Logical Languages

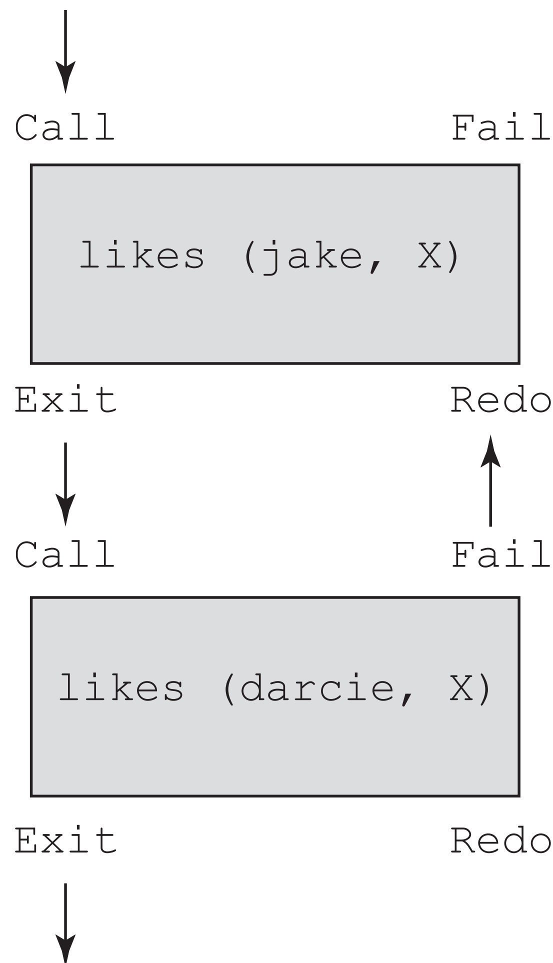
part 5

2020

Instructor: Odelia Schwartz

Prolog

Control flow model for
likes(jake,X), likes(darcie,X)



- Four parts for each subgoal
- Can enter goal through call (forward) or redo (backward)
- Can exit through fail or exit

Prolog

- **append**

```
append([], List, List)
append ( [Head | List_1], List_2, [Head | List_3]) :-
append(List_1, List_2, List_3)
```

```
[trace]  ?- append([bob,jo],[jake, darcie],Family).
   Call: (6) append([bob, jo], [jake, darcie], _G1110) ? creep
   Call: (7) append([jo], [jake, darcie], _G1189) ? creep
   Call: (8) append([], [jake, darcie], _G1192) ? creep
   Exit: (8) append([], [jake, darcie], [jake, darcie]) ? creep
   Exit: (7) append([jo], [jake, darcie], [jo, jake, darcie]) ? creep
   Exit: (6) append([bob, jo], [jake, darcie], [bob, jo, jake, darcie]) ? creep
Family = [bob, jo, jake, darcie].
```

Prolog

Prolog append more flexible than Scheme/ML!

```
append([], List, List)
append ( [Head | List_1], List_2, [Head | List_3]) :-
append(List_1, List_2, List_3)
```

Let's try:

- append(X,Y,[a,b,c]).

Returns:

X = []
Y = [a,b,c]

X = [a]
Y = [b,c]

X = [a,b]
Y = [c]

X = [a,b,c]
Y = []

Prolog

- **reverse**

```
reverse([],[])
reverse ([Head|Tail], List) :-
reverse (Tail, Result) , append(Result,[Head],List)
```

Prolog

- **reverse**

```
reverse([],[])
reverse ([Head|Tail], List) :-
reverse (Tail, Result) , append(Result,[Head],List)
```

Run in Prolog compiler reverse([a,b,c],Q).

Prolog

- **reverse**

```
[trace]  ?- reverse([a,b,c],Q).
   Call: (6) reverse([a, b, c], _G1106) ? creep
   Call: (7) reverse([b, c], _G1188) ? creep
   Call: (8) reverse([c], _G1188) ? creep
   Call: (9) reverse([], _G1188) ? creep
   Exit: (9) reverse([], []) ? creep
   Call: (9) lists:append([], [c], _G1192) ? creep
   Exit: (9) lists:append([], [c], [c]) ? creep
   Exit: (8) reverse([c], [c]) ? creep
   Call: (8) lists:append([c], [b], _G1195) ? creep
   Exit: (8) lists:append([c], [b], [c, b]) ? creep
   Exit: (7) reverse([b, c], [c, b]) ? creep
   Call: (7) lists:append([c, b], [a], _G1106) ? creep
   Exit: (7) lists:append([c, b], [a], [c, b, a]) ? creep
   Exit: (6) reverse([a, b, c], [c, b, a]) ? creep
Q = [c, b, a].
```

```
reverse([],[])
reverse ([Head|Tail], List) :-
reverse (Tail, Result) , append(Result,[Head],List)
```

Prolog

- Another list operation: member

Prolog

- Another list operation: member
- Remember assignment hint: you can use this built in function

Prolog

- Another list operation: member
- Remember assignment hint: you can use this built in function
- Here we will go through the code

Prolog

- Member: Is Element a member of List? Start from base condition:

Main idea: If Element is in the head of the list, then yes, it is a member (true); otherwise recurse on the tail…

Prolog

- Member: Is Element a member of List? Start from base condition:

Scheme…

```
(define (member atm lis)
  (cond
    ((null? lis) #f)
    ((eq? atm (car lis)) #t)
    (else (member atm (cdr lis)))
  )
)
```

Prolog

- Member: Is Element a member of List? Start from base condition:

Member(Element, [Element | _]).

Prolog

- Member: Is Element a member of List? Start from base condition:

```
Member( Element, [Element | _ ] ).
```

anonymous variable;
do not care what instantiation
it gets from unification

Prolog

- Member: Is Element a member of List? Start from base condition:

Member(Element, [Element | _]).

anonymous variable;
do not care what instantiation it gets from unification

also remember: | separates the head and tail. So tail can be anything.

Prolog

- Member: Is Element a member of List? Start from base condition:

Member(Element, [Element | _]).

anonymous variable;
do not care what instantiation
it gets from unification

also remember | separates
the head and tail. So tail
can be anything.

What does this mean then?

Prolog

- Member: Is Element a member of List? Start from base condition:

Member(Element, [Element | _]).

anonymous variable;
do not care what instantiation
it gets from unification

Base statement succeeds if Element is head of list (either initially or after several recursions). In that case, returns true

Prolog

- Member: Is Element a member of List?

```
Member( Element, [Element | _ ] ).

Member(Element, [_ | List] ) :- member(Element, List)
```

Prolog

- Member: Is Element a member of List?

```
Member( Element, [Element | _ ] ).

Member(Element, [_ | List] ) :- member(Element, List)
```

Meaning?

Prolog

- Member: Is Element a member of List?

```
Member( Element, [Element | _ ] ).

Member(Element, [_ | List] ) :- member(Element, List)
```

If Element is in the tail of the list, then recurse on the tail of the list (head can be anything, as in the _ symbol !)

Prolog

- Member: Is Element a member of List?

```
Member( Element, [Element | _ ] ).
Member(Element, [_ | List] ) :- member(Element, List).
```

Overall:

First statement base condition: If Element is in the head of the list, succeeds (true)

Recursion: If Element is in the tail of the list, then recurse on the tail of the list

Prolog

- Member: let’s do trace on Prolog.

```
trace.
member(a,[b,a,c]).
```



```
Member( Element, [Element | _ ] ).
Member(Element, [_ | List] ) :- member(Element, List).
```

Prolog

- Member: let's do trace on Prolog.

```
trace.
member(a,[b,a,c]).

[trace]  ?- member(a,[b,a,c]).
   Call: (6) member(a, [b, a, c]) ? creep
   Call: (7) member(a, [a, c]) ? creep
```

Recurse on tail


```
Member( Element, [Element | _ ] ).
Member(Element, [_ | List] ) :- member(Element, List).
```

Prolog

- Member: let’s do trace on Prolog.

```
trace.
member(a,[b,a,c]).

[trace]  ?- member(a,[b,a,c]).
   Call: (6) member(a, [b, a, c]) ? creep
   Call: (7) member(a, [a, c]) ? creep
   Exit: (7) member(a, [a, c]) ? creep
```

First statement true

```
Member( Element, [Element | _ ] ).
Member(Element, [_ | List] ) :- member(Element, List).
```

Prolog

- Member: let's do trace on Prolog.

```
trace.
member(a,[b,a,c]).

[trace]  ?- member(a,[b,a,c]).
   Call: (6) member(a, [b, a, c]) ? creep
   Call: (7) member(a, [a, c]) ? creep
   Exit: (7) member(a, [a, c]) ? creep
   Exit: (6) member(a, [b, a, c]) ? creep
```

Putting back head of implication


```
Member( Element, [Element | _ ] ).
Member(Element, [_ | List] ) :- member(Element, List).
```

Prolog

- Member: let's do trace on Prolog.

```
trace.
member(a,[b,a,c]).

[trace]  ?- member(a,[b,a,c]).
   Call: (6) member(a, [b, a, c]) ? creep
   Call: (7) member(a, [a, c]) ? creep
   Exit: (7) member(a, [a, c]) ? creep
   Exit: (6) member(a, [b, a, c]) ? creep
true .
```

Returns true


```
Member( Element, [Element | _ ] ).
Member(Element, [_ | List] ) :- member(Element, List).
```

Prolog

- Member: let’s do trace on Prolog.

```
trace.
member(a,[b,c,d]).
```



```
Member( Element, [Element | _ ] ).
Member(Element, [_ | List] ) :- member(Element, List).
```

Prolog

- Member: let’s do trace on Prolog.

```
?- trace.
[trace]  ?- member(a,[b,c,d]).
   Call: (6) member(a, [b, c, d]) ? creep
   Call: (7) member(a, [c, d]) ? creep
   Call: (8) member(a, [d]) ? creep
   Call: (9) member(a, []) ? creep
```

recursion on tail


```
Member( Element, [Element | _ ] ).
Member(Element, [_ | List] ) :- member(Element, List).
```

Prolog

- Member: let’s do trace on Prolog.

```
?- trace.
[trace]  ?- member(a,[b,c,d]).
   Call: (6) member(a, [b, c, d]) ? creep
   Call: (7) member(a, [c, d]) ? creep
   Call: (8) member(a, [d]) ? creep
   Call: (9) member(a, []) ? creep
   Fail: (9) member(a, []) ? creep
```

Failure first statement head match


```
Member( Element, [Element | _ ] ).
Member(Element, [_ | List] ) :- member(Element, List).
```

Prolog

- Member: let’s do trace on Prolog.

```
?- trace.
[trace]  ?- member(a,[b,c,d]).
   Call: (6) member(a, [b, c, d]) ? creep
   Call: (7) member(a, [c, d]) ? creep
   Call: (8) member(a, [d]) ? creep
   Call: (9) member(a, []) ? creep
   Fail: (9) member(a, []) ? creep
   Fail: (8) member(a, [d]) ? creep
   Fail: (7) member(a, [c, d]) ? creep
   Fail: (6) member(a, [b, c, d]) ? creep
false.
```

Failures

```
Member( Element, [Element | _ ] ).
Member(Element, [_ | List] ) :- member(Element, List).
```

Prolog

- Member: let’s do trace on Prolog.

```
?- member(X,[a,b,c]).
```

Answer?


```
Member( Element, [Element | _ ] ).
Member(Element, [_ | List] ) :- member(Element, List).
```

Prolog

- Member: let’s do trace on Prolog.

```
?- member(X,[a,b,c]).
```

Answer?

X = a ;
X = b ;
X = c ;


```
Member( Element, [Element | _ ] ).
Member(Element, [_ | List] ) :- member(Element, List).
```

Prolog

- Assignment hint

For the union of two lists, you need to consider what happens if the head of the first list is a member of the second list, and what happens when it is not. You are allowed to use the built in Prolog member function.

First, consider the base of the recursion.
Then, for the actual recursion: consider for List1, List2, List3, what happens when the right side of the implication includes member(H, List2) and union(List1, List2, List3) (what is the left side of the implication?).
Then consider what the implication should be when H is not a member of List2.

Prolog

- Issues

Create a file not.pl with:

```
parent(bill, jake).
parent(bill, shelley).
sibling(X,Y) :- parent(M,X), parent(M,Y)
```

Run in Prolog sibling(X,Y)

Returns?

Prolog

- Issues

File not.pl :
```
parent(bill, jake).
parent(bill, shelley).
sibling(X,Y) :- parent(M,X), parent(M,Y)
```

```
?- sibling(X,Y).
X = Y, Y = jake ;
X = jake,
Y = shelley ;
X = shelley,
Y = jake ;
X = Y, Y = shelley.
```

Prolog

- Issues

File not.pl :

```
parent(bill, jake).
parent(bill, shelley).
sibling(X,Y) :- parent(M,X), parent(M,Y)
```

```
?- sibling(X,Y).
X = Y, Y = jake ;
X = jake,
Y = shelley ;
X = shelley,
Y = jake ;
X = Y, Y = shelley.
```

What is strange here?

Prolog

- Issues

File not.pl :

```
parent(bill, jake).
parent(bill, shelley).
sibling(X,Y) :- parent(M,X), parent(M,Y)
```

What is strange here?

```
?- sibling(X,Y).
X = Y, Y = jake ;
X = jake,
Y = shelley ;
X = shelley,
Y = jake ;
X = Y, Y = shelley.
```

Prolog

- Issues

File not.pl :

```
parent(bill, jake).
parent(bill, shelley).
sibling(X,Y) :- parent(M,X), parent(M,Y)
```

What is strange here?

Prolog “thinks” Jake is sibling of self!

Prolog

- Issues

File not.pl :

```
parent(bill, jake).
parent(bill, shelley).
sibling(X,Y) :- parent(M,X), parent(M,Y)
```

```
[trace]  ?- sibling(X,Y).
   Call: (6) sibling(_G1096, _G1097) ? creep
   Call: (7) parent(_G1181, _G1096) ? creep
   Exit: (7) parent(bill, jake) ? creep
   Call: (7) parent(bill, _G1097) ? creep
   Exit: (7) parent(bill, jake) ? creep
   Exit: (6) sibling(jake, jake) ? creep
X = Y, Y = jake
```

Prolog

- Issues

```
File not.pl :
parent(bill, jake).
parent(bill, shelley).
sibling(X,Y) :- parent(M,X), parent(M,Y)
```

Why?
System instantiates M with bill, X with jake (first subgoal, parent(M,X)) yielding true.
Then for second subgoal (parent(M,Y)) starts again at beginning of database and instantiates Y to jake, yielding true.

Prolog

- Issues

File not.pl :

```
parent(bill, jake).
parent(bill, shelley).
sibling(X,Y) :- parent(M,X), parent(M,Y)
```

Why?

System instantiates M with bill, X with jake (first subgoal, parent(M,X)) yielding true.

Then for second subgoal (parent(M,Y)) starts again at beginning of database and instantiates Y to jake, yielding true.

How to fix?

Prolog

- Issues

File not2.pl :

```
parent(bill, jake).
parent(bill, shelley).
sibling(X,Y) :- parent(M,X), parent(M,Y), not(X=Y).
```

Prolog

- Issues

File not2.pl :

```
parent(bill, jake).
parent(bill, shelley).
sibling(X,Y) :- parent(M,X), parent(M,Y), not(X=Y).
```

Run in Prolog sibling(X,Y)

Returns?

Prolog

- Issues

File not2.pl :

```
parent(bill, jake).
parent(bill, shelley).
sibling(X,Y) :- parent(M,X), parent(M,Y), not(X=Y).
```

Run in Prolog sibling(X,Y)

Returns?

Prolog

- Issues

File not2.pl :

```
parent(bill, jake).
parent(bill, shelley).
sibling(X,Y) :- parent(M,X), parent(M,Y), not(X=Y).
```

```
?- sibling(X,Y).
X = jake,
Y = shelley ;
X = shelley,
Y = jake ;
```

Prolog

- Issues

File not2.pl :

```
parent(bill, jake).
parent(bill, shelley).
sibling(X,Y) :- parent(M,X), parent(M,Y), not(X=Y).
```

Try trace…

Prolog

- Issues

File not2.pl :

```
parent(bill, jake).
parent(bill, shelley).
sibling(X,Y) :- parent(M,X), parent(M,Y), not(X=Y).
```

```
[trace]  ?- sibling(X,Y).
   Call: (6) sibling(_G1096, _G1097) ? creep
   Call: (7) parent(_G1181, _G1096) ? creep
   Exit: (7) parent(bill, jake) ? creep
   Call: (7) parent(bill, _G1097) ? creep
   Exit: (7) parent(bill, jake) ? creep
^  Call: (7) not(jake=jake) ? creep
^  Fail: (7) not(user: (jake=jake)) ? creep
   Redo: (7) parent(bill, _G1097) ? creep
   Exit: (7) parent(bill, shelley) ? creep
^  Call: (7) not(jake=shelley) ? creep
^  Exit: (7) not(user: (jake=shelley)) ? creep
   Exit: (6) sibling(jake, shelley) ? creep
X = jake,
Y = shelley .
```

Prolog

Questions?