Exercise3

Attached Files:

- Makex3.zip (21.487 KB)
- Exercise3.zip (5.855 KB)

Topic: Symbolic Programming

Instructions:

- Download M**akex3.zip**, uncompress (produces M**akex3** folder, and use your command-line/terminal window inside that folder to issue the command: **stack install**. If that succeeds, then you will have installed an executable called **makex3**.
- Go to the folder (FPCW say) where you put Exercise1. It should still have the **makex.conf** file left over from that exercise. Download and uncompress **Exercise3.zip** here.
- Execute the following in FPCW: **makex3**. You should not need to re-enter any details. It should create file **Exercise3/src/Ex3.hs**.
- Enter Exercise3 and give the command **stack install**, which will install an executable called **ex3**. Running ex2 will result in a short message.
- Your task is to edit **Ex3.hs** (only) so that it produces the correct output - this is defined in the comments in **Ex3.hs**.
- You are reccommended to use **stack ghci src/Ex3.hs** to observe and debug your code. You can re-compile your code using **stack install**, and the re-enter ghci as above
- Your submission is ONLY the file **Ex3.hs** (as is, do not compress/tar it in any way)

This exercise supplies you with a tailored partial expression data-type, a dictionary implementation and asks you to write three functions:

- Function **eval** evaluates the expression without handling errors.
- Function **meval** is like eval but does handle all errors
- Function **simp** simplifies expressions use a specified collection of laws

Notes:

To understand the required functionality, please review the lecture notes regarding the **Expr** datatype (end of week4, week5, week6).

Function **eval** will only be tested in situations were all the variables are in the dictionary, and there is no division by zero.

For **simp**, there are only two simplification laws you have to implement, depending on which binary operator you have (+,*,-).

Q&A

- Q: here are some test results. Have I got it right?
  A: I am not going to respond to these queries - I am not doing all your testing for you.
- Q: Is it OK if **eval** returns such-and-such a value if an error occurs
  A: Function **eval** will only be checked by me using safe values so no errors arise
- Q: Is it OK if **meval** returns a number when an error occurs?
  A: No, the whole point of **meval** is that the caller knows if they have messed up as they get nothing. The problem with returnng just some number is the caller has no idea that something is not right. Fight that instinct to "fix" code like this - it's dangerous!

- Q: should a "logical" operation return a result based on the expressions *themselves*, or on the *evaluation* of those expressions?
  A: It should depend on the ***evaluations*** of those expressions.
  The logical operations have the same status as the arithmetic ones in that they are also being evaluated.
- Q: should **simp** just simply once?
  A: No, it should simplify as much as possible
- Q: For **simp** do we need to do any evaluation at all?
  A: No, **simp** should only apply the relevant simplification laws
- Q: For the dictionary are we supposed to assume that it is already populated? ie are we supposed to insert anything into the dictionary as part of our **eval**/**meval** code?
  A: See notes above Q&A about how your code will be evaluated. However, you will need to create your own dictionaries to test your code.
- I get a result for negation of **-0.0**. Is this OK?
  Haskell says. **0.0 == -0.0** is **True**, so either "value" is fine for tests
- Q: In **meval**, doing a comparison, is it ok to say (for example) that 42.0 = Nothing is False, and Nothing = Nothing is True?
  A: No, **each** Nothing indicates **some specific** error condition, comparing these is generally meaningless.