Homework 9 Solutions

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

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

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

Solution:

Inheritance is where one class inherits interfaces, implementations or both from a base class.

Subtype (Subclass) Polymorphism is where I pass a subtype object to a function that expects a supertype (e.g., passing a Dog to a function that accepts Mammals). It's about the typing relationship that allows the subtype object to be substituted for a supertype object.

SP is only used in statically-typed languages, since variables don't have types in dynamically typed languages.

Dynamic dispatch is where a program determines which function to call at runtime, by inspecting the object's type and figuring out the proper function to call. Dynamic dispatch is used in both statically typed languages (for virtual functions) and dynamically typed languages (for duck typing).

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.

Solution:

Subtype polymorphism requires us to use a *super-type variable* to refer to *a subtype object*, as we see here:

```
public void foo(Shape s) {
  System.out.println(s.area());
}
public void bar() {
  Circle c = new Circle(10);
  foo(c);       // uses subtype polymorphism

  Shape s  = c; // also uses subtype polymorphism
```

```
    System.out.println(s.area());
  }
```

Since in dynamically-typed languages variables don't have types, (only values have types) we cannot possibly refer to a subtype object via a supertype variable.

However, we can and must use dynamic dispatch in dynamically-typed languages. Any time we call a method on an object, the language uses dynamic dispatch to determine the proper method to call (using a vtable embedded in the object).

OOP10: SOLID Principles (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?

Solution:

The ElectricVehicle class violates the Dependency Inversion Principle. Rather than having an ElectricVehicle's charge() method directly take a SuperCharger object as its parameter, we should define an interface, e.g.:

```
class ICharger {
public:
  virtual void get_power() = 0;
  virtual double get_max_amps() const = 0;
  virtual double check_price_per_kwh() const = 0;
};
```

Then define our SuperCharger using this interface:

```
class SuperCharger: public ICharger { ... }
```

And finally update our ElectricVehicle class to take a Charger interface rather than a SuperCharger class.

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

This way we could define other chargers, e.g. a CheapCharger and use it to charge our ElectricVehicle too:

```
class CheapCharger: public ICharger { ... }
```

The ElectricVehicle class could also be said to violate the open/closed principle since our ElectricVehicle class is tied to a specific charger - the SuperCharger, and can't work with other chargers should we define them. Migrating to an interface, as we did above, would help solve this problem. Or we could define a Charging base class, and then derive our SuperCharge and CheapCharger from that. Our ElectricVehicle's charge() method would then accept a reference to that Charging base class rather than a specific charger like a SuperCharger.

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?

Solution:

Yes - it does apply. The LSP states that an object of a particular class may be replaced by an object of a subclass without breaking the code that uses those objects. The definition does not require that we have subtype polymorphism. We have superclasses and subclasses in many dynamically-typed languages, and if we pass a subclass object to a function that otherwise works on supertype objects, the function should work as expected.

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.

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.

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.

Solution:

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

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.

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
```

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.

Solution:

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

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

Solution:

```
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
```

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))
```

Solution:

```
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
```

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

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())
```

Solution:

Main start
Foo start
Bar start
Sync call
Main middle
Foo end
Bar end
Main end Foo result Bar result

Here's an explanation, courtesy of ChatGPT:

1. **`main()` starts** and prints `"Main start"`.
2. `foo` is added to the event loop's queue when `asyncio.create_task(foo())` is called. However, `foo` does not start immediately—it is just in the queue.
3. `bar` is added to the event loop's queue when `asyncio.create_task(bar())` is called. Similarly, `bar` does not start immediately—it is just at the end of the queue.
4. **`main()` continues** and hits `await asyncio.sleep(0.5)`. Now, `main()` is suspended for 0.5 seconds, allowing the event loop to start executing the tasks in the queue.
5. The event loop dequeues coroutine **`foo`**, which starts and prints `"Foo start"`. It then reaches `await asyncio.sleep(1)` and will be added to the end of the queue after 1 second expires.
6. Next, the event loop dequeues coroutine **`bar`**, which starts and prints `"Bar start"`. It immediately calls `sync_function()`, which prints `"Sync call"`. Afterward, `bar` reaches `await asyncio.sleep(2)` and will be added to the end of the queue after 2 seconds expire.
7. 0.5 seconds after main issued its `await asyncio.sleep(0.5)` call, **`main()`** is added back to the end of the queue (which is now empty)
8. Next, the event loop dequeues coroutine **`main`**, which starts and prints `"Main middle"`.
9. 1 second after foo issued its `await asyncio.sleep(1)`, the event loop adds **`foo`** back to the end of the queue, as its sleep time has finished.

10. Next, the event loop dequeues coroutine **`foo`**, and `foo` prints `"Foo end"` and completes, returning `"Foo result"`.
11. 1 second after bar issued its `await asyncio.sleep(2)`, the event loop adds **`bar`** back to the end of the queue, as its sleep time has finished.
12. Next, the event loop dequeues coroutine `bar` and `bar` prints `"Bar end"` and completes, returning `"Bar result"`.
13. Finally, the event loop dequeues coroutine **`main`** and runs it, and it prints `"Main end Foo result Bar result"`.

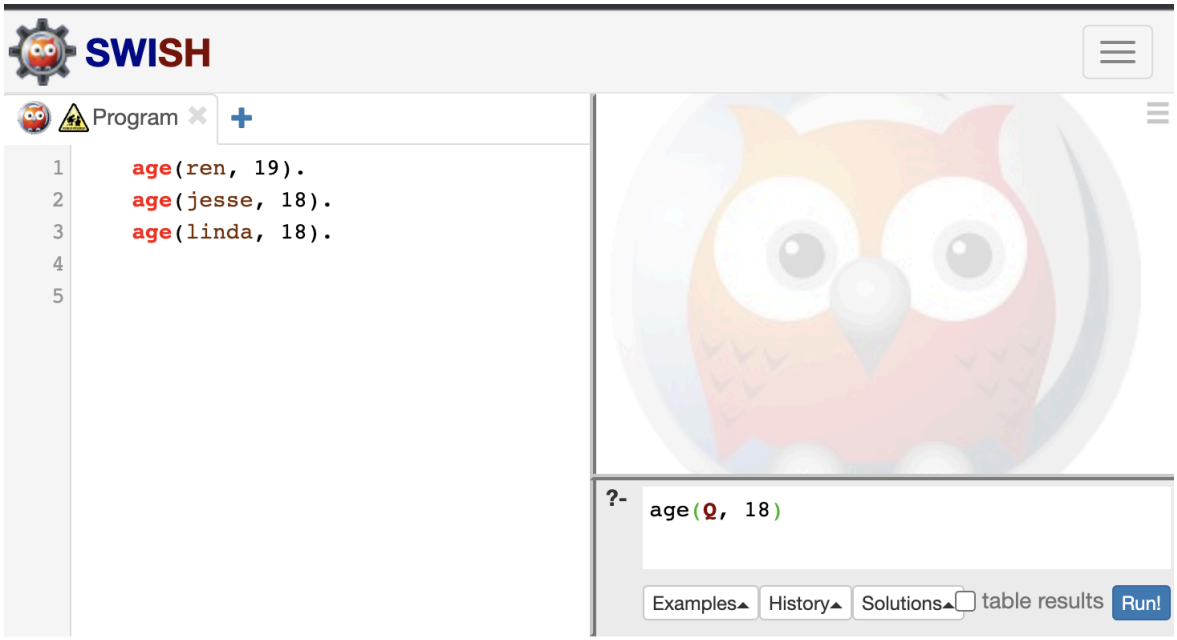
PRLG1: Swish Prolog (5 min)

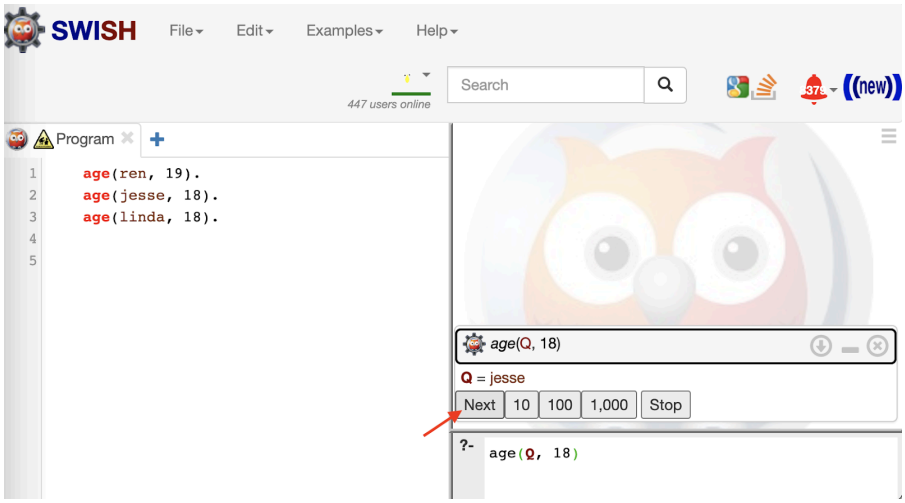
For the following Prolog-related problems, you'll want to use the SWI Prolog website to test out your code:

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)

Solution:

```
X = green
```

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

Solution:

```
false
```

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

Solution:

```
Q = tomato
Q = beet
```

Part 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

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.

Solution:

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

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
```

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).
```

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

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]).
```

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)
```

Solution:

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]

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).
```

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).

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).
```

Solution:

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

PRLG8: Lists, Recursion (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.

Solution:

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

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].`

Solution:

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