Homework 9 - Spring 2023

- Vincent Lin
- [UID REDACTED]
- Section 1D

Question 1

For this question, I'm assuming `&&` has greater operator precedence than `||`.

In general, **short-circuiting** evaluates boolean subexpressions until it's confirmed that the entire expression is satisfied (an OR chain encounters a `true`) or invalidated (an AND chain encounters a `false`).

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

If `e()` returns `true`, `f()`, `g()`, and `h()` are never evaluated, and `do_something()` is evaluated.

If `e()` returns `false` but `f()` and `g()` return `true`, then `h()` is never evaluated, and `do_something()` is evaluated. Furthermore, if `f()` returns `false()`, then `g()` is never evaluated.

Only if `e()` returns `false` and `f() && g()` resolved to `false`, then `h()` is evaluated. `do_something()` is evaluated if `h()` returns `true`.

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

If either `e()` or `f()` returns `true`, then `g()` and `h()` are never evaluated. Furthermore, if `e()` returns `false`, then `f()` is never evaluated.

If `e() && f()` resolved to `false` and `g()` resolved to `false`, the entire expression resolves to `false` and `do_something()` is not evaluated. Otherwise if `g()` evaluates to `true`, `h()` is also evaluated, and only if `h()` returns `true`, then `do_something()` is evaluated.

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

If `e()` returns `false`, the entire expression resolves to `false` and `do_something()` is not evaluated.

Otherwise if `f()` returns `true`, `(f() || g())` resolves to `true`, so `g()` is never evaluated, and since the entire expression resolves to `true`, `do_something()` is never evaluated.

Only if `e()` returns `false` and `f()` returns `false` is `g()` evaluated, and in that case, `do_something()` is evaluated only if `g()` returns `true`.

Question 2

Question 2a

I added this method under `HashTable`:

```
def __iter__(self) -> Iterator:
    def hash_table_generator() -> Generator:
        for head in self.array:
            while head is not None:
                yield head.value
                head = head.next
    return hash_table_generator()
```

NOTE: I took the liberty to define the generator as a closure within `__iter__` such that it has access to the `HashTable` through `self`. I found this approach to better respect encapsulation as opposed to defining a global function that operates on an internal member of `HashTable`.

Question 2b

I defined the iterator class:

```
class HashTableIterator:
    def __init__(self, array: list[Node | None]) -> None:
        self.array = array
        self.bucket = -1
        self.node: Node | None = None
        # Set initial node to the head of the first non-None list.
        self._set_to_next_bucket()

    def _set_to_next_bucket(self) -> None:
        for i in range(self.bucket + 1, len(self.array)):
            if self.array[i] is not None:
                self.bucket = i
                self.node = self.array[i]
                break

    def __next__(self) -> Any:
        # Attempt to move on to the next bucket.
        if self.node is None:
            self._set_to_next_bucket()
        # No more buckets left.
        if self.node is None:
            raise StopIteration
        # Return the current value and advance the current list.
        value = self.node.value
        self.node = self.node.next
        return value
```

```python
    def __iter__(self) -> Iterator:
        return self
```

And updated `HashTable.__iter__` to:

```python
def __iter__(self) -> Iterator:
    return HashTableIterator(self.array)
```

Question 2c

```python
# Initialization code.
table = HashTable(5)
for num in range(20):
    table.insert(num)

# For loop using the iterator interface.
for value in table:
    print(value)
```

Question 2d

`for`-`in` syntax in Python is syntactic sugar for this pattern of code:

```python
iterator = table.__iter__()
while True:
    try:
        value = iterator.__next__()
        print(value)
    except StopIteration:
        break
```

Question 2e

I added this method under `HashTable`:

```python
def forEach(self, callback: Callable[[Any], Any]) -> None:
    for head in self.array:
        if head is None:
            continue
        current = head
        while current is not None:
            callback(current.value)
            current = current.next
```

Question 3

Question 3a

This is a fill-in-the-blank query for matching the functor `color` between atom `celery` and variable `X`.

```
X = green
```

Question 3b

This is a true/false query for matching the functor `color` between atoms `tomato` and `orange`.

```
false
```

Question 3c

This is a fill-in-the-black query for matching the functor `color` with the variable `Q` and atom `red`.

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

`tomato` is returned before `beet` because `color(tomato, red)` is defined before `color(beet, red)`, and Prolog executes from top to bottom.

Question 3d

This is a fill-in-the-blank query for matching the functor `color` with the variables `Q` and `R`. Because the entire parameter list is variables, Prolog will return all combinations of atoms for `color`.

```
Q = celery,
R = green
Q = tomato,
R = red
Q = persimmon,
R = orange
Q = beet,
R = red
Q = lettuce,
R = green
```

Similar to in Question 3c, the order the solutions are returned match the order in which the facts are defined. Note the comma after each `Q =` signifying that the following `R =` line is part of one solution.

Question 4

Question 4a

```
likes_red(P) :- likes(P, F), food(F), color(F, red).
```

In English: "A person `P` likes red if they like a food `F` and the color of that food `F` is red."

Question 4b

```
likes_foods_of_colors_that_menachen_likes(P) :-
    likes(menachen, F1),
    food(F1),
    color(F1, C),
    likes(P, F2),
    food(F2),
    color(F2, C).
```

In English: "A person `P` likes foods of colors that menachen likes if menachin likes a food `F1` with color `C`, and person `P` likes food `F2` with the same color `C`."

Question 5

I assume this is testing **recursive rules**, but I can't quite seem to get the output to match what is given in the question.

```
% Base cases:
reachable(A, B) :- road_between(A, B) ; road_between(B, A).
reachable(A, B) :- road_between(A, X), reachable(X, B).
```

Question 6

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

This unifies. We get the mappping `{X -> bar}`

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

This does NOT unify because the arity does not match.

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

This unifies. We get the mapping `{Z <-> X}`.

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

This unifies. We get the mapping `{X -> barf, Y -> bletch}`.

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

This does NOT unify because the atoms `bletch` and `barf` do not match.

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

This unifies. We get the mapping `{X -> bar, Y -> barf}`.

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

This unifies. We get the mapping `{Y -> bar(a, Z)}`.

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

This does NOT unify because the variable `Z` first gets mapped to `barf`, but then `Z` appears again as the head of the left list, which doesn't match the atom `bletch` in the right list.

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

This unifies. We get the mapping `{Q -> [A,B|C]}`.

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

This does NOT unify because the variable `X` first gets mapped to the atom `a`, but then `X` appears again at the third position of the left list, which doesn't match the list `[a]` in the right list.

Question 7

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

```
% the new value is < all values in list
insert_lex(X,[Y|T],[X,Y|T]) :- X =< Y.

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

In the third definition, we're destructuring the two lists to get rid of the common head (`Y`) such that we can progress closer to the base case(s). Then, the tail of the existing list (`T`) serves as the existing list for the next call, and the "new tail" (`NT`, or "final tail") will serve as the result of the recursive call thus the return value. Kind of confusing.

Question 8

```
% count_elem(List, Accumulator, Total)
% Accumulator must always start at zero
count_elem([], Total, Total).
count_elem([Hd|Tail], Sum, Total) :-
    Sum1 is Sum + 1,
    count_elem(Tail, Sum1, Total).
```

Here we're simply accumulating the sum by passing on `Sum1 = Sum + 1` onto the recursive calls, which process the `Tail` (remaining part) of the list, ultimately saving the final total into `Total` at the base case.

Question 9

```
gen_list(_, 0, []).
gen_list(X, N, [X|XS]) :-
    N1 is N - 1,
    gen_list(X, N1, XS).
```

We first define the base case as a fact - if `N` is 0, we present the empty list. We then use a rule to define the inductive case. At each unification step, we build up the final list by consing `X`, the value we want to repeat, and `XS`, the rest of the list built by recursively unifying `gen_list`.

Question 10

```
append_item([], X, [X]).
append_item([Head|Tail], X, [Head|FinalTail]) :-
    append_item(Tail, X, FinalTail).
```

Similarly, we define the base case as a fact - appending to an empty list results in a list with that singular element. We then use a familiar pattern to implement the inductive case. We want to reach the end of the list to append the item, so we progress towards it by recursing with the tail parts of both the input and output lists, "dropping" the head at each iteration until we hit the base case.