Started on	Wednesday, 9 August 2023, 2:27 PM
State	Finished
Completed on	Friday, 11 August 2023, 2:22 PM
Time taken	1 day 23 hours
Marks	14.00/14.00
Grade	**1.00** out of 1.00 (**100**%)

Question 1

Correct

Mark 1.00 out of 1.00

Consider the following knowledge base.

```
a and b
b implies c
not d or e
```

The symbols 'and', 'or', 'not', and 'implies' are logical connectives with their usual meanings. The binding priorities are also as usual; so the last formula is '(not d) or e'.

How many atoms are there in the knowledge base? 5 ✔

How many distinct interpretation functions are there for this knowledge base? 32 ✔

How many models does this knowledge base have? 3 ✔

How many atoms are true in all the models of the knowledge base (i.e. logical consequences)? 3 ✔

Correct

Marks for this submission: 1.00/1.00.

Question 2

Correct

Mark 3.00 out of 3.00

Write a function `interpretations(atoms)` that takes a non-empty set of atoms and returns the list of all possible interpretations for the given atoms. Each atom is represented as a string. Each interpretation is a dictionary where the keys are all the atoms and each value is either True or False. The keys of the dictionary must be in alphabetical order. The output list must be ordered such that for every two interpretations that are only different in the value of one atom, the one with value of False comes before the other one.

Notes

- Consider using itertools.product. The `repeat` parameter can be useful here.
- The size of the output grows exponentially.
- Dictionaries are ordered by default (since Python 3.7). So as long as you add the entries in alphabetical order, the expected order in this question is automatically achieved.
- In order to achieve the expected order for the output list, pass a list (with two elements in the right order) to `itertools.product`.
- If you are importing any modules, please remember to include the import statement.

For example:

Test	Result

`from student_answer import interpretations`
 	
`atoms = {'q', 'p'}`
`for i in interpretations(atoms):`
`  print(i)`
	`{'p': False, 'q': False}`
	`{'p': False, 'q': True}`
	`{'p': True, 'q': False}`
	`{'p': True, 'q': True}`

`from student_answer import interpretations`
 	
`atoms = {'human', 'mortal', 'rational'}`
`for i in interpretations(atoms):`
`  print(i)`
	`{'human': False, 'mortal': False, 'rational': False}`
	`{'human': False, 'mortal': False, 'rational': True}`
	`{'human': False, 'mortal': True, 'rational': False}`
	`{'human': False, 'mortal': True, 'rational': True}`
	`{'human': True, 'mortal': False, 'rational': False}`
	`{'human': True, 'mortal': False, 'rational': True}`
	`{'human': True, 'mortal': True, 'rational': False}`
	`{'human': True, 'mortal': True, 'rational': True}`

Answer: (penalty regime: 0, 15, ... %)

```
import itertools
def interpretations(atoms):
    """takes a non-empty set of atoms and returns the list of
    all possible interpretations for the given atoms.
    """
    ans = []
    table = {True, False}
    for tup in itertools.product(table, repeat=len(atoms)):
        i=0
        dic = {}
        for atom in sorted(atoms):
            dic[atom] = tup[i]
            i += 1
        ans.append(dic)
    return ans
```

	Test	Expected	Got	

✔	`from student_answer import interpretations`
	
	`atoms = {'q', 'p'}`
	`for i in interpretations(atoms):`
	`  print(i)`
		`{'p': False, 'q': False}`
		`{'p': False, 'q': True}`
		`{'p': True, 'q': False}`
		`{'p': True, 'q': True}`
			`{'p': False, 'q': False}`
			`{'p': False, 'q': True}`
			`{'p': True, 'q': False}`
			`{'p': True, 'q': True}`
				✔

	Test	Expected	Got	

✔	`from student_answer import interpretations`
	
	`atoms = {'human', 'mortal', 'rational'}`
	`for i in interpretations(atoms):`
	`  print(i)`
		`{'human': False, 'mortal': False, 'rational': False}`
		`{'human': False, 'mortal': False, 'rational': True}`
		`{'human': False, 'mortal': True, 'rational': False}`
		`{'human': False, 'mortal': True, 'rational': True}`
		`{'human': True, 'mortal': False, 'rational': False}`
		`{'human': True, 'mortal': False, 'rational': True}`
		`{'human': True, 'mortal': True, 'rational': False}`
		`{'human': True, 'mortal': True, 'rational': True}`
			`{'human': False, 'mortal': False, 'rational': False}`
			`{'human': False, 'mortal': False, 'rational': True}`
			`{'human': False, 'mortal': True, 'rational': False}`
			`{'human': False, 'mortal': True, 'rational': True}`
			`{'human': True, 'mortal': False, 'rational': False}`
			`{'human': True, 'mortal': False, 'rational': True}`
			`{'human': True, 'mortal': True, 'rational': False}`
			`{'human': True, 'mortal': True, 'rational': True}`
				✔

Passed all tests! ✔

Correct

Marks for this submission: 3.00/3.00.

Question 3

Correct

Mark 3.00 out of 3.00

Write a function `models(knowledge_base)` that takes a knowledge base in the form of a non-empty set of logical formulas and returns a (possibly empty) list of interpretations that are the models of the knowledge base. Each logical formula is represented as a lambda expression that evaluates to a boolean value. The parameters of the lambda expression are the atoms used in the formula. For example `lambda a, b: a and b` is a compound proposition that involves atoms `a` and `b` and the formula is the logical conjunction of the two. The keys of the dictionaries and the output list must follow the same order established in the previous question (interpretations).

Notes

- The answer box has been preloaded with two auxiliary functions that will be useful.
- You need the function `interpretations` in order to answer this question. Include your answer from the previous question together with all the necessary import statements.
- You need to first construct a set that contains all the atoms used in the entire knowledge base.
- Consider using the built-in function `all`.
- After answering the question, you should be able to see how your function can be used to derive all the logical consequences of a knowledge base.
- Note that the time complexity of this procedure is exponential. This is the price we pay for being able to have a knowledge containing any type of propositional formula.

For example:

Test	Result

`from student_answer import models`
 	
`knowledge_base = {`
`  lambda a, b: a and not b,`
`  lambda c: c`
`}`
 	
`print(models(knowledge_base))`
	`[{'a': True, 'b': False, 'c': True}]`

`from student_answer import models`
 	
`knowledge_base = {`
`  lambda a, b: a and not b,`
`  lambda c, d: c or d`
`}`
 	
`for interpretation in models(knowledge_base):`
`  print(interpretation)`
	`{'a': True, 'b': False, 'c': False, 'd': True}`
	`{'a': True, 'b': False, 'c': True, 'd': False}`
	`{'a': True, 'b': False, 'c': True, 'd': True}`

Answer: (penalty regime: 0, 15, ... %)

Reset answer

```
import itertools
def interpretations(atoms):
    """takes a non-empty set of atoms and returns the list of
    all possible interpretations for the given atoms.
    """
    ans = []
    table = {True, False}
    for tup in itertools.product(table, repeat=len(atoms)):
        i=0
        dic = {}
        for atom in sorted(atoms):
            dic[atom] = tup[i]
            i += 1
        ans.append(dic)
    return ans

def atoms(formula):
    """Takes a formula in the form of a lambda expression and returns a set of
    atoms used in the formula. The atoms are parameter names represented as
    strings.
    """
```

```
    return {atom for atom in formula.__code__.co_varnames}

def value(formula, interpretation):
    """Takes a formula in the form of a lambda expression and an interpretation
    in the form of a dictionary, and evaluates the formula with the given
    interpretation and returns the result. The interpretation may contain
    more atoms than needed for the single formula.
    """
    arguments = {atom: interpretation[atom] for atom in atoms(formula)}
    return formula(**arguments)

def models(knowledge_base):
    """takes a knowledge base in the form of a non-empty set of logical formula
    and returns a (possibly empty) list of interpretations that are the models
    of the knowledge base.
    """
    ans = []
    atom = set()
    for formula in knowledge_base:
        atom = atom.union(atoms(formula))
    inters = interpretations(atom)

    for inter in inters:
        in_model = True
        for formula in knowledge_base:
            if value(formula, inter) == 0:
                in_model = False
        if in_model:
            ans.append(inter)
```

	Test	Expected	Got	

✔	`from student_answer import models`
	
	`knowledge_base = {`
	`  lambda a, b: a and not b,`
	`  lambda c: c`
	`}`
	
	`print(models(knowledge_base))`
		`[{'a': True, 'b': False, 'c': True}]`	`[{'a': True, 'b': False, 'c': True}]`	✔

Passed all tests! ✔

Correct

Marks for this submission: 3.00/3.00.

Information

Knowledge bases containing only propositional definite clauses

In the remainder of this quiz, we work with knowledge bases that only contain *propositional definite clauses*. While this restricts us in terms of the kind of logical formula that we can put in the knowledge base, it enables us to use more interesting and relatively more efficient algorithms for inference.

The syntax of propositional definite clauses

We follow the following syntactic rules to describe knowledge bases containing propositional definite clauses.

1. A knowledge base is a collection of zero or more propositional definite clauses (PDCs).
2. Atoms must start with a lower case letter.
3. Each PDC must have a head. The head is an atom.
4. A PDC can optionally have a body. If a body is present, the `:-` symbol (which means 'if') follows the head. The body comes after this symbol. The body is one or more atoms conjuncted by `,` symbol. For example `p :- q, r.` means "p if (q and r)" or equivalently "(q and r) implies p".
5. Every PDC must end with a full stop.

Question 4

Correct

Mark 1.00 out of 1.00

Consider the following knowledge base.

```
p :- q.
a :- b.
b.
```

How many distinct interpretation functions are there for this knowledge base? 16 ✔

How many models does this knowledge base have? 3 ✔

In how many models is a true? 3 ✔

In how many models is b true? 3 ✔

In how many models is p true? 2 ✔

In how many models is q true? 1 ✔

Your answer is correct.

You have correctly answered 6 part(s) of this question.

Information

Reading knowledge bases

In the following programming questions, your program will need to read in a knowledge base in the form of a string and perform automatic inference. If you wish you can use the following generator function to read the knowledge base.

```
import re

def clauses(knowledge_base):
    """Takes the string of a knowledge base; returns an iterator for pairs
    of (head, body) for propositional definite clauses in the
    knowledge base. Atoms are returned as strings. The head is an atom
    and the body is a (possibly empty) list of atoms.

    -- Kourosh Neshatian - 2 Aug 2021

    """
    ATOM   = r"[a-z][a-zA-Z\d_]*"
    HEAD   = rf"\s*(?P<HEAD>{ATOM})\s*"
    BODY   = rf"\s*(?P<BODY>{ATOM}\s*(,\s*{ATOM}\s*)*)\s*"
    CLAUSE = rf"{HEAD}(:-{BODY})?\."
    KB     = rf"^({CLAUSE})*\s*$"

    assert re.match(KB, knowledge_base)

    for mo in re.finditer(CLAUSE, knowledge_base):
        yield mo.group('HEAD'), re.findall(ATOM, mo.group('BODY') or "")
```

You can use `list(clauses(a_knowledge_base_str))` to see what it returns and whether it is useful to you.

Please note that the function is *not* provided on the server. If you decide to use this function, include it in your code.

▶ [Note: If you are working with older versions of Python 3 at home you can click here to use the following which works for Python 3.5 and below.

Question 5

Correct

Mark 3.00 out of 3.00

Write a function `forward_deduce` that takes the string of a knowledge base containing propositional definite clauses and returns a (complete) set of atoms (strings) that can be derived (to be true) from the knowledge base.

Note: remember to include the `clauses` generator function (and `import re`) if necessary.

For example:

Test	Result

`from student_answer import forward_deduce`
 	
`kb = """`
`a :- b.`
`b.`
`"""`
 	
`print(", ".join(sorted(forward_deduce(kb))))`
	`a, b`

`from student_answer import forward_deduce`
 	
`kb = """`
`good_programmer :- correct_code.`
`correct_code :- good_programmer.`
`"""`
 	
`print(", ".join(sorted(forward_deduce(kb))))`	

`from student_answer import forward_deduce`
 	
`kb = """`
`a :- b, c.`
`b :- d, e.`
`b :- g, e.`
`c :- e.`
`d.`
`e.`
`f :- a,`
`g.`
`"""`
 	
`print(", ".join(sorted(forward_deduce(kb))))`
	`a, b, c, d, e`

Answer: (penalty regime: 0, 15, ... %)

```
import re
def clauses(knowledge_base):
    """Takes the string of a knowledge base; returns an iterator for pairs
    of (head, body) for propositional definite clauses in the
    knowledge base. Atoms are returned as strings. The head is an atom
    and the body is a (possibly empty) list of atoms.

    -- Kourosh Neshatian - 2 Aug 2021

    """
    ATOM   = r"[a-z][a-zA-Z\d_]*"
    HEAD   = rf"\s*(?P<HEAD>{ATOM})\s*"
    BODY   = rf"\s*(?P<BODY>{ATOM}\s*(,\s*{ATOM}\s*)*)\s*"
    CLAUSE = rf"{HEAD}(:-{BODY})?\."
    KB     = rf"^({CLAUSE})*\s*$"

    assert re.match(KB, knowledge_base)

    for mo in re.finditer(CLAUSE, knowledge_base):
        yield mo.group('HEAD'), re.findall(ATOM, mo.group('BODY') or "")

def forward_deduce(kb):
    """takes the string of a knowledge base containing propositional definite
    clauses and returns a (complete) set of atoms (strings) that can be
    derived (to be true) from the knowledge base.
    return always true atoms from kb model
    """
```

```
    C = set()
    statements = []
    for clause in clauses(kb):
        statements.append(clause)
    to_remove = [1]
    while len(statements) > 0 and to_remove != []:
        to_remove = []
        for statement in statements:
            if set(statement[1]).union(C) == C and (statement[0] not in C):
                to_remove.append(statement)
        for item in to_remove:
            C.add(item[0])
            statements.remove(item)
    return C
```

	Test	Expected	Got	

✔	`from student_answer import forward_deduce`
	
	`kb = """`
	`a :- b.`
	`b.`
	`"""`
	
	`print(", ".join(sorted(forward_deduce(kb))))`
		a, b	a, b	✔

✔	`from student_answer import forward_deduce`
	
	`kb = """`
	`good_programmer :- correct_code.`
	`correct_code :- good_programmer.`
	`"""`
	
	`print(", ".join(sorted(forward_deduce(kb))))`
				✔

✔	`from student_answer import forward_deduce`
	
	`kb = """`
	`a :- b, c.`
	`b :- d, e.`
	`b :- g, e.`
	`c :- e.`
	`d.`
	`e.`
	`f :- a,`
	`  g.`
	`"""`
	
	`print(", ".join(sorted(forward_deduce(kb))))`
		a, b, c, d, e	a, b, c, d, e	✔

✔	`from student_answer import forward_deduce`
	
	`kb = ""`
	`print(", ".join(sorted(forward_deduce(kb))))`
				✔

✔	`from student_answer import forward_deduce`
	
	`kb = """`
	`a.`
	`z.`
	`"""`
	`print(", ".join(sorted(forward_deduce(kb))))`
		a, z	a, z	✔

✔	`from student_answer import forward_deduce`
	
	`kb = """`
	`wet :- is_raining.`
	`wet :- sprinkler_is_going.`
	`wet.`
	`"""`
	
	`print(len(forward_deduce(kb)))`
		1	1	✔

	Test	Expected	Got	

✔	`from student_answer import forward_deduce`
	
	`kb = """`
	`this_is_true :- this_is_true.`
	`"""`
				✔

Passed all tests! ✔

Correct

Marks for this submission: 3.00/3.00.

Question 6

Correct

Mark 3.00 out of 3.00

Write a class `KBGraph` that poses a knowledge base and a query as a graph. The intention is to use the graph with a graph search algorithm to have a top-down proof procedure. The query will be a set of atoms (strings). If you wish you can use the template provided in the answer box. You must also provide an implementation of `DFSFrontier`. You can simply copy this across from the graph search quiz if you have answered that question.

The graph class will not be tested on the order of edges; you can generate edges in whatever order you wish. The input knowledge base is guaranteed to not have cyclic clauses. For example the following is NOT an example input:

```
a :- b.
b :- c.
c :- a.
d.
```

Notes

- See the top-down proof procedure, answer clauses, and the example search tree for an SLD resolution in the lecture notes.
- The top-down proof procedure presented in the lecture notes has a non-deterministic statement ("choose"). For implementation, this has to be translated to some form of backtracking. This is why you are being asked to provide a graph class and a DFS frontier class. These two together implicitly achieve backtracking.
- You need to think about the right representation for nodes. Since all *answer clauses* are of the form `yes :- a_body`, you can factor out the 'yes' part and only represent the body.
- After answering this question, as an additional exercise for yourself, improve your program such that knowledge bases with cycles can be handled.
- If the graph search determines that the given query is true, the proof of it can be produced by printing the path. Therefore it is useful to have meaningful labels for the edges of the graph (where each label is a string representation of a rule from the KB). You are not required to implement this in your answer.
- The length of the proof will depend on the search strategy (the type of frontier used for the graph search).

For example:

Test	Result

`from search import *`
`from student_answer import KBGraph, DFSFrontier`
 	
`kb = """`
`a :- b, c.`
`b :- d, e.`
`b :- g, e.`
`c :- e.`
`d.`
`e.`
`f :- a,`
`  g.`
`"""`
 	
`query = {'a'}`
`if next(generic_search(KBGraph(kb, query), DFSFrontier()), None):`
`  print("The query is true.")`
`else:`
`  print("The query is not provable.")`
	`The query is true.`

Test	Result

`from search import *`
`from student_answer import KBGraph, DFSFrontier`
 	
`kb = """`
`a :- b, c.`
`b :- d, e.`
`b :- g, e.`
`c :- e.`
`d.`
`e.`
`f :- a,`
`g.`
`"""`
 	
`query = {'a', 'b', 'd'}`
`if next(generic_search(KBGraph(kb, query), DFSFrontier()), None):`
`print("The query is true.")`
`else:`
`print("The query is not provable.")`
	The query is true.

`from search import *`
`from student_answer import KBGraph, DFSFrontier`
 	
`kb = """`
`all_tests_passed :- program_is_correct.`
`all_tests_passed.`
`"""`
 	
`query = {'program_is_correct'}`
`if next(generic_search(KBGraph(kb, query), DFSFrontier()), None):`
`print("The query is true.")`
`else:`
`print("The query is not provable.")`
	The query is not provable.

`from search import *`
`from student_answer import KBGraph, DFSFrontier`
 	
`kb = """`
`a :- b.`
`"""`
 	
`query = {'c'}`
`if next(generic_search(KBGraph(kb, query), DFSFrontier()), None):`
`print("The query is true.")`
`else:`
`print("The query is not provable.")`
	The query is not provable.

Answer: (penalty regime: 0, 15, ... %)

Reset answer

```
import re
from search import *

def clauses(knowledge_base):
    """Takes the string of a knowledge base; returns an iterator for pairs
    of (head, body) for propositional definite clauses in the
    knowledge base. Atoms are returned as strings. The head is an atom
    and the body is a (possibly empty) list of atoms.

    -- Kourosh Neshatian - 2 Aug 2021

    """
    ATOM   = r"[a-z][a-zA-Z\d_]*"
    HEAD   = rf"\s*(?P<HEAD>{ATOM})\s*"
    BODY   = rf"\s*(?P<BODY>{ATOM}\s*(,\s*{ATOM}\s*)*)\s*"
    CLAUSE = rf"{HEAD}(:-{BODY})?\."
    KB     = rf"^({CLAUSE})*\s*$"

    assert re.match(KB, knowledge_base)

    for mo in re.finditer(CLAUSE, knowledge_base):
        yield mo.group('HEAD'), re.findall(ATOM, mo.group('BODY') or "")
```

```
23
24 def children(cans, inputs):
25     ans = []
26     to_add = inputs[len(cans)]
27     for i in to_add:
28         child = cans + [i]
29         ans.append(child)
30     return ans
31
32 def back(cans, inputs, output):
33     if len(cans) == len(inputs):
34         output.append(cans)
35     else:
36         for child in children(cans, inputs):
37             back(child, inputs, output)
38
39 def my_func(all_ways):
40     """take a list of lists and do permutation and finally return a list with
41     unique atoms.
42     """
43     ans = []
44     back([], all_ways, ans)
45     res = []
46     for item in ans:
47         sets = set()
48         for atoms in item:
49             sets = sets.union(set(atoms))
50         res.append(list(sets))
51     return res
52
```

	Test	Expected	Got	

✔	`from search import *`
	`from student_answer import KBGraph, DFSFrontier`
	
	`kb = """`
	`a :- b, c.`
	`b :- d, e.`
	`b :- g, e.`
	`c :- e.`
	`d.`
	`e.`
	`f :- a,`
	`g.`
	`"""`
	
	`query = {'a'}`
	`if next(generic_search(KBGraph(kb, query), DFSFrontier()),`
	`None):`
	`print("The query is true.")`
	`else:`
	`print("The query is not provable.")`
		The query is true.	The query is true.	✔

	Test	Expected	Got	

✔	`from search import *`
	`from student_answer import KBGraph, DFSFrontier`
	
	`kb = """`
	`a :- b, c.`
	`b :- d, e.`
	`b :- g, e.`
	`c :- e.`
	`d.`
	`e.`
	`f :- a,`
	`  g.`
	`"""`
	
	`query = {'a', 'b', 'd'}`
	`if next(generic_search(KBGraph(kb, query), DFSFrontier()), None):`
	`  print("The query is true.")`
	`else:`
	`  print("The query is not provable.")`
		The query is true.	The query is true.	✔

✔	`from search import *`
	`from student_answer import KBGraph, DFSFrontier`
	
	`kb = """`
	`all_tests_passed :- program_is_correct.`
	`all_tests_passed.`
	`"""`
	
	`query = {'program_is_correct'}`
	`if next(generic_search(KBGraph(kb, query), DFSFrontier()), None):`
	`  print("The query is true.")`
	`else:`
	`  print("The query is not provable.")`
		The query is not provable.	The query is not provable.	✔

✔	`from search import *`
	`from student_answer import KBGraph, DFSFrontier`
	
	`kb = """`
	`a :- b.`
	`"""`
	
	`query = {'c'}`
	`if next(generic_search(KBGraph(kb, query), DFSFrontier()), None):`
	`  print("The query is true.")`
	`else:`
	`  print("The query is not provable.")`
		The query is not provable.	The query is not provable.	✔

✔	`from search import *`
	`from student_answer import KBGraph, DFSFrontier`
	
	`kb = ""`
	
	`query = {'proposition'}`
	`if next(generic_search(KBGraph(kb, query), DFSFrontier()), None):`
	`  print("The query is true.")`
	`else:`
	`  print("The query is not provable.")`
		The query is not provable.	The query is not provable.	✔

Passed all tests! ✔

Correct

Marks for this submission: 3.00/3.00.