CSCC24 2024 Summer – Assignment 3
Due: July 28, 11:59PM
This assignment is worth 10% of the course grade.

Question 1: `where`-expressions [10 marks]

In this question, you will implement a recursive descent parser in Haskell. As usual, you should aim for reasonably efficient algorithms and reasonably organized, comprehensible code.

Code correctness (mostly auto-testing) is worth 90% of the marks; code quality is worth 10%.

Mathematicians like to write like "$x + y$ where $x = 4$ and $y = 5$", which corresponds to the `let-in` construct in Haskell (*not* the `where` construct).

But it is possible to design a language to support mathematicians' convention! Here is one such design in EBNF (the start symbol is `<wexpr>`):

```
<wexpr> ::= <expr> [ "where" <def> { "and" <def> } ]
<def> ::= <var> "=" <expr>
<expr> ::= <expr> <binop> <expr>
         | <uop> <expr>
         | <var>
         | <natural>
         | "(" <wexpr> ")"
<binop> ::= "+" | "-" | "*"
<unop> ::= "-"
```

It contains deliberate ambiguity and omissions, to be resolved by these points:

- `<var>` can be done by `identifier` from ParserLib, noting that the reserved words are: `where`, `and`.
- `<natural>` can be done by `natural` from ParserLib.
- The part about

  ```
  <expr> ::= <expr> <binop> <expr> | <uop> <expr>
  ```

  is deliberately ambiguous and left-recursive! It is not ready for recursive descent parsing. You need to rewrite to an unambiguous, non-left-recursive form based on these operator precedence levels, from highest to lowest:

operator	associativity
parentheses	
unary minus	
*	left
binary plus, minus	left

- Whitespaces around tokens are possible.

The abstract syntax tree to build is defined by this data type in WexprDef.hs:

```
data Wexpr
  = Nat Integer
  | Var String
  | Neg Wexpr -- unary minus
  | Plus Wexpr Wexpr
  | Minus Wexpr Wexpr
  | Times Wexpr Wexpr
  | Where Wexpr [(String, Wexpr)]
```

Here are some non-obvious examples of input strings and expected answers:

- Input: `5 - 4 + 3`

  Answer: `Plus (Minus (Nat 5) (Nat 4)) (Nat 3)`

- Input: `5 + - 4` or `5 + -4`

  Answer: `Plus (Nat 5) (Neg (Nat 4))`

- Input: `- - 5`

  Answer: `Neg (Neg (Nat 5))`

- Input: `-- 5`

  Error, `--` is better not treated as two consecutive unary minuses. If you use `operator` from ParserLib, you get this behaviour automatically.

- Input: `5 +- 4` or `5 +-4`

  Error, ditto.

- Input: `foo where y = 5 and z = 1`

  Answer: `Where (Var "foo") [("y", Nat 5), ("z", Nat 1)]`

- Input: `(foo where y = 5) where z = (b where b = 1)`

  Answer:

  ```
  Where (Where (Var "foo") [("y", Nat 5)])
        [("z", Where (Var "b") [(Var "b", Nat 1)])]
  ```

- Input: `foo where y=5 where z=b where b=1`

  Error.

Implement the parser as `wexpr` in WexprParser.hs. My tests will only test `wexpr` directly or via `mainParser` in testParser.hs. You are free to organize your helper parsers.

Question 2: Type Inference [10 marks]

Show type inference steps for the following expression. The initial environment has

```
succ :: Int -> Int
ys :: [Bool]
```

You may omit detailed unification steps, but do show how `unify` calls `unify-intern` for clarity. (Similar to examples in the lecture.)

```
let len = \xs -> case xs of [] -> 0
                            (x:xt) -> succ (len xt)
in len ys
```

The starter file infer.txt has the initial environment and the above expression. Complete and hand in.

End of questions.