Homework 9

This homework covers the following topics from class:

- OOP Palooza, part 4, Control Palooza part 1
    - Inheritance Topics cont. (Subtype Polymorphism, Dynamic Dispatch),
    - Expression evaluation (associativity, order of evaluation), short circuiting, control statements (branching, conditionals, multi-way selection)
- Control Palooza, part 2 (VIRTUAL CLASS)
    - Iteration and Iterators (Object-based, Generator-based, via 1st-class-function-based Loops)
    - Assignments:
- Control palooza, part 3, Logic Programming, part 1
    - Concurrency (multithreading and asynchronous programming)
    - Prolog history, language overview, statements (facts, rules, goals)
- Logic programming, part 2
    - Prolog resolution, unification, list processing

Each problem below has a time associated with it. This is the amount of time that we expect a thoroughly-prepared student would take to solve this problem on an exam. We understand, however, that as you learn these concepts, these questions may take more time to solve, and we don't want to overwhelm you with homework. For that reason, only **required** questions need to be completed when you submit this homework. That said, the exams will cover all materials covered (a) in class, (b) from class projects, (c) in the required homework problems, and (d) in the optional homework problems, so at a minimum, please review the optional problems and/or use them as exam prep materials.

Note: Because this homework is longer and covers topics from next week (Concurrency, Prolog), you will have until Sunday of next week to turn it in.

You must turn in a PDF file with your answers via Gradescope - you may include both typed and hand-written 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.

For homework #9, you must complete the following **required** problems:

- OOP8: Inheritance, Subtype Polymorphism, Dynamic Dispatch (5 min)
- OOP9: Subtype Polymorphism, Dynamic Dispatch, Dynamically Typed Languages (5 min)
- OOP11: Liskov Substitution Principle (5 min)
- CNTL1: Short Circuiting (5 min)
- CNTL2: Iterators, Iterator Classes, Generators, First-class Iteration (31 min)
- CNTL3: Async Programming (5 min)
- PRLG1: Swish Prolog (5 min)
- PRLG2: Facts (10 min)
- PRLG3: Facts, Rules (10 min)
- PRLG4: Facts, Rules, Recursion, Transitivity (10 min)
- PRLG5: Unification (5 min)
- PRLG6: Lists, Recursion (10 min)
- PRLG7: Lists, Recursion (10 min)
- PRLG9: Lists, Recursion (15 min)

For homework #9, you may optionally complete the following additional problems:

- OOP10: SOLID Principles (5 min)
- PRLG8: Lists, Recursion (10 min)

OOP8: Inheritance, Subtype Polymorphism, Dynamic Dispatch (5 min)

(5 min.) Explain the differences between inheritance, subtype polymorphism, and dynamic dispatch.

Your answer:

OOP9: Subtype Polymorphism, Dynamic Dispatch, Dynamically Typed Languages (5 min)

(5 min.) Explain why we don't/can't use subtype polymorphism in a dynamically-typed language. Following from that, explain whether or not we can use dynamic dispatch in a dynamically-typed language. If we can, give an example of where it would be used. If we can't, explain why.

Your answer:

OOP10: SOLID Principles (**Optional**, 5 min)

This problem discusses SOLID principles which are principles for good OOP class/interface design. We probably didn't cover this in class, so if you want to learn more about SOLID, check out the hidden slides in the OOP Palooza deck first. Search for "SOLID"

(5 min.) Consider the following classes:

```
class SuperCharger {
public:
  void get_power() { ... }
  double get_max_amps() const { ... }
  double check_price_per_kwh() const { ... }
};

class ElectricVehicle {
public:
  void charge(SuperCharger& sc) { ... }
};
```

Which SOLID principle(s) do these classes violate? What would you add or change to improve this design?

Your answer:

OOP11: Liskov Substitution Principle (5 min)

(10 min.) Does the Liskov substitution principle apply to dynamically-typed languages like Python or JavaScript (which doesn't even have classes, just objects)? Why or why not?

Your answer:

CNTL1: Short Circuiting (5 min)

(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.

Your answer:

CNTL2: Iterators, Iterator Classes, Generators, First-class Iteration (31 min)

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
```

Part 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.

Your answer:

Part 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.

Your answer:

Part 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.

Your answer:

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

Your answer:

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

Your answer:

CNTL3: Async Programming (5 min)

Figure out the order in which this asynchronous Python program runs.

Hints:
1. The asyncio.sleep() coroutine causes the currently-running coroutine to be suspended and re-added to the end of the event queue once the timer has expired.
2. It helps to use a piece of paper or text editor to represent the queue of coroutines and the output so far.

Note: You may be inclined to just use Python to figure out the answer, but we want you to really understand how async programming works (and may test you on this), so please take the time to solve this by hand.

```
import asyncio
```

```python
def sync_function():
    print("Sync call")  # Synchronous function call

async def foo():
    print("Foo start")
    await asyncio.sleep(1)
    print("Foo end")
    return "Foo result"

async def bar():
    print("Bar start")
    sync_function()  # Synchronous function call before the sleep
    await asyncio.sleep(2)
    print("Bar end")
    return "Bar result"

async def main():
    print("Main start")
    task1 = asyncio.create_task(foo())
    task2 = asyncio.create_task(bar())

    await asyncio.sleep(0.5)
    print("Main middle")

    result1 = await task1
    result2 = await task2

    print("Main end", result1, result2)

asyncio.run(main())
```

Your answer:

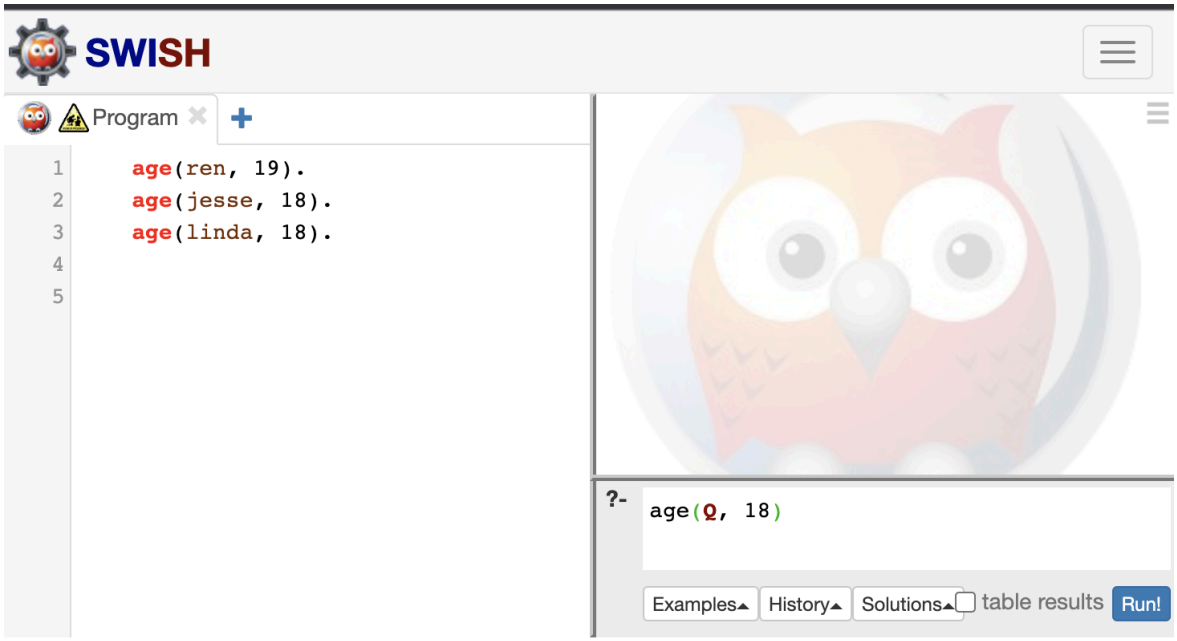
PRLG1: Swish Prolog (5 min)

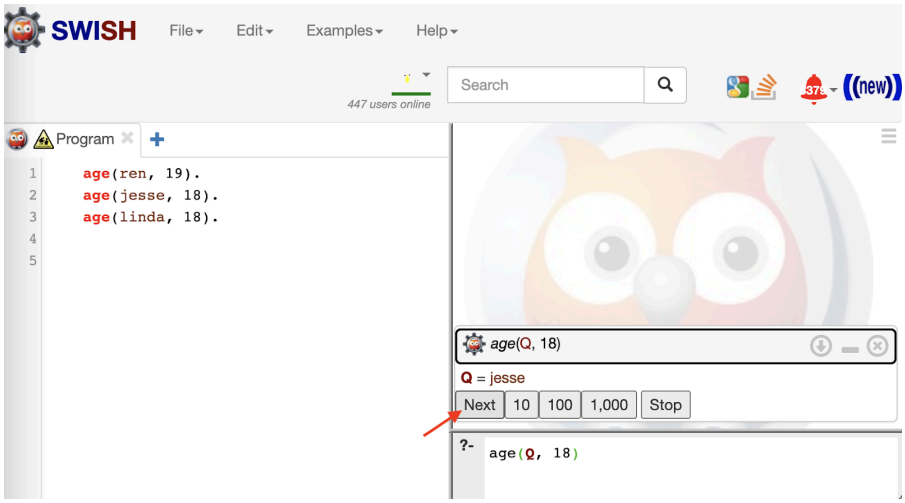
For the following Prolog-related problems, you'll want to use the SWI Prolog website to test out your code:

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)

PRLG2: Facts (10 min)

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.

Part A: (2 min.) color(celery, X)

Your answer:

Part B: (2 min.) color(tomato, orange)

Your answer:

Part C: (2 min.) color(Q, red)

Your answer:

Part D: (4 min.) color(Q, R)

Your answer:

PRLG3: Facts, Rules (10 min)

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).
```

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

Your answer:

Part 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
```

Your answer:

PRLG4: Facts, Rules, Recursion, Transitivity (10 min)

(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

Your answer:

PRLG5: Unification (5 min)

(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)
```

Your answer:

PRLG6: Lists, Recursion (10 min)

(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).
```

Your answer:

PRLG7: Lists, Recursion (10 min)

(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).
```

Your answer:

PRLG8: Lists, Recursion (**Optional**, 10 min)

(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.

Your answer:

PRLG9: Lists, Recursion (15 min)

(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].`

Your answer: