## Get up and running

The dependencies for this project are matplotlib, network, and pygraphviz.

These should all be available on mds computers.

Alternatively, to install pygraphviz, install brew from <https://brew.sh/> on your Linux/macOS machine and install pygraphviz by typing brew install graphviz

To run the parser first

1. cd into the directory containing parser.py
2. type python parser.py <path to input file>
   1. e.g. python parser.py ./example.txt

## What to expect

There will be two outputs into the current directory both time-stamped in their name when created to avoid writing over old files:

1. A log file: “parse\_log Fri Mar 6 04/28/50 2020.log”
   * A complete pre order traversal of the syntax tree
     + Or an incomplete one in the case of encountering an error
     + Note: no traversal will be given if the formula is found to be invalid before attempting to build the tree.
   * A list of error messages, shown recursively similar to how python gives error messages.A close up of text on a white background

     Description automatically generated
2. A PNG file: “parse\_tree Fri Mar 6 04/28/50 2020.png”
   * Showing the parse tree for the formula, if it is valid.
   * And the original formula at the top.
3. A txt file: “grammar Fri Mar 6 04/28/51 2020.txt”, containing the grammar of this specific input file.

## 3. Parser.py file structure

The project consists of 2 classes, formula and userDefinedSyntax. 2 functions, log\_error and formula\_to\_graph.

* userDefinedSyntax parses the desired input file,
  + finds any errors in variable/constant/predicate declarations explained further in section 4.
* Formula is a class with multiple formula objects as its ‘children’
  + it is essentially a recursive function in so far as the constructor,
  + each child object of the root formula object has a type and a value and optionally a child(ren).
  + but obviously not for individual atoms (in FOL it would be either a constant or a variable).
* log\_error is a two-line function,
  + its sole purpose is to simplify the codebase to make logging errors more understandable.
* formula\_to\_graph takes an instance of the formula class and recursively expands on the object’s children, adding the relevant nodes to the networkX graph declared globally.

## 4. Error checking

The parser should be able to parse any of your favourite FO formulae, I will quickly outline the kind of error checking the parser is doing:

* While parsing the input file
  + finds any errors in variable/constant/predicate declarations,
  + such as having non-alphanumeric/underscores names,
  + or a predicate declared with a non-numeric arity,
  + or if there are any duplicate name declarations in the file.
  + There are the suitable number of logical connectives
* While parsing the formula
  + Logical connectives have been negated
  + Predicates used in a formula that have not been declared in the syntax
  + Unexpected number of arguments given to a predicate.
  + Constants being used as arguments for predicates.
  + Attempting to quantify a constant
* Building the tree
  + Any uncaught errors in my understanding of the grammar of first order logic, or otherwise in the syntax or formula. This is clear as any node that is not a term (constant or variable) must have at least one child. The error log will reflect on the exact formula sub formula of the original input where the error was encountered.

## 5. Notes

* I was having a lot of problems with encoding backslashes in the python file with regex and otherwise, for this reason I strip all predicates, variable names qualifiers etc. of it. It was extremely unstable since it more often than not failed in string comparisons of two identical strings.
* I use pygraphviz to give the tree the hierarchical structure it has since the fixed positions I gave by doing an in-order traversal of the graph occasionally had overlapping edges when the plot was made on a low-resolution machine, where the root node had any more than 40 children
* At a few points I reference the different ‘cases’ of the formula given to the constructor

# case 1: ¬formula  
# case 2: primative\_formula  
# case 3: quantifier variable formula  
# case 4: Term  
# case 5: formula connective formula

* These are outlined more in the grammar file, but for clarities sake:
  + A term is a variable or constant. A primitive formula is predicate function that takes any sequence of terms as a parameter (Constants are excluded later for more meaningful error messages; the length of the sequence is also checked).