COS301
University of Maine
Spring 2022

Homework Assignment #3
Assignment Due: Friday, April 8, 2022, by 11:59pm

Grading: *To avoid surprises, refer to the syllabus about the limited possibilities to submit corrections.*

Silent Policy: *A silent policy will take effect at 6pm on the due date. This means that no questions about this assignment will be answered after this time before the submission deadline, not matter how or where it is asked. Questions will be answered after the submission deadline.*

Late Policy: A 'grace extension' system is in effect, see the course syllabus for details. Because this assignment is in pairs, both team members will use up one extension if you need to. If you have a documented medical condition, please inform your instructor **before** the assignment is due.

How to Submit: Submit your solutions electronically on Brightspace by the deadline as a single **zip file** that contains ONE **py file** and 4 **txt files: the four test files your create** for Question 3 and 2 **txt files** containing the output of the lexer and parser for the test files we provided files. Make sure the file is not corrupted and that you do NOT include any other files. In case we cannot open the file, your assignment only counts as submitted at the time we receive a workable file.

For Questions:
Questions concerning the assignment should be directed to the TA, Sepideh Neshatfar, or the instructor either in person (during office hours) or by email (sepideh.neshatfar@maine.edu, torsten.hahmann@maine.edu). Please place "[" + COS301++ "]" and "A3" in your email subject header.
Important corrections and clarifications to the assignment questions will be posted as announcements on Brightspace. You are responsible for monitoring Brightspace.

Policy on Group Work:
This assignment is specifically designed for you to work in pairs. While you may chose to do it alone, we do not recommend it. You should discuss clearly with your partner how you plan to work together. But at the end, both of you are responsible for the entire assignment.
You may discuss questions and approaches with others but all submitted work must be substantially your own group's work. See the course information sheet for more details of what kind of collaboration is acceptable. If in doubt, talk to the instructor. **Be sure to acknowledge all sources of help!** Refer to the course syllabus for more detailed instructions and the course policy on academic honesty.

For this assignment, you're tasked to construct a lexer and parser for a small subset of the CLIF dialect of the Common Logic standard. Before you start looking: currently no full-fledged parser exists for CLIF. Common Logic is a computational language used to write "ontologies", which encode and allow exchangeing knowledge (not just data but also rules) between programs. In order to facilitate that exchange, CLIF files need to be parsed by programs to translate them to other formats and further process them.

```
char = digit | '~' | '!' | '#' | '$' | '%' | '^' | '&' | '*' | '_' | '+' | '{' | '}'
| '|' | ':' | '<' | '>' | '?' | '`' | '-' | '=' | '[' | ']' | ';'| ',' | '.' | '/'
| 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N'
| 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z'
| 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n'
| 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z'

digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

open = '('
close = ')'

white = U+0020 | U+0009 | U+000A | U+000B | U+000C | U+000D
# white encodes any whitespace or other break (newline, pagebreak, carriage return, etc.)
# don't worry about whitespaces; they are already taken care of in the provided lexer

numeral = digit { digit }

reservedelement = 'and' | 'or' | 'iff' | 'if' | 'not' | 'cl:comment'

stringquote = '''
namequote= '"'
quotedstring = stringquote { char | namequote } stringquote

lexicaltoken = open | close | quotedstring | reservedelement
```

This first part defines the lexical tokens (any of the strings that match one of the `lexicaltokens`. For practical reasons, you may want to treat `stringquote` as a separate token.

The remaining grammar rules are provided below; they describe how to construct CLIF sentences from the lexical tokens.

```
interpretedname = numeral | quotedstring
predicate = interpretedname
termseq = { interpretedname }
sentence = atomsent | boolsent
atomsent = open predicate termseq close
boolsent = ( open ('and' | 'or') { sentence } close )
         | ( open ('if' | 'iff')  sentence  sentence close )
         | ( open 'not' sentence close )
```

A CLIF file is then simply a sequence of sentences.

```
cliffile = { sentence }
```

You can earn up to 15 bonus points if you also successfully parse files that use the following extended grammar rules (two of the rules below replace the rules for those nonterminals from above):

```
innerstringquote = '\''

quotedstring = stringquote { char | namequote | innerstringquote } stringquote
# this replaces the prior definition of quotedstring

sentence = atomsent | boolsent | commentsent
# this replaces the prior definition of sentence

commentsent = open 'cl:comment' quotedstring  sentence  close
```

But make sure that you don't break anything else in your attempt to cover these extended rules, otherwise you risk loosing points in other places.

Important: Your python code must be in a single .py file. All functionality needs to be accessible via a `__main__` method that takes two arguments as input (use argparse to read the command-line input):

1. the first argument is the relative path to a CLIF file. For example, it should accept "..
text1.clif" to read a CLIF file located in the directory above the python file;
2. the second argument is a Boolean (`True` or `False`) that indicates whether to run the lexer only (`False`) or the lexer and parser (`True`). If only lexing is enabled, it should write as output the sequence of tokens one line at a time or produce an easily comprehensible error message. If parsing is enabled, the parser should produce the sequence of parsed elements consisting of type and value; use the type names from the grammar (e.g. `atomsent` or `quotedstring`). This sequence of parsed elements should be followed by the output described in the analysis portion of the parser below.

Question 1. (30 points) Lexing

To construct the lexer and parser, you will be using Python's PLY library (`http://www.dabeaz.com/ply/`). You should familiarize yourself with the documentation available at `http://www.dabeaz.com/ply/ply.html` and go through a simple tutorial like `https://www.skenz.it/compilers/ply`. You'll also need to familiarize yourself with Python's re library (`https://docs.python.org/3/library/re.html`) for regular expression as you will need it to specify lexical tokens.

(a) Test the provided lexer to make sure you can parse the sample sentences provided in there (it does not read them from file yet; instead the examples are hard-coded). We assume that all whitespace characters automatically separate.

(b) (20 points) Extend the lexer to recognize all tokens in the grammar. Do not remove or rename any tokens from the provided lexer but rather extend their definitions and add new ones as necessary. But sure to follow the correct naming scheme as the names of tokens and methods (if you use them) must match.

(c) (10 points) Test your lexer using the provided test files and report the results (all in a single textfile).

Question 2. (40 points) Parsing

(a) (20 points) Next, you will specify the parsing rules by writing suitable Python methods. For this part, the naming of the methods and the specification of the rules as comments are crucially important: they need to conform to the nonterminals and lexical token. The first goal of parsing is to ensure that a CLIF file adheres to the grammar above and that syntactical errors are reported (you only need to worry about the first error).

(b) (10 points) As a second step, extend the parser with a basic analysis of the CLIF input. We want to

   (a) count the overall number of separate sentences (atomic and Boolean sentences, but ignoring nested sentences);

   (b) measure each sentence's complexity: the number of Boolean operations they contain (the number of occurrences, not how many different ones are in there), and the number of distinct interpreted names (not counting numerals).

   For this part, you need to extend the parsing methods with appropriate parameters and record-keeping. The output should look as follows:

```
2 sentences
atomic: ('FuncA' 'a' 100): ops=0, names=2
Boolean: (and ('B' 'C') (or ('C' 'D')))): ops=2, names=3
Boolean: (or ('FuncB' ('Func' 100 'A')) 'Func1'): ops=1, names=4
```

   as an example where the input file contains the following three lines:

```
('FuncA' 'a' 100)
(and ('B' 'C') (or ('C' 'D'))))
(or ('FuncB' ('Func' 100 'A')) 'Func1')
```

   For commented sentences (`commentsent` you should similarly produce `comment:   ...`)

(c) (10 points) Test your parser using the provided files and report the results (all in one textfile).

Question 3. (20 points) Testing

(a) (20 points) Write additional input files (at least 4: two being valid, two non-valid but containing non-obvious errors) that are substantially different from the provided test files to increase the portion of the grammar that the tests cover overall. In other words, make sure to include constructs that are allowed or disallowed but not used in the provided files. You should use these files to further test your lexer and parser.

Have fun and good luck!