TEXAS A&M UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CSCE 314: Programming Languages

Homework Assignment 6

Objective

In this assignment you'll operate on expression trees. We're beginning to get within striking distance of programming languages of our own!

Submission

- Like before, this homework is will be graded as a 0, 1, or 2 for each question. (There are two questions: "Evaluating expressions", "Transforming expressions"). With 0 = inadequate or no attempt; 1 = solid effort showing a good attempt at all questions; 2 = faultless/exemplary submission. You may find it helpful to discuss your homework with the peer teachers, TAs, or instructors (say, at their office hours).
- Prepare a **single** Haskell file with a `.hs` extension.
- *Recommended*: Include as a comment at the top of your code: your name and UIN.
- The file should contain functions named precisely as described below. Those functions must have proper type signatures and should follow exactly the ones that have been explicitly given. In addition, you may include any number of additional helper functions you like; you may not include any extra modules, packages, or external libraries.
- Additional documentation in the form of comments throughout your code is strongly encouraged. If you have used outside resources, document these directly in your source file.
- You are encouraged to use the class slack to communicate about/solicit help/ask questions clarifying aspects of the assignment.
- Submission will be via Canvas, due: 17 November 2021.

A tiny language of expressions

Let *E* be a tiny programming language that supports the declaration of arithmetic expressions and equality comparisons on integers. Here is an example program in *E*:

```
1 + 2 == 5 - ( 3 - 1 )
```

When evaluated, this program should evaluate to the truth value true. In this exercise, we will not write *E* programs as strings, but as values of a Haskell data type `E`, that can represent *E* programs as their abstract syntax trees (ASTs). Given the following data type `E`,

```
data E = IntLit Int
       | BoolLit Bool
       | Plus E E
       | Minus E E
       | Multiplies E E
       | Exponentiate E E
       | Equals E E
         deriving (Eq, Show)
```

The above example program is represented as its AST:

```
program = Equals
            (Plus (IntLit 1) (IntLit 2))
            (Minus
              (IntLit 5)
              (Minus (IntLit 3) (IntLit 1)))
```

Another example is that the program for

```
1 + 2 == 5 - 3^1
```

is

```
program' = Equals
            (Plus (IntLit 1) (IntLit 2))
            (Minus
              (IntLit 5)
              (Exponentiate (IntLit 3) (IntLit 1)))
```

Evaluating expressions

Define an evaluator for the language *E*. Its name and type should be

```
eval :: E -> E
```

The result of `eval` should not contain any operations or comparisons, just a value constructed either with `IntLit` or `BoolLit` constructors. The result of the example program above should be `BoolLit True`. Note that *E* allows nonsensical programs, such as `Plus (BoolLit True) (IntLit 1)`. For such programs, the evaluator can abort. Here are examples:

```
> eval (Equals (Plus (IntLit 1) (IntLit 2)) (Minus (IntLit 5) (Minus (IntLit 3) (IntLit 1))))
BoolLit True
> eval (Plus (BoolLit True) (IntLit 1))
*** Exception: hw6.hs:160:1-52: Non-exhaustive patterns in function insideInt
```

Transforming expressions

When simplifying expressions, we often employ transformations that preserve equality. For simplifying equations over the reals one familiar example is "taking logs on both sides."

The idea of transforming an expression from something that involves costly operations to another with potentially cheaper ones, but which preserves equality, is common in applied mathematics, optimization, and computer science. Taking logs, transforms multiplications into additions, and exponentiations into multiplications, etc. When the latter operation is computationally cheaper, this can be a worthwhile step.

Define a function for *E* that preserves the truth value of the expression by taking logs to base 2 if it is possible. If it isn't possible, the program may abort (e.g., with a Non-exhaustive pattern exception).

```
log2Sim :: E -> E
```

Here are some examples:

```
> log2Sim (IntLit 8)
IntLit 3

> log2Sim (IntLit 7)
*** Exception: hw6.hs:(177,1)-(183,55): Non-exhaustive patterns in function log2Sim
```

```
> log2Sim (Multiplies (IntLit 1) (Exponentiate (IntLit 1) (IntLit 3)))
Plus (IntLit 0) (Multiplies (IntLit 3) (IntLit 0))

> let p2 = (Equals (BoolLit True) (Equals (IntLit 16) (Multiplies (IntLit 4) (Exponentiate (IntLit 2) (

> log2Sim p2
Equals (BoolLit True) (Equals (IntLit 4) (Plus (IntLit 2) (Multiplies (IntLit 2) (IntLit 1))))

> eval it
BoolLit True

> eval p2
BoolLit True

> log2Sim (Equals (BoolLit True) (Equals (IntLit 16) (Multiplies (IntLit 4) (Plus (IntLit 2) (IntLit 2)
(some output...)  Exception: hw6.hs:(177,1)-(183,55): Non-exhaustive patterns in function log2Sim
```