lab session 2b: Functional Programming

In the second part of lab session 2 we will focus on data types. In this exercise we will make a program that can parse and manipulate integer expressions.

Integer valued Arithmetic Expressions

In this lab we will consider integer valued arithmetic expressions that can be constructed using the following grammar:

```
E  -> T E'
E' -> + T E' | - T E' | <empty string>
T  -> F T'
T' -> * F T' | / F T' | % F T' | <empty string>
F  -> (E) | <integer> | <variable>
```

Note that the operator '/' is actually the integer `div` operator, and that '%' is the `mod` operator.

The notation `<integer>` denotes an integer literal (like 42), and `<variable>` denotes the name of a variable. The name of a variable is case sensitive (so `"a"` is not the same as `"A"`). A variable name can only be a string of letters, so `aOne` is fine, while `a1` is not allowed.

Some examples of valid expressions are:

```
42
1+2+3+4*5*6+7
5+a*6+b*c
(5+a)*(6+b)*c
```

Kick-off

Start by making a file containing the following types:

```
type Name   = String
type Domain = [Integer]

data Expr = Val Integer
          | Var Name
          | Expr :+: Expr
          | Expr :-: Expr
          | Expr :*: Expr
          | Expr :/: Expr
          | Expr :%: Expr
```

Exercise 1: `show`

Using this data type, we can represent the expression `x+2*3` as `(Var "x") :+: (Val 2 :*: Val 3)`.

As a computer representation for expressions, this is fine. For a human reader this notation is quite cumbersome. Therefore, make the type `Expr` member of the class `Show` by implementing a function `show` that pretty prints expressions. A succesfull implementation makes sure that `ghci` will respond like in the following session:

```
*Expression> (Var "x") :+: (Val 2 :*: Val 3)
(x + (2*3))
```

Note that the parentheses are superfluous, but that is acceptable. If you wish, you can make a neat version that omits superfluous parentheses, but that is more work.

Exercise 2: Variables in an `Expr`

Write a Haskell function `vars` that returns the sorted list of variables that occur in an expression. For example, `vars (Var "b" :+: (Var "a" :+: Var "b"))` should yield the result `["a","b"]`.

Exercise 3: Evaluating `Exprs`

Of course, we want to evaluate an `Expr`. However, in order to do this, you need to know a value for each variable in an `Expr`. Therefore, we introduce the type `Valuation`, which is a list of variable name-value pairs.

```
type Valuation = [(Name, Integer)]
```

Now that we have a type `Valuation`, we can evaluate `Exprs`. Given an `Expr e` and a `Valuation v` we can make a function `evalExpr e v` that returns an `Integer`, the result of evaluating the expression given the values for the variables. Note that, if we want to evaluate an expression that contains no variables at all, we can choose `v` to be anything, including the empty valuation, i.e. `evalExpr e []`. Implement the function `evalExpr`. Here is a log of an example session:

```
*Expression> evalExpr ((Val 1) :+: (Val 2 :*: Val 3)) []
7
*Expression> evalExpr ((Val 1) :+: (Val 2 :*: Val 3)) [("a",42)]
7
*Expression> evalExpr ((Val 1) :+: (Val 2 :*: Var "a")) [("a",3)]
7
```

[Note: It is a good start to verify, before evaluating, whether a valuation is complete. If it is not, this function should give an error message and abort]

Next, make a function `valuations :: [(Name,Domain)] -> [Valuation]` that returns all `Valuations` for the variables given a domain of allowed values for each variable. Here is the log of an example session:

```
*Valuation> valuations [("a", [1,2]), ("b", [1..3])]
[[("a",1),("b",1)],[("a",1),("b",2)],[("a",1),("b",3)],[("a",2),("b",1)],
[("a",2),("b",2)],[("a",2),("b",3)]]
```

Using the data type `Expr`, and the functions `evalExpr` and `valuations`, compute the list `pytriples n` of all Pythagorean triples `(a,b,c)` with $\mathtt{c} \leq \mathtt{n}$. A triple `(a,b,c)` is a Pythagorean triple if $\mathtt{a} \leq \mathtt{b}$ and $\mathtt{a}^2 + \mathtt{b}^2 = \mathtt{c}^2$.

For example, `pytriples 10` yields `[[("a",3),("b",4),("c",5)],[("a",6),("b",8),("c",10)]]`.

Exercise 5: Parsing expressions: `String` to `Expr`

Using the `show` function, it is easy to print expressions on the screen in a human readable form. However, as you may have noticed by now, is would also be handy to have a function that does the opposite: converting a string (like `"a+3*b"`) into an `Expr`. Make a Haskell function `toExpr :: String -> Expr` that performs this task. It is probably smart to implement a helper function `tokenize` first, that transforms a string expressions into a list of tokens. For example, `tokenize "a + 3*b"` should yield the result `["a","+","3","*","b"]`. Note that `tokenize` skips spaces. Once you made `tokenize`, you can implement a simple parser for expressions. Use the grammar given in the introduction for this. You are not allowed to make use of parser libraries that are available on the internet: this grammar is very small, so writing a parser for it by hand does not result in a lengthy code.

Once you made the parser (and the previous exercise), you should be able to run a `ghci` session like the following one:

```
*Expression> toExpr "2*a+b"
((2*a) + b)
*Expression> evalExpr  (toExpr "2*a+b") [("a",2), ("b",1)]
5
```

Bonus exercise: Simplifying expressions

If you solve this problem, you get a bonus grade point (although you cannot get a grade > 10).

Some expressions can be simplified. For example, the expressions `1+1+a` can be simplified into `2+a`. And, what about `a+1+1`? It can be simplified to `a+2`, but it is harder (why?).
An other example is `a+2*21+2*(3+5)*b`, which can be simplified into `a+42+16*b`.

Write a Haskell function `simplifyExpr` that takes as its input an `Expr`, and outputs a simplified expressions in which constant subexpressions (so, without variables) are simplified. Note that you do not have to simplify expressions like `1 + a + 1` into `a+2` (although you are welcome to try it: you will notice that this is even more difficult).