CIS 521: Homework 10 [100 points]

Release Date	Tuesday, April 19, 2016
Due Date	11:59 pm on Tuesday, April 26, 2016

Instructions

In this assignment, you will implement a collection of basic algorithms for working with logical expressions, then will apply them to solve textual puzzles expressible using propositional logic.

A skeleton file `homework10.py` containing empty definitions for each question has been provided. Since portions of this assignment will be graded automatically, none of the names or function signatures in this file should be modified. However, you are free to introduce additional variables or functions if needed.

You may import definitions from any standard Python library, and are encouraged to do so in case you find yourself reinventing the wheel.

You will find that in addition to a problem specification, most programming questions also include a pair of examples from the Python interpreter. These are meant to illustrate typical use cases, and should not be taken as comprehensive test suites.

You are strongly encouraged to follow the Python style guidelines set forth in PEP 8, which was written in part by the creator of Python. However, your code will not be graded for style.

Once you have completed the assignment, you should submit your file on Eniac using the following `turnin` command, where the flags `-c` and `-p` stand for "course" and "project", respectively.

```
turnin -c cis521 -p hw10 homework10.py
```

You may submit as many times as you would like before the deadline, but only the last submission will be saved. To view a detailed listing of the contents of your most recent submission, you can use the following command, where the flag `-v` stands for "verbose".

```
turnin -c cis521 -p hw10 -v
```

1. Propositional Logic [65 points]

In this section, you will implement a collection of classes corresponding to logical expressions, a simple knowledge base, and algorithms for propositional logic inference via truth table enumeration and resolution. All expressions will be subclasses of the provided base Expr class. The fragment of logic we will be working with can be defined recursively as follows:

- Atom: Atom(name) is an expression.

- Negation: If e is an expression, then Not(e) is an expression.
- Conjunction: If $e_1, e_2, \ldots, e_n$ are expressions, then And($e_1, e_2, \ldots, e_n$) is an expression.
- Disjunction: If $e_1, e_2, \ldots, e_n$ are expressions, then Or($e_1, e_2, \ldots, e_n$) is an expression.
- Implication: If e_1 and e_2 are expressions, then Implies(e_1, e_2) is an expression.
- Biconditional: If e_1 and e_2 are expressions, then Iff(e_1, e_2) is an expression.

You will find that a generic hash function has been defined for the Expr class, and that the initialization methods for each of its subclasses have been provided. These should not be altered. Combined with the methods for equality checking to be implemented below, this will ensure that expressions behave as expected when used as elements of sets.

1. **[3 points]** First read through and understand the provided `__init__` methods for the `Atom`, `Not`, `And`, `Or`, `Implies`, and `Iff` expression subclasses, keeping in mind that expressions will be considered immutable for our purposes. In particular, review the `*args` syntax (link) and `frozenset` class (link) if needed. The former allows for convenient n-ary conjunctions and disjunctions, and the latter is used to ensure immutability.

   We use sets rather than lists for conjunctions and disjunctions to guarantee that, e.g., all 24 permutations of the conjuncts in the expression $A \wedge B \wedge C \wedge D$ will be equivalent. Moreover, you may find that certain set functions such as union and set difference will prove useful later on.

   As a first exercise, implement the `__eq__(self, other)` methods in each subclass, which will be called when expressions are compared using the `==` operator. You should check for syntactic equality only, in the sense that two expressions should be considered equal only if they are of the same class and have the same internal structure. No simplification should be performed. As a special case, `Iff(a, b)` should be equal to `Iff(b, a)`. *Hint: Each of these can be implemented in a single line.*

```
>>> Atom("a") == Atom("a")
True
>>> Atom("a") == Atom("b")
False
```

```
>>> And(Atom("a"), Not(Atom("b"))) == \
... And(Not(Atom("b")), Atom("a"))
True
```

2. **[2 points]** Implement the `__repr__(self)` method in each expression subclass, which should return a string representation of the given expression. Any reasonable choice will suffice for this exercise, as this is primarily intended for debugging purposes.

```
>>> a, b, c = map(Atom, "abc")
>>> Implies(a, Iff(b, c))
Implies(Atom(a), Iff(Atom(b), Atom(c)))
```

```
>>> a, b, c = map(Atom, "abc")
>>> And(a, Or(Not(b), c))
And(Atom(a), Or(Not(Atom(b)), Atom(c)))
```

3. **[5 points]** Implement the `atom_names(self)` method in each expression subclass, which should return the set of names that occur in atoms contained within the given expression.

```
>>> Atom("a").atom_names()
set(['a'])
>>> Not(Atom("a")).atom_names()
set(['a'])
```

```
>>> a, b, c = map(Atom, "abc")
>>> expr = And(a, Implies(b, Iff(a, c)))
>>> expr.atom_names()
set(['a', 'c', 'b'])
```

4. **[5 points]** Implement the `evaluate(self, assignment)` method in each expression subclass, which should return the truth value of the given formula under the provided assignment from atom names to Boolean values. You may assume that the assignment dictionary contains the necessary entries to fully evaluate the expression.

```
>>> e = Implies(Atom("a"), Atom("b"))
>>> e.evaluate({"a": False, "b": True})
True
>>> e.evaluate({"a": True, "b": False})
False
```

```
>>> a, b, c = map(Atom, "abc")
>>> e = And(Not(a), Or(b, c))
>>> e.evaluate({"a": False, "b": False,
...     "c": True})
True
```

5. **[10 points]** Write a `satisfying_assignments(expr)` function that generates all assignments from atom names to truth values under which the input expression is true. The assignments may be generated in any order, as long as all satisfying assignments are produced.

```
>>> e = Implies(Atom("a"), Atom("b"))
>>> a = satisfying_assignments(e)
>>> next(a)
{'a': False, 'b': False}
>>> next(a)
{'a': False, 'b': True}
>>> next(a)
{'a': True, 'b': True}
```

```
>>> e = Iff(Iff(Atom("a"), Atom("b")),
...     Atom("c"))
>>> list(satisfying_assignments(e))
[{'a': False, 'c': False, 'b': True},
 {'a': False, 'c': True, 'b': False},
 {'a': True, 'c': False, 'b': False},
 {'a': True, 'c': True, 'b': True}]
```

6. **[20 points]** Implement the `to_cnf(self)` method in each expression subclass, which should return an expression in conjunctive normal form that is logically equivalent to the input. Specifically, the output of this method should be a literal (i.e. an atom or a negated atom), a disjunction of literals, or a conjunction consisting of literals and/or disjunctions of literals.

```
>>> Atom("a").to_cnf()
Atom(a)
>>> a, b, c = map(Atom, "abc")
>>> Iff(a, Or(b, c)).to_cnf()
And(Or(Atom(b), Atom(c), Not(Atom(a))),
    Or(Not(Atom(b)), Atom(a)),
    Or(Not(Atom(c)), Atom(a)))
```

```
>>> Or(Atom("a"), Atom("b")).to_cnf()
Or(Atom(b), Atom(a))
>>> a, b, c, d = map(Atom, "abcd")
>>> Or(And(a, b), And(c, d)).to_cnf()
And(Or(Atom(d), Atom(a)),
    Or(Atom(a), Atom(c)),
    Or(Atom(b), Atom(c)),
    Or(Atom(b), Atom(d)))
```

7. **[20 points]** In this question, we will consider a knowledge base to be an object which stores a collection of facts and supports entailment queries based on those facts.

   First write the `__init__(self)` and `get_facts(self)` methods in the `KnowledgeBase` class, which initialize and return an internal fact set, respectively. Next write the `tell(self, expr)` method, which converts the input expression to conjunctive normal form and adds the resulting conjuncts to the internal fact set. Lastly write the `ask(self, expr)` method, which returns a Boolean value indicating whether the facts in the knowledge base entail the input expression. You should determine entailment using the resolution algorithm discussed in class.

```
>>> a, b, c = map(Atom, "abc")
>>> kb = KnowledgeBase()
>>> kb.tell(a)
>>> kb.tell(Implies(a, b))
>>> kb.get_facts()
set([Or(Atom(b), Not(Atom(a))),
    Atom(a)])
>>> [kb.ask(x) for x in (a, b, c)]
[True, True, False]
```

```
>>> a, b, c = map(Atom, "abc")
>>> kb = KnowledgeBase()
>>> kb.tell(Iff(a, Or(b, c)))
>>> kb.tell(Not(a))
>>> [kb.ask(x) for x in (a, Not(a))]
[False, True]
>>> [kb.ask(x) for x in (b, Not(b))]
[False, True]
>>> [kb.ask(x) for x in (c, Not(c))]
[False, True]
```

2. Logic Puzzles [30 points]

In this section, you will encode some simple logic puzzles in propositional logic and use the functions written in the previous section to solve them. You will be graded both on the correctness of your results and on the correctness of your formulations.

1. **[5 points]** Consider the following set of facts. If the unicorn is mythical, then it is immortal, but if it is not mythical, then it is a mortal mammal. If the unicorn is either immortal or a mammal, then it is horned. The unicorn is magical if it is horned.

   From these, can it be inferred that the unicorn is mythical? That it is magical? That it is horned?

   Using the atomic expressions `Atom("mythical")`, `Atom("mortal")`, `Atom("mammal")`, `Atom("horned")`, and `Atom("magical")`, populate the indicated knowledge base with the appropriate facts, and assign the appropriate queries to the indicated variables. Then, use these to answer the above questions, and record your answers in the corresponding variables.

2. **[5 points]** You would like to throw a party subject to the following conditions: John will come if Mary or Ann come, Ann will come if Mary does not come, and if Ann comes, then John will not come.

   Letting the atomic expressions `Atom("a")`, `Atom("j")`, and `Atom("m")` denote that Ann, John, and Mary come, respectively, encode the given conditions as a single conjunction, and use the previously-defined `satisfying_assignments(expr)` function to compute all valid scenarios. In what way(s) can the guests attend without violating the constraints?

3. **[10 points]** A game show contestant is to choose between two rooms, each of which may either contain a prize or be empty. The sign on the first door states: "This room contains a prize, and the other room is empty." The sign on the second door states: "At least one room contains a prize, and at least one room is empty." The contestant is told that exactly one sign is true.

   Let the atomic expression `Atom("p1")` denote that the first room contains a prize, and let the atomic expression `Atom("e1")` denote that the first room is empty. Similarly define `Atom("p2")` and `Atom("e2")` for the second room. Also let the atomic expressions `Atom("s1")` and `Atom("s2")` denote that the first sign and second sign are true, respectively.

   Using these, populate the indicated knowledge base with the appropriate facts. Then make the necessary queries to determine what the contestant can deduce about the contents of each room and the truth of the signs. What is the correct state of affairs?

4. **[10 points]** There are three suspects for a murder: Adams, Brown, and Clark. Adams says "I didn't do it. The victim was old acquaintance of Brown's. But Clark never knew him." Brown states "I didn't do it. I didn't know the guy." Clark says "I didn't do it. I saw both Adams and Brown downtown with the victim that day; one of them must have done it." Assume that the two innocent men are telling the truth, but that the guilty man is not.

   Let the atomic expressions `Atom("ia")`, `Atom("ib")`, and `Atom("ic")` denote that Adams, Brown, and Clark are innocent, respectively, and let `Atom("ka")`, `Atom("kb")`, and `Atom("kc")` denote that Adams, Brown, and Clark knew with the victim. Populate the indicated knowledge base with the given information, and query it as necessary to determine which suspect is guilty. Record your answer and the corresponding query in the provided variables.

3. Feedback [5 points]

1. **[1 point]** Approximately how long did you spend on this assignment?

2. **[2 points]** Which aspects of this assignment did you find most challenging? Were there any significant stumbling blocks?

3. **[2 points]** Which aspects of this assignment did you like? Is there anything you would have changed?