XCS221 Assignment 7 — Logic (Extra Credit)

Due Sunday, June 26 at 11:59pm PT.

Guidelines

1. If you have a question about this homework, we encourage you to post your question on our Slack channel, at http://xcs221-scpd.slack.com/
2. Familiarize yourself with the collaboration and honor code policy before starting work.
3. For the coding problems, you must use the packages specified in the provided environment description. Since the autograder uses this environment, we will not be able to grade any submissions which import unexpected libraries.

Submission Instructions

Written Submission: Some questions in this assignment require a written response. For these questions, you should submit a PDF with your solutions online in the online student portal. As long as the PDF is legible and organized, the course staff has no preference between a handwritten and a typeset LaTeX submission. If you wish to typeset your submission and are new to LaTeX, you can get started with the following:

- Type responses only in `submission.tex`.
- Submit the compiled PDF, **not** `submission.tex`.
- Use the commented instructions within the `Makefile` and `README.md` to get started.

Coding Submission: Some questions in this assignment require a coding response. For these questions, you should submit only the `src/submission.py` file in the online student portal. For further details, see Writing Code and Running the Autograder below.

Honor code

We strongly encourage students to form study groups. Students may discuss and work on homework problems in groups. However, each student must write down the solutions independently, and without referring to written notes from the joint session. In other words, each student must understand the solution well enough in order to reconstruct it by him/herself. In addition, each student should write on the problem set the set of people with whom s/he collaborated. Further, because we occasionally reuse problem set questions from previous years, we expect students not to copy, refer to, or look at the solutions in preparing their answers. It is an honor code violation to intentionally refer to a previous year's solutions. More information regarding the Stanford honor code can be foudn at https://communitystandards.stanford.edu/policies-and-guidance/honor-code.

Writing Code and Running the Autograder

All your code should be entered into `src/submission.py`. When editing `src/submission.py`, please only make changes between the lines containing `### START_CODE_HERE ###` and `### END_CODE_HERE ###`. Do not make changes to files other than `src/submission.py`.

The unit tests in `src/grader.py` (the autograder) will be used to verify a correct submission. Run the autograder locally using the following terminal command within the `src/` subdirectory:

```
$ python grader.py
```

There are two types of unit tests used by the autograder:

- `basic`: These tests are provided to make sure that your inputs and outputs are on the right track, and that the hidden evaluation tests will be able to execute.

- `hidden`: These unit tests are the evaluated elements of the assignment, and run your code with more complex inputs and corner cases. Just because your code passed the basic local tests does not necessarily mean that they will pass all of the hidden tests. These evaluative hidden tests will be run when you submit your code to the Gradescope autograder via the online student portal, and will provide feedback on how many points you have earned.

For debugging purposes, you can run a single unit test locally. For example, you can run the test case `3a-0-basic` using the following terminal command within the `src/` subdirectory:

```
$ python grader.py 3a-0-basic
```

Before beginning this course, please walk through the Anaconda Setup for XCS Courses to familiarize yourself with the coding environment. Use the env defined in `src/environment.yml` to run your code. This is the same environment used by the online autograder.

Test Cases

The autograder is a thin wrapper over the python `unittest` framework. It can be run either locally (on your computer) or remotely (on SCPD servers). The following description demonstrates what test results will look like for both local and remote execution. For the sake of example, we will consider two generic tests: `1a-0-basic` and `1a-1-hidden`.

Local Execution - Hidden Tests

All hidden tests rely on files that are not provided to students. Therefore, the tests can only be run remotely. When a hidden test like `1a-1-hidden` is executed locally, it will produce the following result:

```
----- START 1a-1-hidden:  Test multiple instances of the same word in a sentence.
----- END 1a-1-hidden [took 0:00:00.011989 (max allowed 1 seconds), ???/3 points] (hidden test ungraded)
```

Local Execution - Basic Tests

When a basic test like `1a-0-basic` passes locally, the autograder will indicate success:

```
----- START 1a-0-basic:  Basic test case.
----- END 1a-0-basic [took 0:00:00.000062 (max allowed 1 seconds), 2/2 points]
```

When a basic test like `1a-0-basic` fails locally, the error is printed to the terminal, along with a stack trace indicating where the error occurred:

```
----- START 1a-0-basic:  Basic test case.
<class 'AssertionError'>
{'a': 2, 'b': 1} != None
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/grader.py", line 23, in test_0
    submission.extractWordFeatures("a b a"))
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
----- END 1a-0-basic [took 0:00:00.003809 (max allowed 1 seconds), 0/2 points]
```


Remote Execution

Basic and hidden tests are treated the same by the remote autograder. Here are screenshots of failed basic and hidden tests. Notice that the same information (error and stack trace) is provided as the in local autograder, now for both basic and hidden tests.

1a-0-basic) Basic test case. (0.0/2.0)

```
<class 'AssertionError'>:  {'a': 2, 'b': 1} != None
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/autograder/source/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/grader.py", line 23, in test_0
    submission.extractWordFeatures("a b a"))
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
```

Just like in the local autograder, this error caused the test to fail.

Just like in the local autograder, start your debugging in line 23 of grader.py.

1a-1-hidden) Test multiple instances of the same word in a sentence. (0.0/3.0)

```
<class 'AssertionError'>:  {'a': 23, 'ab': 22, 'aa': 24, 'c': 16, 'b': 15} != None
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/autograder/source/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/grader.py", line 31, in test_1
    self.compare_with_solution_or_wait(submission, 'extractWordFeatures', lambda f: f(sentence))
  File "/autograder/source/graderUtil.py", line 183, in compare_with_solution_or_wait
    self.assertEqual(ans1, ans2)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
```

This error caused the test to fail.

Start your debugging in line 31 of grader.py.

Finally, here is what it looks like when basic and hidden tests pass in the remote autograder.

1a-0-basic) Basic test case. (2.0/2.0)

1a-1-hidden) Test multiple instances of the same word in a sentence. (3.0/3.0)

Introduction

In this assignment, you will get some hands-on experience with logic. You'll see how logic can be used to represent the meaning of natural language sentences, and how it can be used to solve puzzles and prove theorems. Most of this assignment will be translating English into logical formulas, but in Problem 4, you will delve into the mechanics of logical inference.

To get started, launch a Python shell and try typing the following commands to add logical expressions into the knowledge base.

```
(XCS221) $ python
Python 3.6.9
Type "help", "copyright", "credits" or "license" for more information.
>>> from logic import *
>>> Rain = Atom('Rain')           # Shortcut
>>> Wet = Atom('Wet')             # Shortcut
>>> kb = createResolutionKB()     # Create the knowledge base
>>> kb.ask(Wet)
I don't know.
>>> kb.ask(Not(Wet))
I don't know.
>>> kb.tell(Implies(Rain, Wet))
I learned something.
>>> kb.ask(Wet)
I don't know.
>>> kb.tell(Rain)
I learned something.
>>> kb.tell(Wet)
I already knew that.
>>> kb.ask(Wet)
Yes.
>>> kb.ask(Not(Wet))
No.
>>> kb.tell(Not(Wet))
I don't buy that.
```

To print out the contents of the knowledge base, you can call `kb.dump()`. For the example above, you get:

```
==== Knowledge base [3 derivations] ===
* Or(Not(Rain),Wet)
* Rain
- Wet
```

In the output, '*' means the fact was explicitly added by the user, and '-' means that it was inferred.

Here is a table that describes how logical formulas are represented in code. Use it as a reference guide:

Name	**Mathematical Notation**	**Code**
Constant symbol	stanford	`Constant('stanford')` (must be lowercase)
Variable symbol	x	`Variable('$x')` (must be lowercase)
Atomic formula (atom)	Rain	`Atom('Rain')` (predicate must be uppercase)
	LocatedIn(stanford, x)	`Atom('LocatedIn','stanford','$x')` (arguments are symbols)
Negation	¬Rain	`Not(Atom('Rain'))`
Conjunction	Rain ∧ Snow	`And(Atom('Rain'), Atom('Snow'))`
Disjunction	Rain ∨ Snow	`Or(Atom('Rain'), Atom('Snow'))`
Implication	Rain → Wet	`Implies(Atom('Rain'), Atom('Wet'))`
Equivalence	Rain ↔ Wet (syntactic sugar for: Rain → Wet ∧ Wet → Rain)	`Equiv(Atom('Rain'), Atom('Wet'))`
Existential quantification	$\exists x$.LocatedIn(stanford, x)	`Exists('$x', Atom('LocatedIn', 'stanford', '$x'))`
Universal quantification	$\forall x$.MadeOfAtoms(x)	`Forall('$x', Atom('MadeOfAtoms', '$x'))`

The operations `And` and `Or` only take two arguments. If we want to take a conjunction or disjunction of more than two, use `AndList` and `OrList`. For example: `AndList([Atom('A'), Atom('B'), Atom('C')])` is equivalent to `And(And(Atom('A'), Atom('B')), Atom('C'))`.

1. **Propositional Logic**

Write a propositional logic formula for each of the following English sentences in the given function in `submission.py`. For example, if the sentence is *"If it is raining, it is wet"*, then you would write `Implies(Atom('Rain'), Atom('Wet'))`, which would be Rain $\rightarrow$ Wet in symbols (see `examples.py`).

Note: Don't forget to return the constructed formula!

(a) **[1 point (Coding, Extra Credit)]** *"If it's summer and we're in California, then it doesn't rain."*

(b) **[2 points (Coding, Extra Credit)]** *"It's wet if and only if it is raining or the sprinklers are on."*

(c) **[2 points (Coding, Extra Credit)]** *"Either it's day or night (but not both)."*

You can run the following command to test each formula:

```
python grader.py 1a-0-basic
```

If your formula is wrong, then the grader will provide a counterexample, which is a model that your formula and the correct formula don't agree on. For example, if you accidentally wrote `And(Atom('Rain'), Atom('Wet'))` for *"If it is raining, it is wet"*, then the grader would output the following:

```
Your formula (And(Rain,Wet)) says the following model is FALSE, but it should be TRUE:
* Rain = False
* Wet = True
* (other atoms if any) = False
```

In this model, it is not raining and it is wet, which satisfies the correct formula Rain $\rightarrow$ Wet (`TRUE`), but does not satisfy the incorrect formula Rain $\wedge$ Wet (`FALSE`). Use these counterexamples to guide you in the rest of the assignment.

2. **First Order Logic**

Write a first-order logic formula for each of the following English sentences in the given function in `submission.py`. For example, if the sentence is *"There is a light that shines"*, then you would write `Exists('$x', And(Atom('Light', '$x'), Atom('Shines', '$x')))`, which would be $\exists x.\text{Light}(x) \wedge \text{Shines}(x)$ in symbols (see `examples.py`).

Tip: Python tuples can span multiple lines, which help with readability when you are writing logic expressions (some of them in this homework can get quite large)

(a) **[0.50 points (Coding, Extra Credit)]** *"Every person has a mother."*

(b) **[0.50 points (Coding, Extra Credit)]** *"At least one person has no children."*

(c) **[0.50 points (Coding, Extra Credit)]** Create a formula which defines `Daughter(x,y)` in terms of `Female(x)` and `Child(x,y)`.

(d) **[0.50 points (Coding, Extra Credit)]** Create a formula which defines `Grandmother(x,y)` in terms of `Female(x)` and `Parent(x,y)`.

3. **Liar Puzzle**

Someone crashed the server, and accusations are flying. For this problem, we will encode the evidence in first-order logic formulas to find out who crashed the server. You've narrowed it down to four suspects: John, Susan, Mark, and Nicole. You have the following information:

- John says: "It wasn't me!"
- Susan says: "It was Nicole!"
- Mark says: "No, it was Susan!"
- Nicole says: "Susan's a liar."
- You know that exactly one person is telling the truth.
- You also know exactly one person crashed the server.

(a) **[2 points (Coding, Extra Credit)]** Fill out `liar()` to return a list of 6 formulas, one for each of the above facts. Be sure your formulas are exactly in the order specified.

You can test your code using the following commands:

```
python grader.py 3a-0-basic
python grader.py 3a-1-basic
python grader.py 3a-2-basic
python grader.py 3a-3-basic
python grader.py 3a-4-basic
python grader.py 3a-5-basic
python grader.py 3a-all-basic  # Tests the conjunction of all the formulas
```

To solve the puzzle and find the answer, `tell` the formulas to the knowledge base and `ask` the query `CrashedServer('$x')`, by running:

```
python grader.py 3a-run-basic
```

4. **Logical Inference**

Having obtained some intuition on how to construct formulas, we will now perform logical inference to derive new formulas from old ones. Recall that:

- Modus ponens asserts that if we have two formulas, $A \to B$ and A in our knowledge base, then we can derive B.
- Resolution asserts that if we have two formulas, $A \vee B$ and $\neg B \vee C$ in our knowledge base, then we can derive $A \vee C$.
- If $A \wedge B$ is in the knowledge base, then we can derive both A and B.

(a) **[1 point (Written, Extra Credit)]** Some inferences that might look like they're outside the scope of Modus ponens are actually within reach. Suppose the knowledge base contains the following two formulas:

$$\text{KB} = \{(A \vee B) \to C, A\}.$$

Your task:

First, convert the knowledge base into conjunctive normal form (CNF). Then apply Modus ponens to derive C. Please show how your knowledge base changes as you apply derivation rules.

Hint: You may use the fact that $P \to Q$ is equivalent $\neg P \vee Q$.

Remember, this isn't about you as a human being able to arrive at the conclusion, but rather about the rote application of a small set of transformations (which a computer could execute).

(b) **[1 point (Written, Extra Credit)]** Recall that Modus ponens is not complete, meaning that we can't use it to derive everything that's true. Suppose the knowledge base contains the following formulas:

$$\text{KB} = \{A \vee B, B \to C, (A \vee C) \to D\}.$$

In this example, Modus ponens cannot be used to derive D, even though D is entailed by the knowledge base. However, recall that the resolution rule is complete.

Your task: Convert the knowledge base into CNF and apply the resolution rule repeatedly to derive D.

5. **Odd and Even Integers**

In this problem, we will see how to use logic to automatically prove mathematical theorems. We will focus on encoding the theorem and leave the proving part to the logical inference algorithm. Here is the theorem:

If the following constraints hold:

1. Each number x has exactly one successor, which is not equal to x.
2. Each number is either odd or even, but not both.
3. The successor of an even number is odd.
4. The successor of an odd number is even.
5. For every number x, the successor of x is larger than x.
6. Larger is a transitive property: if x is larger than y and y is larger than z, then x is larger than z.

Then we have the following consequence:

- For each number, there is an even number larger than it.

Note: in this problem, "larger than" is just an arbitrary relation, and you should not assume it has any prior meaning. In other words, don't assume things like "a number can't be larger than itself" unless explicitly stated.

(a) **[2 points (Coding, Extra Credit)]** Fill out `ints()` to construct six formulas for each of the constraints. The consequence has been filled out for you (`query` in the code). You can test your code using the following commands:

```
python grader.py 5a-0-basic
python grader.py 5a-1-basic
python grader.py 5a-2-basic
python grader.py 5a-3-basic
python grader.py 5a-4-basic
python grader.py 5a-5-basic
python grader.py 5a-all-basic  # Tests the conjunction of all the formulas
```

To finally prove the theorem, `tell` the formulas to the knowledge base and `ask` the query by running model checking (on a finite model):

```
python grader.py 5a-run
```

(b) **[1 point (Written, Extra Credit)]** Suppose we added another constraint:

7. A number is not larger than itself.

Prove that there is no finite, non-empty model for which the resulting set of seven constraints is consistent.

Hint: Consider a finite, non-empty model, assume that all the constraints can be satisfied, then prove by contradiction.

6. **Semantic Parsing**

Semantic parsing is the task of converting natural lanugage utterances into first-order logic formulas. We have created a small set of grammar rules in the code for you in `createBaseEnglishGrammar()`. In this problem, you will add additional grammar rules to handle a wider variety of sentences. Specifically, create a `GrammarRule` for each of the following sentence structures.

(a) **[2 points (Coding, Extra Credit)]** Example: *Every person likes some cat.*

General template:

```
$Clause ← every $Noun $Verb some $Noun
```

(b) **[2 points (Coding, Extra Credit)]** Example: *There is some cat that every person likes.*

General template:

```
$Clause ← there is some $Noun that every $Noun $Verb
```

(c) **[2 points (Coding, Extra Credit)]** Example: *If a person likes a cat then the former feeds the latter.*

General template:

```
$Clause ← if a $Noun $Verb a $Noun then the former $Verb the latter
```

After implementing these functions, you should be able to try some simple queries using `nli.py`! For example:

```
(XCS221) $ python nli.py
============================================================
Hello!  Talk to me in English.
Tell me something new (end the sentence with '.') or ask me a question (end the sentence with '?')
    .
Type 'help' for additional commands.
------------------------------
> Every person likes some cat.

>>>>> I learned something.
------------------------------
> Every cat is a mammal.

>>>>> I learned something.
------------------------------
> Every person likes some mammal?

>>>>> Yes.
```

This handout includes space for every question that requires a written response. Please feel free to use it to handwrite your solutions (legibly, please). If you choose to typeset your solutions, the `README.md` for this assignment includes instructions to regenerate this handout with your typeset LaTeX solutions.

4.a

4.b

5.b