# CMPUT 274—Assignment 5 (Fall 2025)

R. Hackman

Due Date: Monday December 8th, 8:00PM

Per course policy you are allowed to engage in *reasonable* collaboration with your classmates. You must include in the comments of your assignment solutions a list of any students you collaborated with on that particular question.

**For each assignment you may not use any Python features or functions not discussed in class!** This means, for example, you may not use any built-in functions or methods we have not discussed. If you are in doubt, you may ask on the discussion forum — however remember if you are including your code, or even your plan to solve a question, you must make a private post. Use of disallowed features will result in a grade of zero on the question(s) in which they were used.

**All functions you write must have a complete function specification as presented in the course notes.** Failure to provide function specifications for functions will lead to the loss of marks.

Unless explicitly forbidden by a question, you are always allowed to write additional helper functions that may help you solve a problem.

## `fimpl` - **F**un **Im**perative **P**rogramming **L**anguage

We will now expand the `fimpl` grammar to be more than just a calculator that calculates expressions. We will do so by now defining a program to be a series of *statements* and to add three types of statements to our grammar. Our new `fimpl` grammar contains everything from the first assignment as well as the following new symbols and production rules

```
Program     : Statements

Statements  : Statement
            | Statement Statements

Statement   : AsgnStmt
            | DisplayStmt
            | RepeatStmt

AsgnStmt    : Identifier '<-' Expr

RepeatStmt  : 'Rpt' Expr '{' Statements '}'

DisplayStmt : 'Disp' Expr
```

From this grammar we can see that a program is one or more sequential statements. Our possible statements are either a `RepeatStmt`, a `DisplayStmt` or a `AsgnStmt`. In order to parse a `fimpl` program we simply need to repeatedly parse a statement, and we can always tell which statement we have by the first token, assuming we always remove all the tokens we parse when constructing a statement.

The semantics of these statements are as follows:

- An `AsgnStmt` first evaluates the given expression and then assigns the identifier to the result of the expression.
- A `DisplayStmt` first evaluates the given expression and then prints that result to the screen followed by a newline.
- A `RepeatStmt` first evaluates the given expression, if the expression evaluates to an integer greater than 0 then each of the statements enclosed in the curly braces are executed a number of times equal to the evaluation of the expression. If the expression evaluated to 0 or to a negative number then the enclosed statements are not executed.
  - **Note:** the expression is only evaluated *once* before beginning the repetition of the given statements. You do not repeat the evaluation of the expression each time. That is, you evaluate the expression once, if for example it evaluated to five then you would repeat the sequence of statements five times.

# Part Two

This assignment, which is part two of implementing our `fimpl` interpreter, is a series of functions to implement which will be used together to complete our fimpl interpreter. Unlike previous assignments where each question was worth the same weight, questions in this assignment will be worth varying weight which is listed with each question.

**Note:** Since this is part two of building our `fimpl` interpreter all of your work from part one will also be necessary here. You should begin this assignment by copying all your code from part one into the starter code file provided for part two.

1. **(20%)** In order to execute a `fimpl` program we must keep track of the state of the running `fimpl` program. For this assignment we will not implement memory as part of our state nor input as our `fimpl` language has no way to read in input. Additionally, while we could represent output as a string that we repeatedly grow we will do what is typical of an interpreter and instead simply mutate output directly by printing messages to the screen when the `fimpl` program executes a `Disp` statement. That leaves the need to model the *definitions* of our `fimpl` program, which is a mapping between identifiers and the values they represent since we are omitting memory.

   We will represent our definitions as a `LList` of `LL(str, int)` since all of our `fimpl` values are integers, and our identifiers are all strings. In this way our definitions is represented as a `LList` of pairs, where in each pair the first item is an identifier and the second item is the value that identifier corresponds to.

   For this question write the function `replaceOrAdd` which takes three parameters, the first is the existing definitions `LList`, the second parameter `key` is a string which represents an identifier, and the third parameter `val` is an integer which represents the value that identifier should be bound to.

   Your function should return a `LList of LL(str, int)` which is the given `LList` with one of two changes made to it:

   - If `key` exists in one of the pairs in the given definitions `LList` then the pair in the `LList` has the second item replaced with the given `val` parameter.
   - if `key` does not exist in any of the pairs in the given definitions `LList` then the pair `LL(key, val)` is added to the end of the definitions `LList` in the return value.

   For example:

   ```
   replaceOrAdd(LL(LL("y", 3)), "x", 5) -> LL(LL("y", 3), LL("x", 5))
   replaceOrAdd(LL(LL("y", 3), LL("x", 5))) -> LL(LL("y", 17), LL("x", 5))
   ```

2. **(30%)** Much as we built `ExprTrees` in part one of building our `fimpl` interpeter, we must now build `StmtTrees` which represent individual statements in our `fimpl` program.

   We will define a `StmtTree` as follows:

   ```
   A StmtTree is one of:
   - LL("assign", ExprTree, ExprTree)
   - LL("display", ExprTree)
   - LL("repeat", ExprTree, LList of StmtTree)
   '''
   ```

Where each possibility represents one type of `fimpl` expression.

- The node type `LL("assign", ExprTree, ExprTree)` represents an assignment statement in our `fimpl` language. The first `ExprTree` represents the identifier which is being assigned to, and the second `ExprTree` represents the expression which is evaluated to produce the value to assign to the identifier. For example:

  ```
  x <- add 3 y
  ```

  When parsed would be represented by the following:

  ```
  ('assign',
            ('ident', 'x'),
            ('binOp', 'add',
                            ('intLit', 3),
                            ('ident', 'y')))
  ```

- The node type `LL("display", ExprTree)` represents a display statement in our `fimpl` language. The `ExprTree` represents the expression whose evaluation should be displayed to the screen. For example:

  ```
  Disp add x sub y 3
  ```

  When parsed would be represented by the following:

  ```
  ('display', ('binOp', 'add',
                             ('ident', 'x'),
                             ('binOp', 'sub',
                                            ('ident', 'y'),
                                            ('intLit', 3))))
  ```

- The node type `LL("repeat", ExprTree, LList of StmtTree)` represents a repeat statement in our `fimpl` language. The `ExprTree` represents the expression whose evaluation specifies how many times to repeat, and the `LList of StmtTree` represents the statements to be repeated. For example:

```
Rpt sub x 1 {
  y <- sub y 1
  Disp y
}
```

When parsed would be represented by the following:

```
('repeat',
      ('binOp', 'sub',
                ('ident', 'x'),
                ('intLit', 1)),
      (('assign', ('ident', 'y'), ('binOp', 'sub', ('ident', 'y'), ('intLit', 1))),
       ('display', ('ident', 'y'))))
```

For this question you must write the function `tokensToStmtTree` which behaves very similarly to your `tokensToExprTree` function from the previous assignment. The function `tokensToStmtTree` takes a single `LList` parameter which is a `LList` of tokens and returns a `LList` which is a pair of values, the first value in the pair is the `StmtTree` which was produced by parsing one complete statement from the parameter (which may involve parsing multiple statements if it is in fact a repeat statement), the second value in the pair is a `LList` of tokens which is the original parameter with all the tokens that were used in construction of this `StmtTree` removed.

The function specification is given below:

```
'''
tokensToStmtTree takes a LList of tokens and builds a StmtTree that represents the
                 first statement seen in tokens. It returns a LL(StmtTree, LList of str)
                 where the first item in the pair is the produced StmtTree and the
                 second item in the pair is the remaining tokens.

tokens  - LList of str, which represents tokens of a valid fimpl program
returns - LL(StmtTree, LList of str)

Examples:
  prog1 = "x <- 5\nDisp x\n x <- add x 1"
  tokensToStmtTree(strToTokenList(prog1))
   -> (('assign', ('ident', 'x'), ('intLit', 5)),
       ('Disp', 'x', 'x', '<-', 'add', 'x', '1'))

  prog2 = "Rpt x {\n y <- sub y 1\nDisp y\n}\nx <- add x 1"
  tokensToStmtTree(strToTokenList(prog2))
    -> (('repeat',
          ('ident', 'x'),
          (('assign', ('ident', 'y'),
                      ('binOp', 'sub',
                                ('ident', 'y'),
                                ('intLit', 1))),
            ('display', ('ident', 'y')))),
        ('x', '<-', 'add', 'x', '1'))

'''
```

**Hint 1:** You will need to use your function `tokensToExprTree` from the previous assignment in order to implement this function.

**Hint 2:** Check the first token you have, you should immediately be able to tell by that token if you have a repeat, display, or assignment statement. From there, you should choose how to process the remaining tokens based on what sequence of tokens may comprise that type of statement.

**Hint 3:** In each case you will need to call your function `tokensToExprTree` as each type of statement includes at least one expression. However, the only time `tokensToStmtTree` will need to recursively call itself is in the case of a repeat statement as it contains one or more statements. Since each of these functions also returns in their return pairs the unparsed tokens you know where to continue your work after each call. When will you know that you are done parsing the statements that comprise a repeat statement?

3. **(10%)** For this question you will write the function `tokensToSTreeList` which takes a single `LList of str` parameter which represents the `LList` of tokens of a valid `fimpl` program and produces the `LList of StmtTree` that represents the `fimpl` program. It should perform this task by simply repeatedly calling `tokensToStmtTree` on remaining tokens until all tokens have been consumed. The function specification is given in the provided code, including examples.

4. **(30%)** Now that we have produced a `LList of StmtTrees` that can be used to represent an entire `fimpl` program it is time to write a function can execute `fimpl` statements.

   For this question you must write the function `runStmt` which takes two parameters:

   - `stmt` — a `StmtTree` that represents the statement to execute
   - `defns` — a `LList of LL(str, int)` that represents the definitions in the given `fimpl` program so far.

   Your function should execute the given `stmt` and produce a new definitions `LList` of the statement modified the programs definitions, and should have the side effect of printing to the screen if the given `stmt` was or included `Disp` statements.

   For some examples:

```
stmt0 = first(tokensToSTreeList(strToTokenList("x <- 3")))
runStmt(stmt1, empty()) -> LL(LL("x", 3))

stmt1 = first(tokensToSTreeList(strToTokenList("x <- add x 1")))
runStmt(stmt1, LL(LL("x", 3))) -> LL(LL("x", 4))

stmt2 = first(tokensToSTreeList(strToTokenList("Disp sub x y")))
runStmt(stmt2, LL(LL("x", 2), LL("y", 7))) -> LL(LL("x", 2), LL("y", 7))
   # Side effect: prints "-5\n" to the screen

stmt3 = first(tokensToSTreeList(strToTokenList("Rpt 5 {\nx <- mul x 2\n}")))
runStmt(stmt3, LL(LL("x", 2))) -> LL(LL("x", 64))
```

5. **(10%)** Now that we can produce a `LList of StmtTrees` to represent a `fimpl` program, and we have a function to execute a single `fimpl` statement, it is time to finally complete our `fimpl` interpreter.

   noindent For this question you must write the function `interp` which takes a string that represents a `fimpl` program and executes that `fimpl` program.

   **Hint:** Executing a `fimpl` program is simply executing each of its statements in order, one after another. Using the functions you've already written it should be very simple to transform a `fimpl` program into a `LList of StmtTree` and then execute each of those statements one after another.

**Deliverables:** All the functions specified in this assignment, along with any helper functions, should be contained within one single Python file named `interp.py` which should be submitted to Canvas. The functions you wrote in part one will certainly be necessary to complete part two, so you should include all the code from your part one as well.