that made sure that all the tests cases evaluated to True. After making sure all these tests were passed and that we passed the online TA's tests we felt comfortable with the accuracty of our implementations. The result of running the main function is shown below, and the test file *Tests.hs* is included in the *src.zip*

```
*Main> testAll
True
```

5 Assessment

All in all the assignment went well. Once we became familiar with Haskell syntax and Haskell paradigms, the code came very quickly. The biggest problems in our code as we alluded to in the re-factoring section pertain to the readability of our code. We decided to make the code less redundant at the cost of readability. However there still exist redundant code. It is highly likely that with programming paradigms such as Monads we would be able to eliminate such redundancy. This is a construct that we were not familiar with and therefore elected not to use at the time.