CSCC24 & CSC324 Winter 2018 – Assignment 4
Due: Friday, April 5, midnight
This assignment may be done in pairs.
This assignment is worth 10% of the course grade.

In this assignment, you will write a parser (question 1) and an interpreter (question 2) for a simple functional language.

As usual, you should also aim for reasonably efficient algorithms and reasonably good and simple coding style.

1. [8 marks] Implement a parser for the following language. The syntax is defined by starting with the following grammar in EBNF, and then there are amendments and remarks afterwards.

```
block    ::= cond | lambda | let | infix
infix    ::= arith [ cmp arith ]
cmp      ::= "==" | "<"
arith    ::= { addend addop } addend
addop    ::= "+" | "-"
addend   ::= { factor mulop } factor
mulop    ::= "*" | "/" | "%"
factor   ::= { atom } atom
atom     ::= "(" block ")" | literal | var
cond     ::= "if" block "then" block "else" block
lambda   ::= "\" var "->" block
let      ::= "let" { equation } "in" block
equation ::= var "=" block ";"
literal  ::= integer | boolean
boolean  ::= "True" | "False"
```

   Amendments and remarks:

   - The start symbol is block.
   - Two non-terminals, integer and var, are specified informally by:
     - integer: An optional minus sign, followed by one or more decimal digits.
     - var: A letter, followed by zero or more letters or digits. However, reserved words are not allowed to be vars. The reserved words are:
       `if`, `then`, `else`, `let`, `in`, `True`, `False`.
       If the parser sees a reserved word when a var is expected, it is considered a parser error.
   - There can be spaces, tabs, and newlines (collectively known as whitespaces) around integers, vars, and terminal strings. For example

```
   let   x
         =  4 * (
  5+6) ;
in \ y ->  if  x< y then x else y
```

is allowed.

The best way to skip whitespaces is to adopt this strategy consistently. The entry-point parser consumes leading whitespaces once and for all. Note that aftewards you can think of all remaining whitespaces as trailing whitespaces. So parsers for integers, vars, and terminal strings consume trailing whitespaces. There are building blocks in ParserLib.hs that do this.

The parser's answer, if there is no error, is an abstract syntax tree represented by the Expr type in A4Def.hs. Most of the expected answers should be self-evident. Here are a few non-obvious points:

- The arithmetic infix operators `+`, `-`, `*`, `/`, `%` associate to the left.
- The grammar rule
  `factor ::= { atom } atom`
  represents function application. When there are 2 or more atoms, the answer is an `App` tree. Function application associates to the left. For example `f x y` is parsed to `App (App (Var "f") (Var "x")) (Var "y")`.

2. [8 marks] Implement an interpreter for the language defined by the Expr type.

The semantics is defined by the following points:

- Evaluation order: Call by value for the most part, including `Let`, `App`, comparison, and arithmetic. The exceptions are in the next point.
- In `Cond`, if the test evaluates to true, evaluate the then-branch but not the else-branch. The opposite if the test evaluates to false.
  If the test evaluates to a non-boolean value, it is a type error.
- Semantics of `Let`: Like `let*` in Scheme. A later equation sees the variables bound by earlier equations, but not the other way round; and recursion is unsupported. For example, think of
  `let x=2+3; y=x+4; in x+y`
  as
  `let x=2+3; in let y=x+4; in x+y`
  Recall that the call-by-value order for `Let` means that in this example the order is `2+3`, then `x+4`, then `x+y`.
- `Div` and `Mod` have the same meaning as Haskell's `div` and `mod`, Python's `//` and `%`.
  If the divisor is zero, it is a divide-by-zero error.

- For simplicity, `Eq` and `Lt` work for integer operands only. If an operand is not an integer, it is a type error.
- Perform run-time checks that operands have the correct types, variables are bound, and divisors are non-zero. The interpreter uses `Left` to indicate failure, with a value of type `Error` to indicate the kind of error.

  If an expression contains multiple errors, evaluation order determines which error is seen. For example,

  ```
  let x=2/0; in 1+True
  ```

  is a divide-by-zero error because `2/0` is evaluated first.
- Most other unspecified aspects follow common conventions, e.g., `Plus` means adding two integers.

End of questions.