Homework 9 – Spring 2023 (Due: June 9, 2023)

In this homework, you'll explore Prolog, creating facts and rules to answer simple queries, understanding how unficiation works, and processing lists. Some questions have multiple, distinct answers which would be acceptable, so there might not be a "right" answer: what's important is your ability to justify your answer with clear and concise reasoning that utilizes the appropriate terminology discussed in class. Each question has a time estimate; you'll know you're ready for the exam when you can solve them roughly within their time constraints.

We understand, however, that as you learn these questions may take more time. For that reason, only **starred red** questions need to be completed when you submit this homework (the rest should be used as exam prep materials). Note that for some multi-part questions, not all parts are marked as red so you may skip unstarred subparts in this homework.

You must turn in a PDF file with your answers via Gradescope - you may include both typed and handwritten solutions, so long as they are legible and make sense to our TAs. Make sure to clearly label each answer with the problem number you're solving so our TAs can more easily evaluate your work.

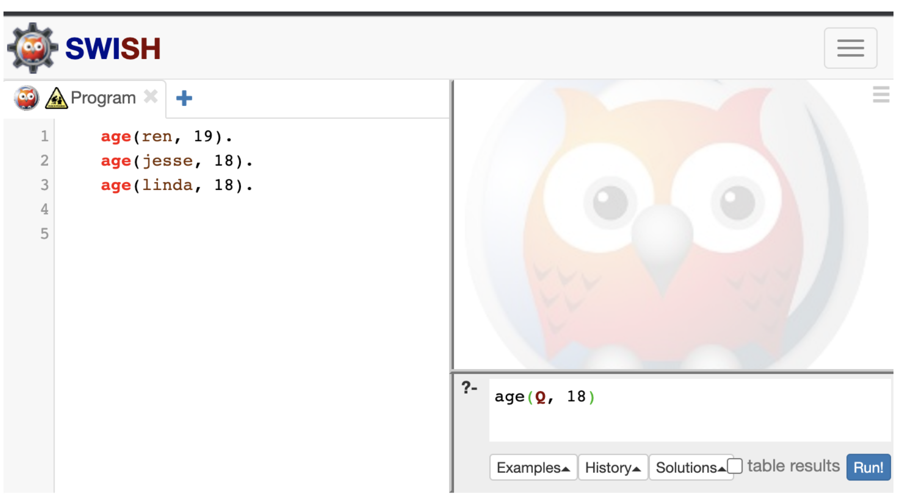
Details on using SWI Prolog

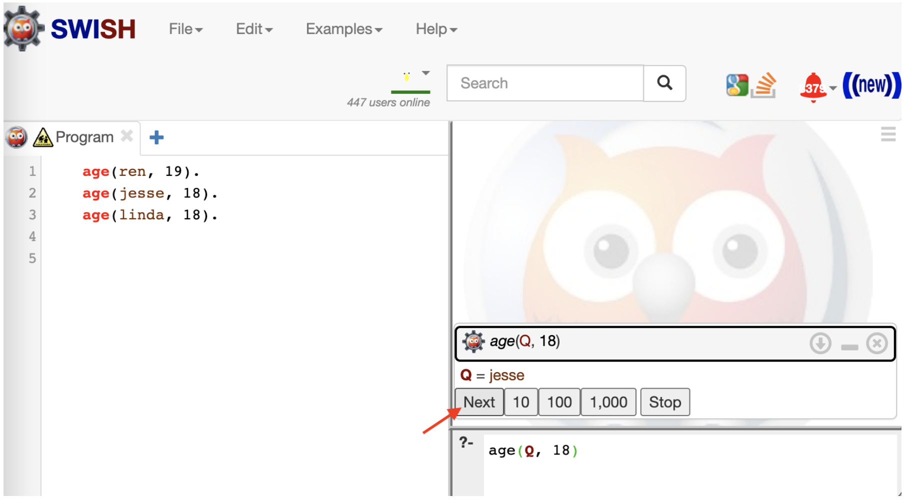
For these problems, you'll want to use the SWI Prolog website to test out your code:

https://swish.swi-prolog.org/

Using SWI Prolog:

- Go to the website
- Start by clicking "Program" in the top-middle of the screen (next to Notebook)
- Type in your facts/rules in the box on the left box which is titled "Program"; don't forget periods at the end of each fact/rule!
- Type in a single query at a time in the bottom-right box which has a ?- prompt. You don't want a period at the end of the query here.
- To run the query, click the "Run!" button at the bottom right of the query box.
- A query may return multiple values. To get the subsequent values after the first one has been returned, click the Next button at the bottom of the "owl" window:





- When you're done with a query, click the Stop button if one is shown to terminate the query (see image above, four buttons right of the Next button)

1. ** (10 min.) Consider the following if-statements which evaluates both AND and OR clauses:

```
if (e() || f() && g() || h())
  do_something();
if (e() && f() || g() && h())
  do_something();
if (e() && (f() || g())
  do_something();
```

How do you think short-circuiting will work with such an expression with both boolean operators and/or parenthesis? Try out some examples in C++ or Python to build some intuition. Give pseudocode or a written explanation.

Solution: It's just as you expect, except we have to deal with operator precedence. In general, AND takes precedence over OR in most languages. So consider the following if:

if (a() || b() && c() || d())

Since && has precedence over ||, this would be evaluated as follows:

if (a() || (b() && c()) || d())

Similarly, consider this if:

if (a() && b() || c() && d())

Again && has precedence over ||, so this would be evaluated as follows:

```
if ((a() && b()) || (c() && d()))
```

Once you explicitly take into account the precedence (e.g., by adding parentheses) just evaluate the expression from left to right, short circuiting as before. When we begin evaluating a parenthesized clause, e.g., (a() && b()), we evaluate its components from left to right, and short circuit internally as before. One we get a result from a parenthesized clause, we then continue on the next higher-level clause and continue if necessary.

2. ** For this problem, you will be using the following simple Hash Table class and accompanying Node class written in Python:

```
class Node:
 def __init__(self, val):
   self.value = val
   self.next = None

class HashTable:
 def __init__(self, buckets):
   self.array = [None] * buckets

 def insert(self, val):
   bucket = hash(val) % len(self.array)
   tmp_head = Node(val)
   tmp_head.next = self.array[bucket]
   self.array[bucket] = tmp_head
```

a) ** (10 min.) Write a Python generator function capable of iterating over all of the items in your HashTable container, and update the HashTable class so it is an iterable class that uses your generator.

Solution:

```
def generator(array):
 for i in range(len(array)):
   cur = array[i]
   while cur != None:
     yield cur.value
     cur = cur.next
```

b) ****** (10 min.) Write a Python iterator class capable of iterating over all of the items in your HashTable container, and update the HashTable class so it is an iterable class that uses your iterator class.

Solution:

```
class HTIterator:
  def __init__(self,array):
    self.array = array
    self.cur_buck = -1
    self.cur_node = None

  def __next__(self):
   while self.cur_node == None:
     self.cur_buck += 1
     if self.cur_buck >= len(self.array):
       raise StopIteration
     self.cur_node = self.array[self.cur_buck]
   val = self.cur_node.value
   self.cur_node = self.cur_node.next
   return val
```

c) ****** (1 min.) Write a for loop that iterates through your hash table using idiomatic Python syntax, and test this with both your class and generator.

Solution:

```
a = HashTable(100)
a.insert(10)
a.insert(20)
a.insert(30)
for i in a:
      print(i)
```

d) ** (5 min.) Now write the loop manually, directly calling the dunder functions (e.g., __iter__) to loop through the items.

```
a = HashTable(100)
a.insert(10)
a.insert(20)
a.insert(30)
iter = a.__iter__()
try:
  while True:
    val = iter.__next__()
    print(val)
except StopIteration:
 pass
```

e) ** (5 min.) Finally, add a forEach() method to your HashTable class that accepts a lambda as its parameter, and applies the lambda that takes a single parameter to each item in the container:

```
ht = HashTable()
```

```python
# add a bunch of things
ht.forEach(lambda x: print(x))
```

Solution:

```python
class HashTable:
…
 def forEach(self, f):
   for i in range(len(self.array)):
     cur = array[i]
     while cur != None:
       f(cur.value)
       cur = cur.next
```

3. ** Consider the following facts:

```
color(celery, green).
color(tomato, red).
color(persimmon, orange).
color(beet, red).
color(lettuce, green).
```

What will the following queries return? And in what order will Prolog return their results (e.g., if you ask what vegetables are red, will `beet` be output first or `tomato`)? Try to figure out the result in your head first, then use SWI Prolog if you can't figure out what will happen.

a) ** (2 min.) `color(celery, X)`

`Solution: X = green`

b) ** (2 min.) `color(tomato, orange)`

`Solution: false`

c) ** (2 min.) `color(Q, red)`

```
Solution:
Q = tomato
Q = beet
```

d) ** (4 min.) `color(Q, R)`

Solution:

Q = celery, **R** = green
Q = tomato, **R** = red
Q = persimmon, **R** = orange
Q = beet, **R** = red

Q = lettuce, **R** = green

4\. ** Consider the following facts:

```
color(carrot, orange).
color(apple, red).
color(lettuce, green).
color(subaru, red).
color(tomato, red).
color(broccoli, green).
food(carrot).
food(apple).
food(broccoli).
food(tomato).
food(lettuce).
likes(ashwin, carrot).
likes(ashwin, apple).
likes(xi, broccoli).
likes(menachen, subaru).
likes(menachen, lettuce).
likes(xi, mary).
likes(jen, pickleball).
likes(menachen, pickleball).
likes(jen, cricket).
```

a) ** (5 min.) Write a rule named `likes_red` that determines who likes foods that are red.

Solution:

```
likes_red(Who) :- food(X), likes(Who, X), color(X, red).
```

b) ** (5 min.) Write a rule named `likes_foods_of_colors_that_menachen_likes` that determines who likes foods that are the same colors as those that `menachen` likes. For example, if the foods `menachen` likes are `lettuce` and `banana_squash`, which are `green` and `yellow` respectively, and `jane` likes `bananas` (which are `yellow`), and `ahmed` likes `bell_peppers` (which are `green`), then your rule should identify `jane` and `ahmed`.

Example:

`likes_foods_of_colors_that_menachen_likes(X)` should yield:

```
X = xi
X = menachen
```

Solution:

```
likes_foods_of_colors_that_menachen_likes(Who) :-
    likes(menachen,F), food(F), color(F,C),
    likes(Who,F2), food(F2), color(F2,C).
```

5. ** (10 min.) Consider the following facts:

```
road_between(la, seattle).
road_between(la, austin).
road_between(seattle, portland).
road_between(nyc, la).
road_between(nyc,boston).
road_between(boston,la).
```

The `road_between` fact indicates there's a bi-directional road directly connecting both cities. Write a predicate called `reachable` which takes two cities as arguments and determines whether city A can reach city B through zero or more intervening cities.

Examples:

`reachable(la, boston)` should yield `True`.

`reachable(la, X)` should yield `X = seattle, X = austin, X = portland, X = nyc, X = la, X = boston`

The cities need not be in this order. Also, notice that la is reached from la (e.g., by going from la to seattle and back to la via the bidirectional edge), via a

Basic Solution which doesn't account for cycles:

```
reachable(X,Y) :- road_between(X,Y).
```

```
reachable(X,Y) :- road_between(Y,X).
reachable(X,Y) :- road_between(X,Z), reachable(Z,Y).
reachable(X,Y) :- road_between(Y,Z), reachable(Z,X).
```

Better Solution which accounts for cycles:

```
reachable(X, Y) :- reachable(X, Y, [X]).
reachable(X, Y, _) :- road_between(X, Y).
reachable(X, Y, Visited) :-
    road_between(X, Z),
    not(member(Z, Visited)),
    reachable(Z, Y, [Z | Visited]).
reachable(X, Y, Visited) :-
    road_between(Z, X),
    not(member(Z, Visited)),
    reachable(Z, Y, [Z | Visited]).
```

6. ** (5 min) Which of the following predicates unify? If they unify, what mappings are outputted? If they do not unify, why not?

```
foo(bar,bletch) with foo(X,bletch)

foo(bar,bletch) with foo(bar,bletch,barf)

foo(Z,bletch) with foo(X,bletch)

foo(X, bletch) with foo(barf, Y)

foo(Z,bletch) with foo(X,barf)

foo(bar,bletch(barf,bar)) with foo(X,bletch(Y,X))

foo(barf, Y) with foo(barf, bar(a,Z))

foo(Z,[Z|Tail]) with foo(barf,[bletch, barf])
```

`foo(Q)` with `foo([A,B|C])`

`foo(X,X,X)` with `foo(a,a,[a])`

Hint: If you want to check your work, you can use SWI Prolog and type this in the query window to check for unification and see what mappings Prolog finds:

```
foo(todd) = foo(X)
```

The following predicates unify:

`foo(bar,bletch)` with `foo(X,bletch)`:

```
X = bar
```

`foo(Z,bletch)` with `foo(X,bletch)`:

```
X = Z
```

`foo(X, bletch)` with `foo(barf, Y)`:

```
X = barf, Y = bletch
```

`foo(bar,bletch(barf,bar))` with `foo(X,bletch(Y,X))`:

```
X = bar, Y = barf
```

`foo(barf, Y)` with `foo(barf, bar(a,Z))`:

```
Y = bar(a,Z)
```

`foo(Q)` with `foo([A,B|C])`:

`Q = [A,B|C]`

The following predicates do not unify:

`foo(bar,bletch)` with `foo(bar,bletch,barf)`:

Different arity.

`foo(Z,bletch)` with `foo(X,barf)`:

`bletch` ≠ `barf`

`foo(Z,[Z|Tail])` with `foo(barf,[bletch, barf])`:

`barf` ≠ `bletch`

`foo(X,X,X)` with `foo(a,a,[a])`:

`a` ≠ `[a]`

7. ** (10 min.) Below is a partially-written predicate named `insert_lex` which inserts a new integer value into a list in lexicographical order. Your job is to identify what atoms, Variables, or numbers should be written in the blanks.

Example:

`insert_lex(10, [2,7,8,12,15], X)` should yield `X = [2,7,8,10,12,15].`

```
% adds a new value X to an empty list
insert_lex(X,[],[______]).

% the new value is < all values in list
```

```
insert_lex(X,[Y|T],[X,______|T]) :- X =< Y.

% adds somewhere in middle
insert_lex(X,[Y|______],[Y|______]) :-
  X > Y, insert_lex(______,T,NT).
```

Solution:

```
insert_lex(X,[],[X]).
insert_lex(X,[Y|T],[X,Y|T]) :- X =< Y.
insert_lex(X,[Y|T],[Y|NT]) :-
  X > Y, insert_lex(X,T,NT).
```

Case #1 (base): We insert X into an empty list. We just return a list with only one item, X: `[X]`

Case #2 (base): X is smaller than the first item (Y) in the list of potentially many items. The second term `[Y|T]` pattern matches to break up the list we're inserting into the head item (Y) and the tail (T) items. If the item to insert X is less

than or equal to the head item, then we return a list with X concatenated before the first item.

Case #3: Again, we break up the list into the first item, Y, and all the tail items T. This rule runs if X > Y. The output will be Y (the current head item, which is less than X) concatenated onto some new list NT, where NT is the result of inserting X somewhere into T (the tail part of the original list).

8. ** (10 min.) Below is a partially-written predicate named `count_elem` which counts the number of items in a list. Your job is to identify what atoms, Variables, or numbers should be written in the blanks.

Examples:

`count_elem([foo, bar, bletch], 0, X)` should yield `X = 3.`

`count_elem([], 0, X)` should yield `X = 0.`

```
% count_elem(List, Accumulator, Total)
% Accumulator must always start at zero
count_elem([], ______, Total).
count_elem([Hd|______], Sum, ______) :-
```

```
  Sum1 is Sum + ______,
  count_elem(Tail, ______, Total).
```

Solution:

```
count_elem([], Total, Total).
count_elem([Hd|Tail], Sum, Total) :-
     Sum1 is Sum + 1,
     count_elem(Tail, Sum1, Total).
```

9. ** (15 min.) Write a predicate named `gen_list` which, if used as follows:

`gen_list(Value, N, TheGeneratedList)`

is provable if and only if `TheGeneratedList` is a list containing the specified `Value` repeated `N` times.

Example:

`gen_list(foo, 5, X)` should yield `X = [foo, foo, foo, foo, foo].`

Hint: You will need both a fact and a rule to implement this.

Solution:

```
gen_list(_, 0, []).
gen_list(Q, C,[Q | Output]) :-
    C > 0,
    NextCount is C - 1,
    gen_list(Q, NextCount, Output).
```

10. ** (15 min.) Write a predicate named `append_item` which, if used as follows:

`append_item(InputList, Item, ResultingList)`

is provable if and only if `ResultingList` is the result of appending `Item` onto the end of `InputList`.

Example:

`append_item([ack, boo, cat], dog, X)` should yield

`X = [ack, boo, cat, dog].`

Solution:

```
append_item([],X,[X]).
append_item([H|T],X,[H|L]) :- append_item(T,X,L).
```