Logical Languages

part 4

2020

Instructor: Odelia Schwartz

Prolog

To access the lab computers, ssh into johnston and then ssh into one of the host computers in the lab. To see what hosts are available type in the johnston command line cat ~irina/hostnames

Prolog

:- implies symbol
, and symbol

- Right side implies left side
  Right side can have and

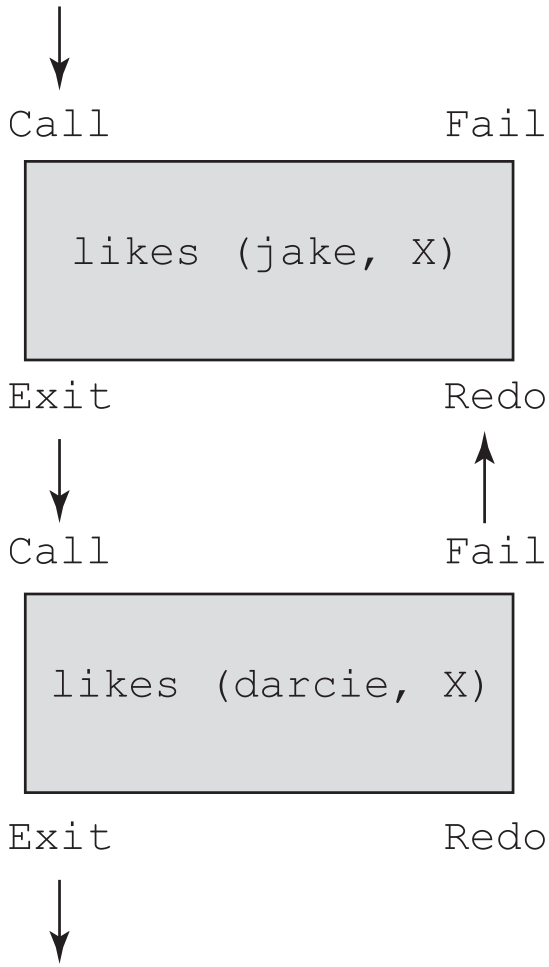
- Headless or headed

- Facts
  Rules
  Goals/Queries

- Variables: start with capital letter

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

List structure

- Lists can be created by a proposition:
  new_list([apple, prune, grape, kumquat]).

- This states that the constant list [apple, prune, grape, kumquat] is a new element of the relation name new_list (a name we just made up).

- Does a similar thing to male(jake) …
  It states that [apple, prune, grape, kumquat] is a new element of new_list

- So we can also have a second statement
  new_list([apricot, peach, pear)].

Prolog

lists_simple.pl

```
new_list([apple,prune,grape,kumquot]).
new_list([apricot,peach,pear]).
```

Prolog

lists_simple.pl

```
new_list([apple,prune,grape,kumquot]).
new_list([apricot,peach,pear]).
```

Run in compiler:

```
new_list(X).
new_list([X|Y]).
```

use ; after entering.

Returns?

Prolog

lists_simple.pl

```
new_list([apple,prune,grape,kumquot]).
new_list([apricot,peach,pear]).
```

Run in compiler:

```
new_list(X).
?- new_list(X).
X = [apple, prune, grape, kumquot] ;
X = [apricot, peach, pear].
```

Prolog

lists_simple.pl

```
new_list([apple,prune,grape,kumquot]).
new_list([apricot,peach,pear]).
```

Run in compiler:

```
new_list([X|Y]).
```

```
?- new_list([X|Y]).
X = apple,
Y = [prune, grape, kumquot] ;
X = apricot,
Y = [peach, pear].
```

Returns the head and tail of each list!

Prolog

- The | notation can both dismantle and construct lists
- We saw dismantling into a head and tail

Prolog

- The | notation can both dismantle and construct lists
- We saw dismantling into a head and tail
- But we can also construct:
  [pickle, [peanut, prune, popcorn]]

  creates [pickle, peanut, prune, popcorn]

Prolog

- The | notation can both dismantle and construct lists
- We saw dismantling into a head and tail
- But we can also construct:
  [pickle, [peanut, prune, popcorn]]

  creates [pickle, peanut, prune, popcorn]

These are all equivalent!

[apricot, peach, pear | []]
[apricot, peach | [pear]]
[apricot | [peach, pear]]

Prolog

- File lists_simple4.pl

```
% run in compiler:
% new_list(X).
% use ; after entering.

new_list([apricot,peach,pear | []]).
new_list([apricot,peach | [pear]]).
new_list([apricot | [peach,pear]]).
```

In compiler:

```
?- new_list(X).
X = [apricot, peach, pear] ;
X = [apricot, peach, pear] ;
X = [apricot, peach, pear].
```

Prolog

- File lists_simple2.pl

```
new_list([H|T], H, T).
```

What does this do??

```
?- new_list([apple,prune,grape,kumquot],X,Y).
X = apple,
Y = [prune, grape, kumquot].
```

Returns head and tail

```
?- new_list(X,apple,[prune, grape, kumquot]).
X = [apple, prune, grape, kumquot].
```

Constructs list

Prolog

- File lists_simple2.pl

```
new_list([H|T], H, T).
```

```
?- new_list([apple,prune,grape,kumquot],prune,
[prune, grape, kumquot]).
```

Returns?? false.

Prolog

- Append function

Prolog

- Append function
- Similar to ML conceptually…

Prolog

- Append function
- Similar to ML conceptually…
- But here recursion controlled by resolution process!

Prolog

- Append function
- Similar to ML conceptually…
- But here recursion controlled by resolution process!

Prolog

Append function: base case

- append([], List, List)

Lists to be appended

result

Prolog

Append function: base case

- append([], List, List)

Lists to be appended

result

Note how this is different than we are used to with “functions”: The first two “parameters” are the lists we are appending and the last is the result

Prolog

Append function: base case

- append([], List, List)

Lists to be appended

result

Note how this is different than we are used to with "functions": The first two "parameters" are the lists we are appending and the last is the result

Reads as??

Prolog

Append function: base case

- append([], List, List)

Lists to be appended — result

Note how this is different than we are used to with "functions": The first two "parameters" are the lists we are appending and the last is the result

When the empty list is appended to any other list, the other list is the result. This is the terminating case of the recursion

Prolog

Append function: base case

- append([], List, List)

Lists to be appended   result

Note how this is different than we are used to with “functions”: The first two “parameters” are the lists we are appending and the last is the result

When the empty list is appended to any other list, the other list is the result. This is the terminating case of the recursion

Includes pattern matching

Prolog

Append function:

```
append([], List, List)
```



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

Prolog

Append function:

```
append([], List, List)
```

Lists to be appended — result

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

Let's unpack this!

Prolog

Append function:

```
append([], List, List)
```

Lists to be appended — result

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

Let’s unpack this!

Prolog

Append function recursion:

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

Recursion is through the implication

Append function recursion:

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

Recursion is through the implication: right side (here bottom) implies left side (here top)

Append function recursion:

To be appended   Result

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

append(List_1, List_2, List_3)

In the recursive step, this implication is essentially adding The same **Head** to the first given list, and to the resulting third new list

Append function recursion:

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

append(List_1, List_2, List_3)

So first element of the new list is the same as the first element of the first given list (both named **Head**)

Append function recursion:

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

Right side of implication (here bottom):

The tail of the first given list (List_1) has the second list (List_2) appended to form the tail of the resulting list (List_3)

Append function recursion:

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

Right side of implication (here bottom):

The tail of the first given list (List_1) has the second list (List_2) appended to form the tail of the resulting list (List_3)

Left side of implication (here top):

The Head is then added on the left side of implication

Example append (one step of recursion):

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

List_1 = [b,c]
List_2 = [e,f]
List_3 = [b,c,e,f] (note it is appending List_1 and List_2)
Head = a

Prolog

Example append (one step of recursion):

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

List_1 = [b,c]
List_2 = [e,f]
List_3 = [b,c,e,f] (note it is appending List_1 and List_2)
Head = a

[**Head** | List_1] = [a,b,c]
List_2 = [e,f]
[**Head** | List_1] = [a,b,c,e,f]
(note it is appending first two lists)

Example append (one step of recursion):

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

List_1 = [b,c]
List_2 = [e,f]
List_3 = [b,c,e,f] (note it is appending List_1 and List_2)
Head = a

[**Head** | List_1] = [a,b,c]
List_2 = [e,f]
[**Head** | List_1] = [a,b,c,e,f]
(note it is appending first two lists)

Example append (one step of recursion):

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

List_1 = [b,c]
List_2 = [e,f]
List_3 = [b,c,e,f] (note it is appending List_1 and List_2)
Head = a

[**Head** | List_1] = [a,b,c]
List_2 = [e,f]

Example append (one step of recursion):

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

List_1 = [b,c]
List_2 = [e,f]
List_3 = [b,c,e,f] (note it is appending List_1 and List_2)
Head = a

[**Head** | List_1] = [a,b,c]
List_2 = [e,f]
[**Head** | List_1] = [a,b,c,e,f]
(note it is appending first two lists)

Append function recursion:

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

append(List_1, List_2, List_3)

Summary of recursion:
In the recursive step, this implication is essentially adding
The same **Head** to the first given list, and to the
resulting third new list

Prolog

Append full function:

append([], List, List)

Lists to be appended — result

When empty list appended to any other List, the other List is the result

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

append(List_1, List_2, List_3)

In the recursive step, this implication is essentially adding The same **Head** to the first given list, and to the resulting third new list

Prolog

Append in ML reminder: Base case:

fun append ([],lis2) = lis2

Lists to be appended   result

Prolog:

append([], List, List)

Lists to be appended   result

Append in ML reminder:

fun append ([],lis2) = lis2

| append(**h**::t,lis2) = **h**::append(t,lis2);

Adding head to first list

Adding head to new list

Append in ML reminder:

Base condition result

fun append ([],lis2) = lis2

result

| append(**h**::t,lis2) = **h**::append(t,lis2);

Adding head to first list

Adding head to new list (was third list in Prolog)

Prolog:

Base condition result

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

result

Prolog

Append in ML reminder:

```
fun append ([],lis2) = lis2
| append(h::t,lis2) = h::append(t,lis2);
```

Scheme...

(cons (**car list1**) (append (cdr list1) list2))

Head of first   append remaining of first with second

Prolog:

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

Prolog

Append full function:

append([], List, List)

Lists to be appended

result

When empty list appended to any other List, the other List is the result

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

append(List_1, List_2, List_3)

In the recursive step, this implication is essentially adding The same **Head** to the first given list, and to the resulting third new list

Prolog

How the recursion works:

append([], List, List)

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

In both Prolog and ML/Scheme, the resulting list is not constructed until the recursion produces the terminating condition; in this case the first list becomes empty, after each time taking out its head (essentially keeps recursing on right side of second proposition, each time taking out head).

How the recursion works:

append([], List, List)

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

In both Prolog and ML/Scheme, the resulting list is not constructed until the recursion produces the terminating condition; in this case the first list becomes empty, after each time taking out its head (essentially keeps recursing on right side of second proposition, each time taking out head).

Then list is built using append function and on left side of proposition, elements from the first list added one at a time as the head

Prolog

How the recursion works:

```
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).
```

What is Family?

Prolog

How the recursion works:

```
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).
```

Variable representing third list

How the recursion works:

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) ?
```

Start from goal on left side of second statement, **and create recursion via right side of second statement, each time taking out head...**

How the recursion works:

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) ?

Keep taking head out…

How the recursion works:

```
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) ?
```

Keep taking head out…
Now what?

Prolog

How the recursion works:

```
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) ?
```

Keep taking head out…
Now what?
Matches first (base) proposition!

How the recursion works:

```
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]) ?
```

Keep taking head out…
Now what?
Matches first (base) proposition!

How the recursion works:

```
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]) ?
```

Element removed from first list appended to resulting list Family, by left side of implication succeeding

How the recursion works:

```
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]) ?
```

Keep adding back the heads…

Prolog

How the recursion works:

```
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].
```

Exit of initial goal accomplished!

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?

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 = []

Using append to create other list operations

list_op2 ([],[])   Base: Empty list returns itself

Prolog

Using append to create other list operations

```
list_op2 ([],[])
list_op2 ([Head|Tail], List) :-
list_op2 (Tail, Result) , append(Result,[Head],List)
```

Result appended with
[Head] gives list

What is this doing?
Try it with Prolog and a list

Using append to create other list operations

```
list_op2 ([],[])
list_op2 ([Head|Tail], List) :-
list_op2 (Tail, Result) , append(Result,[Head],List)
```

Result appended with [Head] gives list

What is this doing?

Tail and Result are reversed…

Intuitive example for recursion

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

Original list: [apple, orange, grape]

2nd statement right side: Head = apple
Tail = [orange, grape]
Result = [grape, orange]

Result reversed from Tail according to first part of right side of proposition

Intuitive example for recursion

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

Original list: [apple, orange, grape]

2nd statement right side: Head = apple
Tail = [orange, grape]
Result = [grape, orange]

Now according to second part of right side of proposition,
append(Result,[Head],List) = [grape, orange, apple]

Intuitive example for recursion

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

Original list: [apple, orange, grape]

2nd statement right side: Head = apple
Tail = [orange, grape]
Result = [grape, orange]

Now according to second part of right side of proposition,
append(Result,[Head],List) = [grape, orange, apple]

This implies left side of second proposition:

Intuitive example for recursion

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

Original list: [apple, orange, grape]

2nd statement right side: Head = apple
Tail = [orange, grape]
Result = [grape, orange]

Now according to second part of right side of proposition,
append(Result,[Head],List) = [grape, orange, apple]

This implies left side of second proposition:
reverse ([Head|Tail], List) = ?

Prolog

Intuitive example for recursion

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

Original list: [apple, orange, grape]

2nd statement right side: Head = apple
Tail = [orange, grape]
Result = [grape, orange]
Now according to second part of right side of proposition,
append(Result,[Head],List) = [grape, orange, apple]

This implies left side of second proposition:
reverse ([Head|Tail], List) = ?
[Head|Tail] = [apple, orange, grape]

Intuitive example for recursion

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

Original list: [apple, orange, grape]

2nd statement right side: Head = apple
Tail = [orange, grape]
Result = [grape, orange]

Now according to second part of right side of proposition,
append(Result,[Head],List) = [grape, orange, apple]

This implies left side of second proposition:
reverse ([Head|Tail], List) = ?
[Head|Tail] = [apple, orange, grape] reversed from
List = [grape, orange, apple]

Try reverse in compiler with trace

```
[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
```

Each time take out head

Try reverse in compiler with trace

```
[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
```

Each time take out head per first subgoal right side of imply

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

Prolog

Try reverse in compiler with trace

```
[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
```

Resolved by base condition

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

Try reverse in compiler with trace

```
[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
```

Added head that was taken out

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

Try reverse in compiler with trace

```
[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
```

Added head that
was taken out
So reached resolution
and can exit append

List=[c]
Tail=[]
Head=[c]
[Head|Tail]=[c]

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

Try reverse in compiler with trace

```
[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
```

Reached resolution of reverse following resolution of the two subgoals

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

Try reverse in compiler with trace

```
[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
```

Keep adding head that was taken out

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

Try reverse in compiler with trace

```
[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
```

and resolving append,
then reversing

List=[c,b]
Tail=[c]
Head=[b]
[Head|Tail]=[b,c]

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

Prolog

Try reverse in compiler with trace

```
[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].
```

and one more time
add head, resolve
append, and then reverse

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