# Exercise 1:
# Propositional Logic Syntax

In this exercise, you are asked to understand the syntax of propositional logic formulae. Specifically, most of the exercise is focused on translating formulae back and forth between representation as a string and as an expression-tree data structure. In the next exercise we will deal with the semantic meaning of such formulae; in this exercise we do not attempt to assign any semantics to such formulae, but only deal with their syntax.

The file `propositions/syntax.py` defines a Python class `Formula` for holding a propositional formula as a data structure. The formula can use the binary operators '&' (*and*) and '|' (*or*), the unary operator '~' (*negation*), the constants 'T' (*True*) and 'F' (*False*), and named atomic propositions (variables) that by convention consist of a letter in 'p'...'z' that is possibly followed by a non-negative integer. The exact syntactic rules specifying a formula are:

**Definition.** The following strings are valid propositional formulae:

- 'T'
- 'F'
- A letter in 'p'...'z' optionally followed by a non-negative integer
- '~$\phi$', where $\phi$ is a valid propositional formula
- '($\phi$ | $\psi$)' where each of $\phi$ and $\psi$ is a valid propositional formula
- '($\phi$&$\psi$)' where each of $\phi$ and $\psi$ is a valid propositional formula

These are the only valid propositional formulae.

Notice that the above definition is very specific about the use of parentheses: '($\phi$&$\psi$)' is a valid formula, but '$\phi$&$\psi$' is not and neither is '(($\phi$&$\psi$))'; '~$\phi$' is a valid formula but '(~$\phi$)' is not, etc. These restrictive choices are made to ensure that there is a unique and easy way to **parse** a formula: to take a string that is a formula and figure out the complete **derivation tree** of how it is decomposed into simpler and simpler formulae according to the derivation rules from the above definition. Such a derivation tree is naturally expressed in a program as a tree data structure, which is what the class `Formula` holds. The constructor of this class (which is already implemented) takes as arguments the components (between one and three) of which the formula is composed, and constructs the composite formula. For instance, to represent the formula '($\phi$&$\psi$)', it will be given the three "components": the operator '&' which will serve as the "root" of the tree, and the two sub-formulae $\phi$ and $\psi$.

**Example.** the data structure for representing '~(p&q7)' is constructed using the following code:

```
f = Formula('~', Formula('&', Formula('p'), Formula('q7')))
```

There are several ways in which such a tree data structure can be printed or represented as a string. The "regular" one is the usual **infix** notation in which binary operators are printed between their two operands, as in the above definition.

**Task 1.** Implement the missing code for the method `infix()` (of class `Formula`), which returns a string that represents the formula in infix notation (the notation described above). Thus, for example, for the formula `f` defined in the example above, `f.infix()` should return the string `'~(p&q7)'`. (We have also defined the `__repr__()` method of `Formula` to return the output of `infix()` so that the infix notation will serve as the default string representation of a `Formula` object.)

**Remark.** In Task 1, as well as in all other tasks in all exercises in this course, you may always assume that the input is legal. For example, in Task 1 you may assume that the variables of the `Formula` object on which the function operates are as created by the class constructor, and in Task 2 you may assume that the string argument `s` is a valid infix representation of a propositional formula (and has no "syntax errors").

Going in the opposite direction, i.e., taking a string that represents a formula (in infix notation) and **parsing** it into the appropriate data structure is usually a bit more difficult, since you need to algorithmically figure out where to "break" the complex formula into its different components. This type of parsing challenge is quite common when dealing with many cases of "languages" that have recursive so-called "context free" definitions, for example when compilers need to understand programs written in a programming language. While there is a general theory that deals with how to parse such languages, as well as specialized tools for such tasks, the language for valid formulae that we chose for this course is simple enough so that you can figure out how to surmount the parsing challenge yourself.

**Task 2.** Implement the missing code for the function `from_infix(s)` (a static method of class `Formula`), which takes a string that represents the formula in infix notation, and returns a corresponding `Formula` object. In particular, for every string `s` that represents a valid propositional formula in infix notation, we should have `Formula.from_infix(s).infix() == s`.

While infix notation is the "usual" way to represent formulae, there are other ways as well, in particular the **prefix** and **postfix** notations.[1] In the former, the operator is printed *before* the (two, in the case of a binary operator) sub-formulae that it operates on, and in the latter it is printed after them. Of course, these sub-formulae themselves are recursively printed in the same way. One nice advantage of these formats is that it turns out that parentheses are no longer needed.

**Task 3.** Implement the missing code for the method `prefix()` (of class `Formula`), which returns a string that represents the formula in prefix notation. Thus, for example, for the formula `f` defined in the example above, `f.prefix()` should return the string `'~&pq7'`. (Remember that there are no parentheses in prefix notation.)

[1] Prefix notation is sometimes called **Polish notation** after the Polish logician Jan Łukasiewicz who invented it.

Parsing prefix notation is usually a bit easier that parsing infix notation, even though there are no parentheses.

**Task 4.** Implement the missing code for the function `from_prefix(s)` (a static method of class `Formula`), which takes a string that represents the formula in prefix notation, and returns a corresponding `Formula` object. In particular, for every string `s` that represents a valid propositional formula in prefix notation, we should have `Formula.from_prefix(s).prefix() == s`.

The final task for this exercise asks for the set of all the "variables," i.e. atomic propositions, that appear in a given formula.

**Task 5.** Implement the missing code for the method `variables()` (of class `Formula`), which returns the Python `set` of all named atomic propositions in the formula. Thus, for example, for the formula `f` defined in the example above, `f.variables()` should return `{'p', 'q7'}`.