# Mutable(?) data structures

## CS3100 Fall 2019

# Review

## Previously

- Control in Prolog

## This lecture

- Simulating mutable data structure in Prolog.

# Variables in terms

- So far all of our uses of variables have been in queries or rules, but not in terms representing objects.
- Here is a open list which has a prefix of `[a,b]`.

```
?- L = [1,2 | X]
L = [1, 2|X].
```

- We can (pretend to) extend the list `L` by unifying `X` with something else.

```
?- L = [1,2 | X], X = [3 | Y]
L = [1, 2, 3|Y],
X = [3|Y].
```

Such lists are said to be **open lists**.

# Jupyter + Prolog fail!

Jupyter + Prolog is a solution in development (read as does not work as intended).

In [1]:

```
?- L = [1,2 | X].
```

```
X = _1676, L = [ 1, 2 ] .
```

The result should have been `X = _G861, L = [ 1, 2 | X] .`.

We will use the SWI-Prolog interpreter directly for this lecture.

# Queues

We will use open lists to represent queues.

- A queue is represented by `q(L,E)` , where
  - `L` is be an open list
  - `E` is some suffix of L.
- The contents of the queue are the elements in `L` that are not in `E` .

## Enter and Leave

We will use predicates `enter` and `leave` to capture elements entering and leaving the queue.

- `enter(a,Q,R)` : when an element `a` enters the queue `Q` , we get the queue `R` .
- `leave(a,Q,R)` : when an element `a` leaves the queue `Q` , we get the queue `R` .

## Implementing the queues

```
setup(q(X,X)).
leave(A, q(X,Z), q(Y,Z)) :- X = [A | Y].
enter(A, q(X,Y), q(X,Z)) :- Y = [A | Z].
wrapup(q([],[])).
```

Let's try

```
?- setup(Q), enter(0,Q,R).
Q = q([0|_9530], [0|_9530]),
R = q([0|_9530], _9530).
```

- Quite a strange behaviour: Remove `0` from the suffix of `Q` !
  - But as a result, the queue `R` has one element `0` which is not in the suffix.
  - Therefore, the queue `R` has one element `0` .

## Implementing Queues

```
leave(A, q(X,Z), q(Y,Z)) :- X = [A | Y].
```

while leave removes an element from the prefix.

```
enter(A, q(X,Y), q(X,Z)) :- Y = [A | Z].
```

enter removes element from the suffix!

## Working with the queues

```
?- setup(Q), enter(a,Q,R), enter(b,R,S),
   leave(X,S,T), leave(Y,T,U), wrapup(U).
Q = q([a, b], [a, b]),
R = q([a, b], [b]),
S = q([a, b], []),
X = a,
T = q([b], []),
Y = b,
U = q([], []).
```

## Quiz 1

Given

```
?- setup(Q), enter(a,Q,R), enter(b,R,S),
   leave(X,S,T), leave(Y,T,U), wrapup(U).
Q = q([a, b], [a, b]),
R = q([a, b], [b]),
S = q([a, b], []),
X = a,
T = q([b], []),
Y = b,
U = q([], []).
```

What are the lengths of Q, R, S, T, U?

## Quiz 1

Given

```
?- setup(Q), enter(a,Q,R), enter(b,R,S),
   leave(X,S,T), leave(Y,T,U), wrapup(U).
Q = q([a, b], [a, b]),
R = q([a, b], [b]),
S = q([a, b], []),
X = a,
T = q([b], []),
Y = b,
U = q([], []).
```

What are the lengths of Q, R, S, T, U? **0, 1, 2, 1, 0**.

## Deficit queues

Interestingly, the implementation also works where arbitrary elements are first popped and then unfied with elements pushed later.

```
?- setup(Q), leave(X,Q,R), leave(Y,R,S),
   enter(a,S,T), enter(b,T,U), wrapup(U).
Q = q([a, b], [a, b]),
X = a,
R = q([b], [a, b]),
Y = b,
S = q([], [a, b]),
T = q([], [b]),
U = q([], []).
```

# Quiz 2

Given

```
?- setup(Q), leave(X,Q,R), leave(Y,R,S), enter(a,S,T), enter(b,T,U), wrapup
(U).
Q = q([a, b], [a, b]),
X = a,
R = q([b], [a, b]),
Y = b,
S = q([], [a, b]),
T = q([], [b]),
U = q([], []).
```

What is the length of Q, R, S, T, and U?

# Quiz 2

Given

```
?- setup(Q), leave(X,Q,R), leave(Y,R,S), enter(a,S,T), enter(b,T,U), wrapup
(U).
Q = q([a, b], [a, b]),
X = a,
R = q([b], [a, b]),
Y = b,
S = q([], [a, b]),
T = q([], [b]),
U = q([], []).
```

What is the length of Q, R, S, T, and U? **0, -1, -2, -1, 0**

# Quiz 3

What is the result of this query?

```
?- setup(Q), leave(a,Q,R), wrapup(R).
```

1. false.
2. true with some assignments for variables.

## Quiz 3

What is the result of this query?

```
?- setup(Q), leave(a,Q,R), wrapup(R).
```

1. false. ✓
2. true with some assignments for variables.

## Quiz 4

Given

```
setup(s(X,X)).
leave(A, s(X,Z), s(Y,Z)) :- X = [A | Y].
wrapup(q([],[])).
```

what is the `enter` rule for LIFO stack?

1. `enter(A, s(X,Y), s(X,Z)) :- Y = [A | Z]`
2. `enter(A, s(X,Z), s(Y,Z)) :- Y = [A | X]`
3. `enter(A, s(X,Y), s(Y,Z)) :- X = [A | Y]`
4. `enter(A, s(X,Z), s(Z,Y)) :- Y = [A | X]`

## Quiz 4

Given

```
setup(s(X,X)).
leave(A, s(X,Z), s(Y,Z)) :- X = [A | Y].
wrapup(q([],[])).
```

what is the `enter` rule for LIFO stack?

1. `enter(A, s(X,Y), s(X,Z)) :- Y = [A | Z]`
2. `enter(A, s(X,Z), s(Y,Z)) :- Y = [A | X]` ✓
3. `enter(A, s(X,Y), s(Y,Z)) :- X = [A | Y]`
4. `enter(A, s(X,Z), s(Z,Y)) :- Y = [A | X]`

## Simplifying the queue implementation

```
enter(A, q(X,Y), q(X,Z)) :- Y = [A | Z].
leave(A, q(X,Z), q(Y,Z)) :- X = [A | Y].
```

can be simplified to

```
enter(A, q(X,[A | Z]), q(X,Z)).
leave(A, q([A | Y],Z), q(Y,Z)).
```

by pushing the unification into the head of the rule to make it a fact.

# Motivating Difference Lists

Recall the definition of `append` on regular lists

In [2]:

```
append([],Q,Q).
append([H | P], Q, [H | R]) :- append(P,Q,R).
```

```
Added 2 clauses(s).
```

It is easy to see that this `append` is `O(N)` operation, where `N` is the length of the first list.

# Motivating Difference Lists

Given two lists `[1,2,3]` and `[4,5,6]`, we can rewrite them as

```
append(L1,L2,X)
where
L1 = [1,2,3 | []]
L2 = [4,5,6 | []]
```

Instead of having `[]` as the tail, what if we had a variable.

# Motivating Difference Lists

```
append(L1,L2,X)
where
L1 = [1,2,3 | A]
L2 = [4,5,6 | B]
```

- Then, append is really unifying `A` and `L2` to derive the result list `X = [1,2,3,4,5,6 | B]`.
- Now, `append` becomes an `O(1)` operation.
- Such a list representation is known as a **difference list**.

# Reimplementing Append

```
append(L1,S1,L2,S2,L3,S3) :- ...
```

where `Li` is the reference to the list, and `Si` is the reference to the some suffix of the list.

- Similar to queues, the content of each list is the list of all elements in `Li` not in `Si`
  - Hence the name difference list.

# Reimplementing Append

```
append(L1,S1,L2,S2,L3,S3) :- S1 = L2, L1=L3, S2=S3.
```

Pushing the unification into the head of the rule, we get

```
append(L1,L2,L2,S2,L1,S2).
```

Renaming the variables, we get.

```
append(A,B,B,C,A,C).
```

# Convenient notation for difference lists

- We can introduce an infix function symbol `-` to represent difference lists.
  - `A-B` represents a difference list with list `A` with some suffix `B`.
- Whenever you see `A-B`, you should imagine `[...|B]-B`.

Rewriting the append rule

```
append(A-B,B-C,A-C).
```

# Quiz

How should you represent an empty difference list?

1. []
2. []-[]
3. A-A
4. [A]

# Quiz

How should you represent an empty difference list?

1. []
2. []-[]
3. A-A ✓
4. [A]

# Empty difference list representation

```
append(A-B,B-C,A-C)
```

Consider appending onto an empty difference list.

With the empty list represented using `A-A`, we get

```
append(A-A,[1,2,3|C]-C,A-C)
```

The unifications we get are `A = [1,2,3|C]`. Hence the result is just `[1,2,3|C]-C`, which is what we want.

# Empty difference list representation

```
append(A-B,B-C,A-C)
```

OTOH, with the empty list represented using `[]-[]` , we get

```
append([]-[],[1,2,3|C]-C,A-C)
```

which fails to unify since `[]` does not unify with `[1,2,3|C]` .

- It appears that the correct way to encode an empty difference list is `A-A` .
  - But this can cause problems sometimes.

# Unification issues with empty difference list

Consider

```
A-A = [1,2,3|B]-B
```

The second term on LHS, `A` unfies with `B` on RHS. So we get,

```
A-A = [1,2,3|A]-A
```

Now, unfifying `A` with `[1,2,3|A]` , makes `A` an infinite term `[1,2,3 | [1,2,3 | [1,2,3 [...]]]]` .

This is the lack of **occurs check** before unfication in prolog.

# length of difference list.

Length of an ordinary list

```
len([],0).
len([H|T],N) :- len(T,M), N is M+1.
```

We might try to write down the length of a difference list using the same structure:

```
len(A-A,0).
len([_|T]-T1,N) :- len(T-T1,M), N is M+1.
```

# Quiz

What is the length of `len([1,2,3|A]-A,B)` ?

1. A = _, B = 3
2. Error: Arguments not sufficiently instantiated
3. A = infinite term, B = 0
4. false.

# Quiz

What is the length of `len([1,2,3|A]-A,B)` ?

1. A = _, B = 3
2. Error: Arguments not sufficiently instantiated
3. A = infinite term, B = 0 ✓
4. false.

`len([1,2,3 | A]-A, B)` unifies with `len(A-A,B)`.

## Quiz

What is the length of `len([1,2,3|A]-A,B)` ?

1. A = _, B = 3 ✓
2. Error: Arguments not sufficiently instantiated
3. A = infinite term, B = 0 ✓
4. false.

Surprisingly, `A = _, B = 3` is also one of the results.

**Exercise**: Trace by hand.

## Solution 1: Grounding the empty difference list

You can ground the empty difference list by forcing an empty difference list to unify with a pair of empty lists.

```
len2([]-[],0).
len2([_|T]-T1,N) :- len2(T-T1,M), N is M+1.
```

- This gives the right answer for `len2([1,2,3|A]-A,B)`
  - But unifies the tail of the list `A` with `[]` and destroys extensibility.
  - Seemingly pure length function also mutates the list :-(

## Solution 2: occurs check

- Infinite list problem occurs due to `[1,2,3|A]` unifying with `A`.
  - Let us enable occurs check to prevent these terms from unifying.

```
len3(A-A1,0) :- unify_with_occurs_check(A,A1).
len3([_|T]-T1,N) :- len3(T-T1,M), N is M+1.
```

You can also enable occurs_check by default by the query

```
?- set_prolog_flag(occurs_check,true).
```

## Difference list rotation

```
Define a procedure rotate(X,Y) where both X and Y are
represented by difference lists, and Y is formed by
rotating X to the left by one element.
```

## List rotation

```
rotate([H|T],L) :- append(T,[H],L).
```

## Rewrite with difference lists

```
rotate([H|T],R) :- append(T,[H],R).
```

becomes

```
rotate([H|T]-T1,R-S) :- append(T-T1,[H|A]-A,R-S).
```

## Rename the variables

```
rotate([H|T]-T1,R-S) :- append(T-T1,[H|A]-A,R-S).
```

- `append` will unify `T1 = [H|A]`, `T = R` and `A = S`.
  - Apply this renaming.

```
rotate([H|T]-[H|A],T-A) :- append(T-[H|A],[H|A]-A,T-A).
```

## Get rid of append

```
rotate([H|T]-[H|A],T-A) :- append(T-[H|A],[H|A]-A,T-A).
```

- Observe that the `append` is redundant
  - When this append succeeds, no new unifications are obtained.
  - Remove it to get

```
rotate([H|T]-[H|A],T-A).
```

## Testing Rotate

```
?- rotate([1,2,3|A]-A,R).
A = [1|_12344],
R = [2, 3, 1|_12344]-_12344.
```

## Fin.