Thursday
Week 12
4-6-17

Quiz 19

Write the answer sets of the following programs:

1. `1 { a, b, c } 2.`
   - Set 1: a
   - Set 2: b
   - Set 3: c
   - Set 4: a, b
   - Set 5: a, c
   - Set 6: b, c
2. `{a, b, c } 2.`
   - Set 1: ∅
   - Set 2: a
   - Set 3: b
   - Set 4: c
   - Set 5: a, b
   - Set 6: a, c
   - Set 7: b, c
3. `1 { a, b, c }.`
   - Set 1: a
   - Set 2: b
   - Set 3: c
   - Set 4: a, b
   - Set 5: a, c
   - Set 6: b, c
   - Set 7: a, b, c
4.
```
p(a;b;c).
1 { q(X) : p(X) } 1.
```
   - Set 1: q(a), p(a), p(b), p(c)
   - Set 2: q(b), p(a), p(b), p(c)
   - Set 3: q(c), p(a), p(b), p(c)
5.
```
p(a;b;c).
1 { q(X) } 1 :- p(X).
```
   - Set 1: q(a), q(b), q(c), p(a), p(b), p(c)
6.
```
p(a;b;c).
0 {q(X) } 1 :- p(X).
```
   - Set 1: p(a), p(b), p(c)
   - Set 2: q(a), p(a), p(b), p(c)
   - Set 3: q(b), p(a), p(b), p(c)
   - Set 4: q(c), p(a), p(b), p(c)
   - Set 5: q(a), q(b), p(a), p(b), p(c)

Set 6: `q(a), q(c), p(a), p(b), p(c)`
Set 7: `q(b), q(c), p(a), p(b), p(c)`
Set 8: `q(a), q(b), q(c), p(a), p(b), p(c)`

7.
```
p(a;b).
r(1;2).
1 { q(X, Y) : p(X) } 1 :- r(Y).
```
Set 1: `q(a, 1), q(a, 2), r(1), r(2), p(a), p(b)`
Set 2: `q(a, 1), q(b, 2), r(1), r(2), p(a), p(b)`
Set 3: `q(b, 1), q(a, 2), r(1), r(2), p(a), p(b)`
Set 4: `q(b, 1), q(b, 2), r(1), r(2), p(a), p(b)`

Note:
For problem 6,
```
0 {q(X)} 1 :- p(X).
```
is like three rules:
```
0 {q(X)} 1 :- p(a).
0 {q(X)} 1 :- p(b).
0 {q(X)} 1 :- p(c).
```

How to use Clingo:

1. Go to Clingo directory, create a file with the rules (called "cons1.txt" in this example).
2. Go to that directory in the console.
3. Type `./clingo cons1.txt 0`
    a. Replace cons1.txt with the name of your file
    b. The number following the file name will tell clingo how many answer sets to output.
        i. 0 outputs all answer sets
        ii. 1 outputs 1 answer set, 2 outputs 2 answer sets, etc.

Slide: Use of existential and universal quantifiers

```
block(b1;b2;b3;b4).
color(b1,blue).
color(b2,red).
color(b3,blue).
color(b4,red).

ontable(b1).
ontable(b3).
```

Q1: Write a rule to determine if there's a blue block on the table.
```
ans1(yes) :- ontable(X), color(X,blue).
```

Q2: List all the blocks on the table that are blue in color.
```
ans2(X) :- ontable(X), color(X,blue).
```

Q3: Write rules to determine if all the blocks on the table are blue.

```
ans(no) :- ontable(X), not color(X,blue).
ans(yes) :- not ans(no).
```

To test if the answer to Q3 is correct, add another block to the table:

```
block(b1;b2;b3;b4).
color(b1,blue).
color(b2,red).
color(b3,blue).
color(b4,red).

ontable(b1).
ontable(b2).
ontable(b3).

ans(no) :- ontable(X), not color(X,blue).
ans(yes) :- not ans(no).
```

We are saying

$\forall X\ p(X) = \neg\exists X$ not $p(X)$.

That is, "for all items X, p(X) is true if there are no items not in p(X)."

Or, to rephrase in plain English: "For each blue block, check if it's on the table. If there are no blue blocks that are not on the table, then the answer is 'yes.'"

We must do it this way, because there is no single command to check if all items are true. That is, the rule

`ans(X)` = $\forall X$, X is true.

cannot be written in a single line. Instead, we must check to see if there are any items in X that false, and only return true if none of the items returned false.

This is further explained on the next slide.

Slide: Existential and Universal quantifiers in the if part

1. `p :- domain(X), q(X).`
   a. p is true if there exists an X in the domain where q(X) is true.
   b. Used to make sure at least one item satisfies the condition.
2. `p' :- domain(X), not q(X).`
   `p :- not p'.`
   a. p is true if, for all X in the domain, q(X) is true.
   b. Used to make sure all items satisfy the condition.

Slide: Use of “not” and quantifier scope

```
app(a;b;c).
%a, b, and c are applicants

hclass(d;e;f).
%d, e, and f are honors classes

took(a, d).
took(a, e).
took(a, f).
took(b, d).
took(b, e).
%took(X,Y) means X took class Y.
```

Q1: Write a program such that applicants are refused admission if they have not taken at least one honors class:

```
accept(X) :- app(X), hclass(Y), took(X, Y).
refuse(X) :- app(X), not accept(X).
```

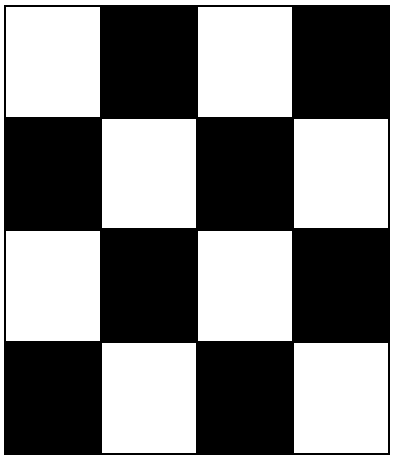
Note: `app(X)` *must* be in the second rule, or else you will get a syntax error due to an “unrestricted variable.”

Note: It's safer to subrtract from predicate with same arity.

TESTING NOTE: Quizzes will have questions of this kind, where we are given facts and have to write rules.

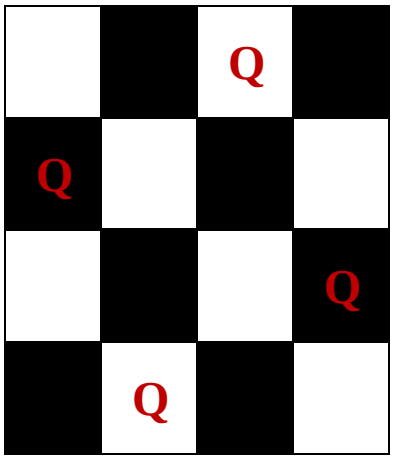
Placing Queens on a chessboard.

Suppose we have a 4 × 4 chessboard:



On this chess board, we wish to arrange Queen pieces to put as many as possible while not allowing any of them to attack others. To accomplish this, we would place queens so that there was one queen per row/column, and no queens were alligned horizontally, vertically, or diagonally.

One example of such a configuration would be:



How would we program this?

Note: The solution presented in class had a minor mistake. These notes have been corrected using the program Professor Baral e-mailed on April 7th. Also, please note that there are differences between Clingo 3.X, which Professor Baral used in class, and Clingo 4.X, which is likely what you have installed. For instructions on how to convert the following from Clingo 3.X to Clingo 4.X, please see the second e-mail sent by Professor Baral on April 7th.

Note: For clarity, the entire solution without comments will be presented at the end.

```
row(1..4).
col(1..4).
```

Note: This is same as `row(1;2;3;4)` and `col(1;2;3;4)`.

It is best to define a predicate before coding in order to figure out the logic.
In this case, our predicate can be defined as:

```
%queen(X,Y) – There is a queen in row X and col Y.
```

One way to code this (note that this is not necessarily the best way):

```
0 { queen(X,Y) } 1 :- row(X), col(Y).
```

As we have four rows and four columns, we will end up with 2^{16} answer sets.

Side note:
If we had:

```
row(1..2).
col(1..2).
```

then

```
0 { queen(X,Y) } 1 :- row(X), col(Y).
```

would result in 2^4 answer sets.
If we had:

```
row(1..3).
col(1..3).
```

then

```
0 { queen(X,Y) } 1 :- row(X), col(Y).
```

would result in 2^9 answer sets.

In short,

```
row(1..w).
col(1..z).
```

results in $2^{w \times z}$ answer sets.

Note that this is related to one of the questions on the quiz.
End side note.

Now we need to eliminate the answer sets which do not satisfy our conditions.

Discard cases where there's a queen in the same row:

```
:- row(X), row(XX), col(Y), queen(X,Y), queen(XX,Y),
     X != XX.
```

Discard cases where there's a queen in the same column.

```
:- col(Y), col(YY), row(X), queen(X,Y), queen(X,YY),
     Y != YY.
```

Discard cases where two queens line up diagonally:

```
:- row(X), row(XX), col(Y), col(YY), queen(X,Y),
     queen(XX,YY), X != XX, Y!=YY,
     #abs(X-XX) == #abs(Y-YY).
```

Ensure there's at least one queen per row/column. To do this, we have three options:

1. Count the queens on the board
2. Make sure each row has a queen.
3. Make sure each column has a queen.

We will use option 2:

```
hasqueen(X) :- row(X), col(Y), queen(X,Y).
:- row(X), not hasqueen(X).
```

And then we can hide the variables that we do not need to see:

```
#hide row(X).
#hide col(Y).
#hide hasqueen(X).
```

This results in two answer sets:

Set 1: queen(2,1), queen(4,2), queen(1,3), queen(3,4)

Set 2: queen(3,1), queen(1,2), queen(4,3), queen(2,4)

Entire Program without comments:

```
row(1..4).
col(1..4).
%queen(X,Y) – There is a queen in row X and col Y.

0 { queen(X,Y) } 1 :- row(X), col(Y).

:- row(X), row(XX), col(Y), queen(X,Y), queen(XX,Y), X != XX.
:- col(Y), col(YY), row(X), queen(X,Y), queen(X,YY), Y != YY.

:- row(X), row(XX), col(Y), col(YY), queen(X,Y), queen(XX,YY), X !=
XX, Y!=YY, #abs(X-XX) == #abs(Y-YY).

hasqueen(X) :- row(X), col(Y), queen(X,Y).

:- row(X), not hasqueen(X).

#hide row(X).
#hide col(Y).
#hide hasqueen(X).
```