# 2019-20 UoB, COMS22201

# Exercises week 19

## Goal of the exercises

The main goal of this week's exercises is to familiarize yourself with regular expressions, automata, grammars, the `alex` lexer generator and the `happy` parser generator.

**Exercise 19.1** Write a regular expression that recognizes all sequences consisting of $a$ and $b$ where two $a$'s are always separated by at least one $b$. For instance, these four strings are legal: $b$, $a$, $ba$, $ababbbaba$; but these two strings are illegal: $aa$, $babaa$.

Construct the corresponding NFA. Try to find a DFA corresponding to the NFA.

**Exercise 19.2** Write out the rightmost derivation of this string from the expression grammar corresponding to `ExprLang/ExprPar.y`.

```
let z = 17 in z + 2 * 3 end
```

(Take note of the grammar rules (1 to 8) used:)

```
Expr    : var                           { Var  $1           } -- Rule 1
        | num                           { CstI $1           } -- Rule 2
        | '-' num                       { CstI (- $2)       } -- Rule 3
        | '(' Expr ')'                  { $2                } -- Rule 4
        | let var '=' Expr in Expr end  { Let $2 $4 $6      } -- Rule 5
        | Expr '*' Expr                 { Prim "*" $1 $3    } -- Rule 6
        | Expr '+' Expr                 { Prim "+" $1 $3    } -- Rule 7
        | Expr '-' Expr                 { Prim "-" $1 $3    } -- Rule 8
```

**Exercise 19.3** Draw the above derivation as a tree.

**Exercise 19.4** From the `ExprLang` directory, use the command prompts to generate (1) the lexer and (2) the parser for expressions by running `alex` and `happy`; then (3) load the expression abstract syntax, the lexer and parser modules, and the expression interpreter and compilers, into an interactive Haskell session (`ghci`):

```
alex ExprLex.x
happy ExprPar.y
ghci Parse.hs Absyn.hs ExprLex.hs ExprPar.hs
```

Now try the parser on several example expressions, both well-formed and ill-formed ones, such as these, and some of your own invention:

```
parseFromString "1 + 2 * 3"
parseFromString "1 - 2 - 3"
parseFromString "1 + -2"
parseFromString "x++"
parseFromString "1 + 1.2"
parseFromString "1 + "
parseFromString "let z = (17) in z + 2 * 3 end"
parseFromString "let z = 17) in z + 2 * 3 end"
parseFromString "let in = (17) in z + 2 * 3 end"
parseFromString "1 + let x = 5 in let y = 7 + x in y + y end + x end"
```

**Exercise 19.5** Using the expression parser from `ExprLang/Parse.hs`, and the expression-to-stack-machine compiler `scomp` and associated datatypes from `ExprLang/Expr.hs`, define a function `compString :: String -> [SInstr]` that parses a string as an expression and compiles it to stack machine code.

**Exercise 19.6** Extend the expression language abstract syntax and the lexer and parser specifications with conditional expressions. The abstract syntax should be `If e1 e2 e3`; so you need to modify file `ExprLang/Absyn.hs` as well as `ExprLang/ExprLex.x` and `ExprLang/ExprPar.y`. The concrete syntax should be in the style:

```
if e1 then e2 else e3
```

Documentation for `alex` and `happy` can be found online.

**Exercise 19.7** (*This question is optional and will require some selfstudy.*) Determine the steps taken by the parser generated from `ExprLang/ExprPar.y` during the parsing of this string:

```
let z = 17 in z + 2 * 3 end
```

For each step, show the remaining input, the parse stack, and the action (shift, reduce, or goto) performed. You will need a printout of the parser states and their transitions to do this exercise; to do this, run the following before calling the `parseFromString` function.

```
happy -da ExprPar.y
ghci Parse.hs Absyn.hs ExprLex.hs ExprPar.hs
```

Sanity check: the sequence of reduce action rule numbers in the parse should be the exact reverse of that found in the derivation in Exercise 19.2.

**Exercise 19.8** Files in the directory `UsqlLang/` contain abstract syntax (file `Absyn.hs`), a lexer specification (`UsqlLex.x`), and a parser specification (`UsqlPar.y`) for micro-SQL, a small subset of the SQL database query language.

Extend micro-SQL to cover a larger class of SQL SELECT statements. Look at the examples below and decide your level of ambition. You should not need to modify file `Parse.hs`. Don't forget to write some examples in concrete syntax to show that your parser can parse them.

For instance, to permit an optional WHERE clause, you may add one more component to the `Select` constructor:

```
data Stmt = Select [Expr]                (* fields are expressions *)
                   [String]              (* FROM ...               *)
                   (Maybe Expr)          (* optional WHERE clause  *)
```

so that `SELECT ... FROM ... WHERE ...` gives `Select ... ... (Just ...)`,
and `SELECT ... FROM ...` gives `Select ... ... Nothing`.

The argument to WHERE is just an expression (which is likely to involve a comparison), as in these examples:

```
SELECT name, zip FROM Person WHERE income > 200000

SELECT name, income FROM Person WHERE zip = 2300

SELECT name, town FROM Person, Zip WHERE Person.zip = Zip.zip
```

More ambitiously, you may add optional GROUP BY and ORDER BY clauses in a similar way. The arguments to these are lists of column names, as in this example:

```
SELECT town, profession, AVG(income) FROM Person, Zip
WHERE Person.zip = Zip.zip
GROUP BY town, profession
ORDER BY town, profession
```