Monadic Parsing

For this assignment, the code discussed in lecture 6 on *monadic parsing*, for evaluating expressions, should be expanded. The definition of expressions are (informally) generalized in the following ways.

- Constant values in expressions are Doubles, for example 2, 2.0, -2, 2.3e4
- In addition to the operators + and *, the following operators should also be supported: / (division), - (subtraction), ^ (exponentiation). Take careful note of operator associativity and precedence. Also, unary minus should have high enough precedence so that -2^2 = 4.
- Recall that parenthesis may be used to change operator precedence.
- Change the type of the function *eval* to *eval:: String -> [Double]*, but to support error messages, consider using *eval:: String -> Either String [Double]*
- *eval* is evaluated on a string that is of the form string1;string2;string3;...;stringn with the following properties:
    - None of string1, string2, string3,... contains the character ";"
    - string1, string2, string3, ... may be a string constructed from operators and constants.
    - string1, string2, string3, ...may be a comment which is defined to be a string beginning with "--", for example, "-- this is just a comment " (assume comments do not contain the character ";")
    - string1, string2, string3, ... may be a variable definition, which is of the form "ident = expr", where "ident" is a string with restrictions in terms of lower and alphanumeric characters as in the code skeleton and "expr" is an expression consisting of constants, operators and previously defined variables
    - string1;string2;string3;... may contain arbitrary whitespace (including tabs and newlines), but not as part of constants or variable names.
    - Example 1: eval "2*2;-- 4.0;1*0;--0.0" returns [4.0,0.0]
    - Example 2: eval "2/2/2" returns [0.5]
    - Example 3: eval "4^2^2" returns [256.0]
    - Example 4: eval "x=3;var = 4; x + var" returns [7.0]
    - Example 5: eval "1+y" throws an exception
    - Example 6: eval "y=2;y=3;y=5;1+y" returns [6.0]
    - Example 7: eval "y=2;y=3;y=5" returns [ ]
- Define a type *Ast* that derives *Show* that represents the abstract syntax tree of an expression. Next define functions *produce_Ast::String->Ast* and *eval_Ast::Ast->[Double]* such that *eval_Ast . produce_Ast = eval*
- Handle the follow errors in a graceful way rather than throwing exceptions (or not reporting them at all):
    - Overflow of Doubles. Make use of `Prelude.floatRange.`
    - Division by zero
    - Parsing errors
    - Use of variables before defining them

Requirements

Use stack and supply your own Hspec test cases. Give a brief description in the Readme on how to run the test cases and the functionality of the evaluator in general. The tests must validate the resulting Ast type using the *eval* function.

Furthermore, tests can be subdivided as follows:

- Test each of the operators in isolated binary evaluations, e.g. 1+1, 2*3
- Test an example of chaining these operators together. E.g. 1*⅔-4
- Test parsing variable definitions and using variables in the place of constants in the tests from the two points above
- Test all 4 errors sufficiently
- Take the 4 points above as a departure point for testing, but test thoroughly in general, for example, are whitespaces and floating point numbers handled correctly, can sensible QuickCheck test cases be given, etc.