lab session 2b: Functional Programming

Part 2b (part2a + part2b together is lab 2) is worth 50 points in total. It consists of 3 exercises (exercise 6, 7, and 8). The exercises 6 and 8 are worth 20 points, while exercise 7 is worth 10 points. Note that passing all tests in Themis will not automatically mean that you scored 50 points. The TAs will manually check accepted solutions for a Haskellish style of programming.

Integer valued Arithmetic Expressions

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

Kick-off

Start by making a file containing the following types:

```
type Name   = String

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

Exercise 6: Showing Parsed `Exprs`

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 for the application that we are going to make. Using the `show` function, it is easy to print expressions on the screen in a human readable form. However, we also want 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. Note that this function is expected to skip spaces (so the input `"a+3*b"` should yield the same output as the input `"a + 3* b"`). 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 `show` function on `Expr`s, you should be able to run a `ghci` session like the following one:

```
*Expression> toExpr "2*(a+3*b+c -  e)%(c-d)"
((2*(((a+(3*b))+c)-e))%(c-d))
```

The file that you submit to Themis must have the file name `toExpr.hs`.

Exercise 7: Variables in an Expression

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

The file that you submit to Themis must have the file name `vars.hs`. This file must be an extension of the file `toExpr.hs` from the previous exercise. So, Themis expects functions from the previous exercise to be present in the file `vars.hs`.

Exercise 8: 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. The list is sorted lexicographically (alphabetical order) on the variable names.

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

Now that we have a type `Valuation`, we can evaluate `Expr`s. Given an `Expr e` and a `Valuation v` we can make a function `evalExpr e v` that returns a `Maybe Integer`. The function should return `Just x`, where `x` is the result of evaluating the expression if a value is given for all variables in the expression and the expression can be evaluated (e.g. no division by zero). If one or more variables have no value in the valuation `v` or the expression cannot be evaluated, then the function should return `Nothing`. 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`. The file name of your submission to Themis must be `evalExpr.hs`.

Here is a log of an example session:

```
*Expression> evalExpr ((Val 1) :+: (Val 2 :*: Val 3)) []
Just 7
*Expression> evalExpr ((Val 1) :+: (Val 2 :*: Val 3)) [("a",42)]
Just 7
*Expression> evalExpr (toExpr "1+2*a") [("a",3)]
Just 7
*Expression> evalExpr (toExpr "1+2*a") [("b",3)]
Nothing
```