CMSC 471 Spring HW4 Reference Solution

```
from logic import *
```

1. Checking Validity (10)

Use the functions in aima's logic.py to see which of the following are valid, i.e., true in every model. You will have to (i) convert these sentences to the appropriate string form that the python code uses (see the comments in the code) and (ii) use the expr() function in logic.py to turn each into an Expr object, and (iii) use the tt_true() function to check for validity. Provide text from a session that shows the result of checking the validity of each. We've done the first one as an example.

1.a P ∨ ¬P

```
tt_true(expr('P | ~P'))
```

```
True
```

1.b (P → Q) ↔ (¬P ∨ Q)

```
tt_true(expr('(P ==> Q) <=> (~P | Q) '))
```

```
True
```

1.c P ∧ Q → (Q ∨ P)

```
tt_true(expr('P & Q ==> (Q | P)'))
```

```
True
```

1.4d P ∧ Q → Q

```
tt_true(expr('P & Q ==> Q'))
```

```
True
```

1.e ((A ∧ B) → C) ↔ (A → (B → C))

```
tt_true(expr('((A & B) ==> C) <=> (A ==> ( B ==> C))'))
```

```
True
```

1.f ((A → B) → A) → A

```
tt_true(expr('((A==>B)==>A)==>A'))
```

```
True
```

2. Satisfiability (14)

Use the functions in logic.py to see which of the following are are satisfiable. We've done the first one as an example. One way to explore the use of dpll_satisfiable is with the notebook file statisfiability.ipynb in the hw4 code repository.

2.a P ∧ Q

```
dpll_satisfiable(expr('P & Q'))
```

```
{Q: True, P: True}
```

2.b ALIVE ↔ ¬DEAD

```
dpll_satisfiable(expr('ALIVE <=> ~DEAD'))
```

```
{ALIVE: True, DEAD: False}
```

2.c P → ¬ P ∨ P

```
dpll_satisfiable(expr('P ==> ~P | P'))
```

```
{P: True}
```

2.d ~ (P ∨ ¬ P)

```
dpll_satisfiable(expr('~(P | ~ P)'))
```

```
False
```

2.e P ∧ (P → Q)

```
dpll_satisfiable(expr('P & (P ==> Q)'))
```

```
{Q: True, P: True}
```

2.f P ∧ (Q →P)

```
dpll_satisfiable(expr('P & (Q ==> ~P)'))
```

```
{Q: False, P: True}
```

2.g P ∧ (Q ∨ ¬P)

```
dpll_satisfiable(expr('P & (Q | ~P)'))
```

```
{Q: True, P: True}
```

2.h P ∧ ¬Q ∧ (P → Q)

```
dpll_satisfiable(expr('P & ~Q & (P ==> Q)'))
```

```
False
```

3. Propositional Consequence (14)

For each of the following entailment relations, say whether or not it is true. The text on the left of the entailment symbol (⊨) represents one or more sentences (separated by commas) that constitute a knowledge base. We've done the first one for you.

3.a P ∧ Q ⊨ P

```
tt_entails(P & Q, P)
```

```
True
```

3.b P ⊨ P ∨ Q

```
tt_entails(P,P | Q)
```

```
True
```

3.c ¬P ⊨ ¬ ¬ P

```
tt_entails(-P,--P)
```

```
False
```

3.d P → Q ⊨ ¬ P → ¬ Q

```
tt_entails(P | '==>'| Q , -P |'==>'| -Q)
```

```
False
```

3.e ¬ P ⊨ P → Q

```
tt_entails(- P, P | '==>' | Q)
```

```
True
```

3.f ¬ Q ⊨ P → Q

```
tt_entails(-Q,P|'==>'|Q)
```

```
False
```

3.g P ∧ (P → Q) ⊨ Q

```
tt_entails(P & (P | '==>'|Q),Q)
```

```
True
```

3.h (¬ P) ∧ (Q → P) ⊨ ¬ Q

```
tt_entails((-P)&(Q|'==>'|P),-Q)
```

```
True
```

4. English to FOL (25)

Translate the following English sentences into first order logic, describing the intended meaning of any non-obvious predicates. Feel free to optionally provide a more direct paraphrase of the meaning of your logic expression in English. If you think a sentence is ambiguous, describe the ambiguity and give logical expressions for all interpretations. We've done the first one for you using a notation with simple ASCII characters for logical operators (e.g., A:∀, E:∃, =>:→, <=>:↔, ^:∧, v:∨, ~:¬, >:>)

4.1 There is no largest prime number.

~(Ex number(x) ^ prime(x) ^ (Ay number(y) ^ prime(y) -> y >= x))

paraphrase: It is not true that there is a prime number and it is greater than or equal to all prime numbers.

predicates: number(x) is true if x is a number and prime(x) is true if x is a prime number.

4.2 Good food is not cheap and cheap food is not good.

```
(Ax) (food(x) ^ good(x) -> ~cheap(x)) ^ (food(x) ^ cheap(x) -> ~good(x))
```

paraphrase:

```
Any food that is good is not cheap and any food that is cheap is not good.
```

predicates:

```
food(X) is true iff X is an instance of a food, good(X) is true iff X is considered to be good, cheap(X) is True iff the cost of X is
```

4.3 John has exactly two brothers.

```
(Ex) (Ey) brother(John,x) ^ brother(John,y ) ^ ~(x = y) ^ (Az) brother(John,z) -> ((x = z) v (y = z))
```

paraphrase:

```
John has two brothers, x and y, that are different. Also, any other brother that john has must be the same as either x or y.
```

predicates:

```
brother(X,Y) is true if person X has person Y as a brother.
```

4.4 Every dog is either male or female and no dog can be both male and female.

```
(Ax) (dog(x) -> (male(x) V female(x))) ^ ~(Ex) (dog(x) ^ male(x) ^ female(x))
```

paraphrase:

```
Any dog is either male or female and no dog is both male and female.
```

predicates:

```
dog(X) is True iff X is a dog
male(X) is true if the sex of organism X is male
female(X) is true if the sex of organism X is female
```

4.5 The friend of your enemy is your enemy.

```
(Ax)(Ay)(Az) (enemy(x,y) ^ friend(y,z)) -> enemy(x,z)
```

paraphrase:

```
Every friend of a person's enemy is also that person's enemy
```

predicates:

```
friend(x,y) is True if x and y are friends
enemy(x,y) is True if  x is y's enemy.
```

4.6 An ancestor of your ancestor is your ancestor.

```
(Ax) (Ay) (Az) ancestor(x,y) ^ ancestor(y,z) -> ancestor(x,z)
```

paraphrase:

```
If you have an ancester, any ancestor of that ancester is also your ancester.
```

predicates:

```
ancestor(x,y) is True x is y's ancester.
```

5. Expressing knowledge in CNF (15)

Express each of the above as a set of clauses in conjunctive normal form

a. A ∨ (B ∧ C)

A ∨ B A ∨ C

b. A ∨ B ∨ C

A ∨ B ∨ C

c. A ∧ B ∧ C

A B C

d. ((A ∧ B) ⇔ C)

(A V ~C) (B V ~C) (C V ~A V ~B)

e. A => (B ⇔ C)

~A V B V ~C ~A V ~B V C

f. A => ((B => C) ∨ ~C)

~A V ~B you should also accept ~A V ~B V ~C V C

We can also use the logic.py function **to_cnf** that takes an arbitraty popositional sentence and returns an equivalent one in conjunctive normal form, i.e., as a conjunction of expressions where each is a discunction of positive or negative atoms

```
# 5a
to_cnf(expr("A | (B & C)"))
```

```
((B | A) & (C | A))
```

```
# 5b
to_cnf(expr("A | B | C"))
```

```
(A | B | C)
```

```
# 5c
to_cnf(expr("A & B & C"))
```

```
(A & B & C)
```

```
# 5d
to_cnf(expr("((A & B) <=> C)"))
```

```
((A | ~C) & (B | ~C) & (C | ~A | ~B))
```

```
# 5e
to_cnf(expr(" A ==> (B  <=>  C)"))
```

```
((B | ~C | ~A) & (C | ~B | ~A))
```

```
# 5f
to_cnf(expr("A ==> (( B ==> C) |  -C)"))
```

```
(C | ~B | ~C | ~A)
```

6. Logic Puzzle (22)

The Three Stooges visited you recently and one of them ate the piece of pie you were saving in your refrigerator. You don't know who did it. But you do know that:

- Only one of them ate the pie
- One of the three always tells the truth and the other two always lie

You don't know which one is the truthful one. So, you ask each one who ate the pie, and they give the following answers.

```
Moe: "I ate the pie"
Larry: "I did not eat the pie"
Curly: "Moe did not eat the pie"
```

Maybe the stooges thought that they support one another by confusing you, but you were able to use logic to determine who the pie eater was. Explain your reasoning by (a) mapping the problem into propositional logic and (b) showing how the AIMA code can be used to solve this problem.

Variables for who ate the pie

```
P1 : True if Moe ate the pie
P2 : True if Larry ate the pie
P3 : True if Curly ate the pie
```

```
# COndition 1: only one of P1, P2 and P3 is true
C1 = expr('(P1 & ~P2 & ~P3) | (~P1 & P2 & ~P3) | (~P1 & ~P2 & P3)')
```

Represent what each stooge says using P1, P2 and P3

```
Moe says "I ate the pie" : P1
Larry says "I did not eat the pie" ~P2
Curly says "Moe did not eat the pie": ~P1
```

```
# Condition 2: only one stooge is correct, the other two lie
C2 = expr('(P1 & ~~P2 & ~~P1) | (~P1 & ~P2 & ~~P1) | (~P1 & ~~P2 & ~P1)')
```

Using dbll_satisfiability we can show that

- P2 is true, i.e., Larry ate the pie

Which means that Larry tells the truth and the others lie

```
dpll_satisfiable( C1 & C2 )
```

```
{P1: False, P2: True, P3: False}
```

Students need not do the following. dpll_satisfiable returns just one way to satisty a statement, but there might be others. To be sure about the problem, we can check that (C1 & C2) is only satisfiable with one of of P1, P2 or P3 being true.

```
print("Assume it's Moe:", dpll_satisfiable( C1 & C2 & expr('P1')))
print("Assume it's Larry:", dpll_satisfiable( C1 & C2 & expr('P2')))
print("assume it's Curly:", dpll_satisfiable( C1 & C2 & expr('P3')))
```

```
Assume it's Moe: False
Assume it's Lsrry: {P2: True, P1: False, P3: False}
assume it's Curly: False
```

We can also use WalkSAT to prove find a way to satisfy the two conditaions

```
WalkSAT([C1, C2])
print("Assume it's Moe:", WalkSAT([C1, C2, expr('P1')]))
print("Assume it's Larry:", WalkSAT([C1, C2, expr('P2')]))
print("assume it's Curly:", WalkSAT([C1, C2, expr('P3')]))
```

```
Assume it's Moe: None
Assume it's Lsrry: {P1: False, P3: False, P2: True}
assume it's Curly: None
```

fin