Lists, Recursion, and Abstract Data Types in Prolog

Dr. Demetrios Glinos
University of Central Florida

CAP4630 – Artificial Intelligence

Today

- Lists
- Recursion
- Abstract Data Types

List

- a data structure for an ordered set of elements
- elements can be other lists
- enclosed by square brackets, elements are comma-separated
- Examples:

  [1, 2, 3, 4]
  [tom, dick, harry]
  [tom, [dick, harry]]
  [ ]

Head and Tail

- Bar operator "|" separates head (an *element*) and tail (a *list*) of a list
- Example: for [tom, dick, harry], head is tom, tail is [dick, harry]
- List matching examples for [tom, dick, harry, fred]

  if matched to [X | Y] then X = tom, Y = [dick, harry, fred]
  if matched to [X, Y | Z] then X = tom, Y = dick, Z = [harry, fred]
  if matched to [X, Y, Z | W] then X = tom, Y = dick, Z = harry, W = [fred]
  if matched to [W, X, Y, Z | V] then W = tom, X = dick, Y = harry, Z = fred, V = [ ]
  will not match [V, W, X, Y, Z | U]
  will match [tom, X | [harry, fred]], giving X = dick

- Can also ***build*** lists using the bar operator

  for X = tom and Y = [dick], then L = [X | Y] will be bound to [tom, dick]

Recursive List Processing

- Prolog has a **member/2** predicate
  - tests whether the first argument is a member of the second
  - e.g., member(a, [a, b, c, d, e]). produces the result "yes"

- How can we define this function recursively?
  - first, check if the desired constant is the ***first*** element of the list

    ```
    member( X, [ X | _ ] ).
    ```

  - if not, then check if it is in the ***rest*** of the list (by stripping off the first element)

    ```
    member( X, [ _ | T ] ) :- member( X, T ).
    ```

  - **important:** terminating condition must appear before the recursion

member1.pl (test with trace)

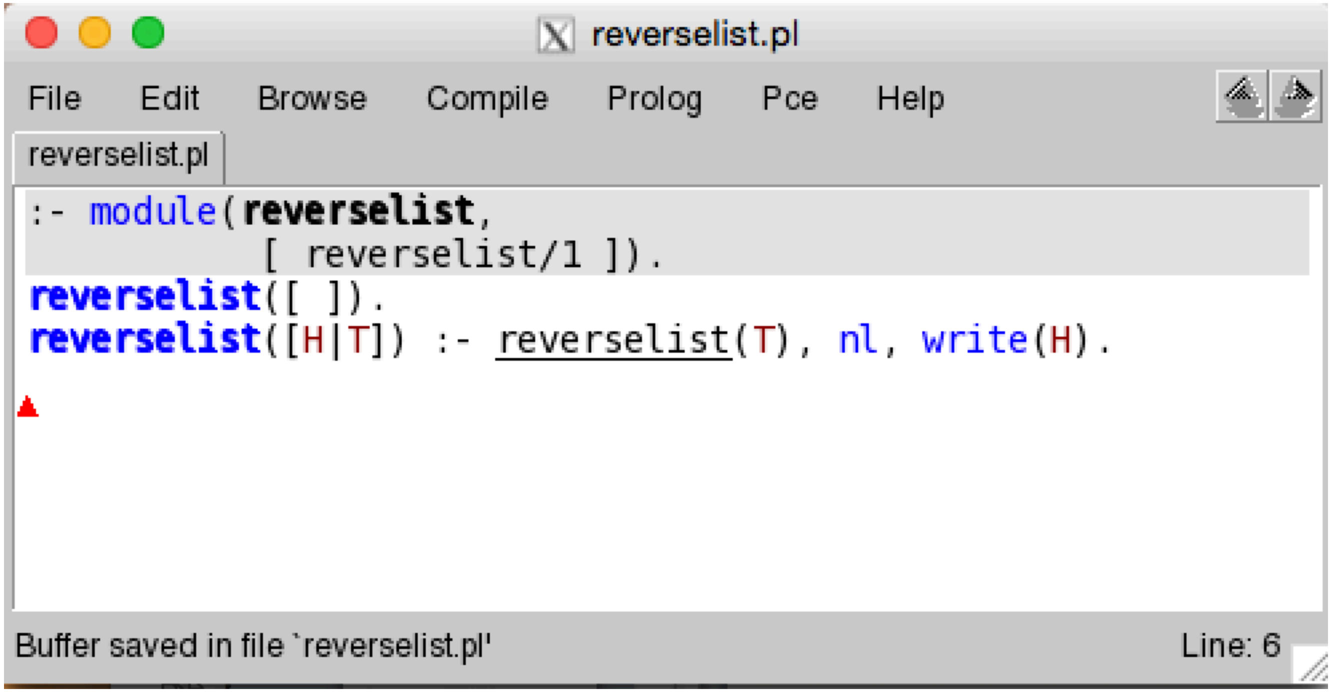
Example: Printing a List

```prolog
:- module( writelist,
           [ writelist/1 ]).
writelist([ ]).
writelist([H|T]) :- write(H), nl, writelist(T).
```

- Uses built-in output predicates:
  - write() – writes out the argument
  - nl – newline

writelist.pl

Example: Reverse Printing a List



- Uses built-in output predicates:
  - write() – writes out the argument
  - nl – newline

reverselist.pl

Today

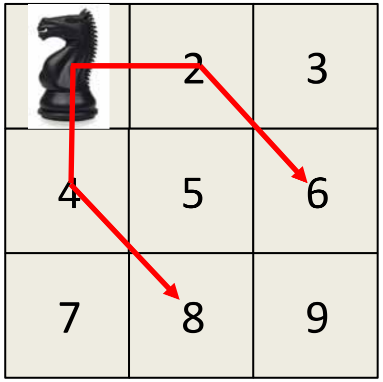
- Lists
- Recursion
- Abstract Data Types

Recursive Search in Prolog

- Consider the simple 3 x 3 Knight's tour problem

- Goal: Find and show a path of Knight moves from one square to another, covering all squares but touching no square twice.

1	2	3
4	5	6
7	8	9



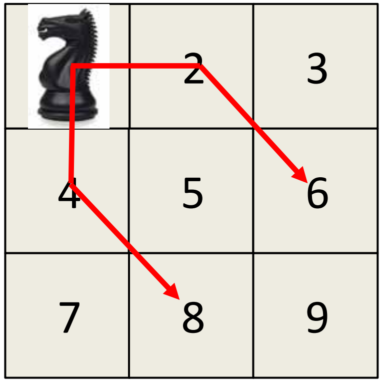
Recursive Search in Prolog

- Consider the simple 3 x 3 Knight's tour problem

- Goal: Find and show a path of Knight moves from one square to another, covering all squares but touching no square twice.

1	2	3
4	5	6
7	8	9

Q: What does our program need to address?



Recursive Search in Prolog

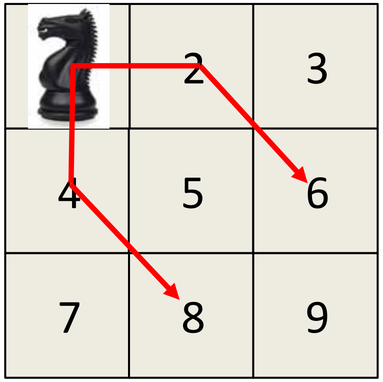
- Consider the simple 3 x 3 Knight's tour problem

1	2	3
4	5	6
7	8	9

- Goal: Find and show a path of Knight moves from one square to another, covering all squares but touching no square twice.

Q: What does our program need to address?

1. Valid moves
2. Building a path
3. Already visited
4. Showing the path



Recursive Search in Prolog

tour3.pl

File Edit Browse Compile Prolog Pce Help

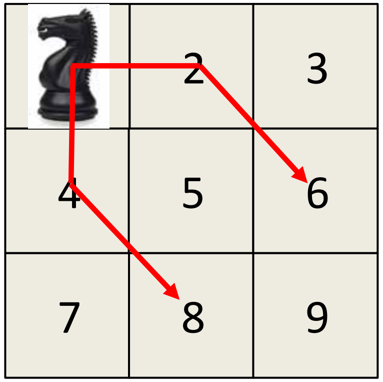
tour3.pl | adts.pl | utils.pl

```prolog
:- module( tour3,
          [
              move/2,
              path/3,
              member/2,
              reverselist/1
          ]).
move(1,6).
move(1,8).
move(2,7).
move(2,9).
move(3,4).
move(3,8).
move(4,3).
move(4,9).
move(6,7).
move(6,1).
move(7,6).
move(7,2).
move(8,3).
move(8,1).
move(9,4).
move(9,2).

path(Z,Z,L) :- reverselist(L).
path(X,Y,L) :- move(X,Z), not(member(Z,L)), path(Z,Y,[Z|L]).

member(A, [A|_]).
member(A, [_|B]) :-
        member(A, B).

reverselist([]).
reverselist([A|B]) :-
        reverselist(B),
        nl,
        write(A).
```

Line: 36

Note that we use the "not" predicate

1	2	3
4	5	6
7	8	9



Knights Tour Program

```
?- tour3:listing.

reverselist([]).
reverselist([B|A]) :-
        reverselist(A),
        nl,
        write(B).

path(A, A, B) :-
        reverselist(B).
path(A, D, C) :-
        move(A, B),
        not(member(B, C)),
        path(B, D, [B|C]).

move(1, 6).
move(1, 8).
move(2, 7).
move(2, 9).
move(3, 4).
move(3, 8).
move(4, 3).
move(4, 9).
move(6, 7).
move(6, 1).
move(7, 6).
move(7, 2).
move(8, 3).
move(8, 1).
move(9, 4).
move(9, 2).

member(A, [A|_]).
member(A, [_|B]) :-
        member(A, B).
```

- Already covered
  - member/2
  - reverselist/1
- Valid moves are listed using the move/2 predicate
- Look at path/3:
  - arg1 = current square
  - arg2 = goal square
  - arg3 = list of squares in path
- path(A, A, B) is executed when tour completed (i.e., the terminating condition for the recursion)
- path(A, D, C) recursively builds a path that does not touch the same square twice
- run query: path(A, A, []). but can be more specific

1	2	3
4	5	6
7	8	9

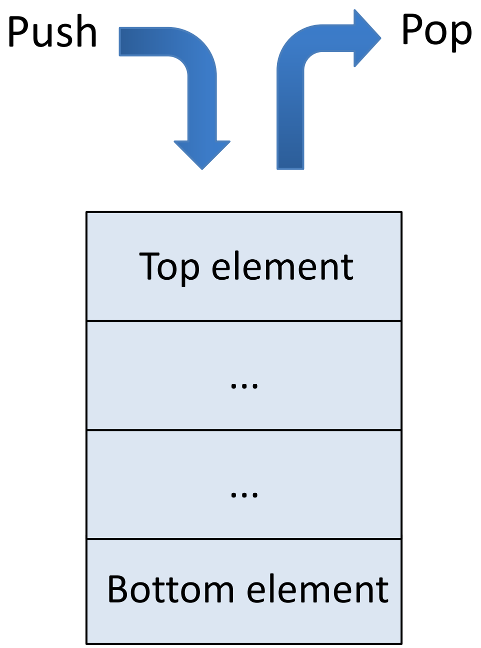
tour3.pl :- path(1,9,[]).

Today

- Lists
- Recursion
- Abstract Data Types

Stack

- A **Last-in, First-out (LIFO)** data structure
- Elements are pushed onto the stack and popped off the stack
- Operatons:
  1. test whether stack is empty
  2. push an element onto the stack
  3. pop the element at the top of the stack
  4. peek at the top element without popping it
  5. test whether an element is in the stack
  6. utility for printing stack in reverse order



Stack Implementation

stack operations
1. test whether stack is empty
2. push an element onto the stack
3. pop the element at the top of the stack
4. peek at the top element without popping it
5. test whether an element is in the stack
6. utility for printing stack in reverse order

- Stack predicates:

1. `empty_stack( [ ] ).` ← can use to test or generate a new empty stack

2 - 4. `stack( Top, Stack, [ Top | Stack ] ).` ← defines stack; use for push, pop, peek

5. `member_stack( Element, Stack ) :- member( Element, Stack ).`

6.
```
reverse_print_stack(S) :- empty_stack(S).
reverse_print_stack(S) :-
        stack(E, Rest, S),
        reverse_print_stack(Rest),
        write(E), nl.
```

Pushing onto a stack

- Basic idea: Bind all variables except the one you wish to find or generate
- To push: stack [E, L1, L2]. where E and L1 are bound, L2 is result

```
[trace]  ?- stack(tom,dick,L).
   Call: (6) stack(tom, dick, _G879) ? creep
   Exit: (6) stack(tom, dick, [tom|dick]) ? creep
L = [tom|dick].
```

```
[trace]  ?- stack(tom,[dick,harry],L).
   Call: (6) stack(tom, [dick, harry], _G885) ? creep
   Exit: (6) stack(tom, [dick, harry], [tom, dick, harry]) ? creep
L = [tom, dick, harry].
```

edit(file('utils.pl')).
[utils].
trace.

Popping and Peeking from a stack

- Basic idea: Bind all variables except the one you wish to find or generate

- To pop: stack[E, L1, L2]. where E and L1 are ***not*** bound, but L2 ***is*** bound

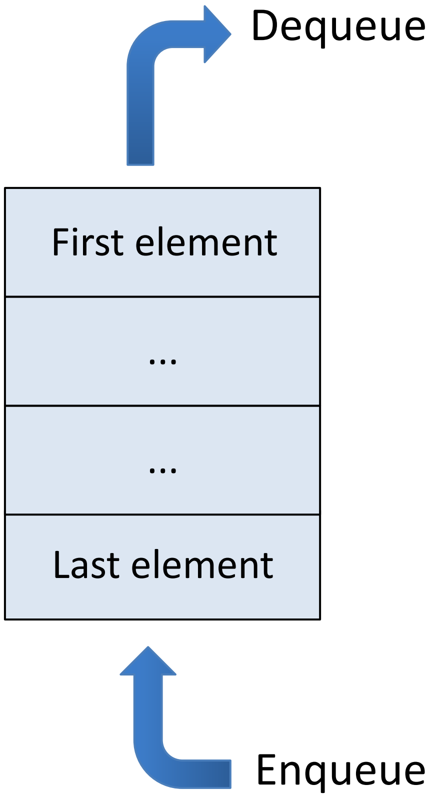
```
[trace]  ?- stack(E,S,[tom,dick,harry]).
   Call: (6) stack(_G886, _G887, [tom, dick, harry]) ? creep
   Exit: (6) stack(tom, [dick, harry], [tom, dick, harry]) ? creep
E = tom,
S = [dick, harry].
```

Here,

- E is the element that has been popped
- L1 is the stack after the top element has been popped
- L2 is the original stack, so if we keep it, we have peeked and found the top element

Queue

- A **First-in, First-out (FIFO)** data structure
- Elements are enqueued (added) to the end of the queue and dequeued (removed) from the front of the queue
- Operations:
    1. test whether queue is empty
    2. add an element to the queue
    3. remove the first element in the queue
    4. peek at the first element without removing it
    5. test whether an element is in the queue
    6. utility for printing queue in reverse order



Queue Implementation

queue operations

1. test whether queue is empty
2. add an element to the queue
3. remove the first element in the queue
4. peek at the first element without removing it
5. test whether an element is in the queue
6. utility for printing queue in reverse order

- Queue predicates:

1. empty_queue([ ]). ← can use to test generate a new empty queue

2 . enqueue(E, [ ], [E]).
   enqueue(E, [H | T], [H | W]) :- enqueue(E, T, W).

3. dequeue(E, [E | T], T).

4. dequeue(E, [E | T], _).

5. member_queue(Element, Queue) :- member(Element, Queue).

Enqueuing, Dequeuing, and Peeking

- Basic idea: Bind all variables except the one you wish to find or generate
- Enqueuing: enqueue (A, B, X). where A and B are bound, X is result

```
?- enqueue(tom,[],X).
X = [tom] .

?- enqueue(tom,[dick,harry],X).
X = [dick, harry, tom] .
```

- Dequeuing: dequeue(E, F, T). where F is bound, E and T not bound.

```
?- dequeue( E, [tom,dick,harry],T ).
E = tom,
T = [dick, harry] .
```

- Peeking: dequeue(E, F, _). where F is bound, E not bound.

```
?- dequeue( E, [tom,dick,harry],_ ).
E = tom .
```

utils.pl

Set

- A data structure for an **unordered collection**
  - duplicate elements are not allowed

- Elements are inserted (added) into the set if they are not already included;
- an element can be deleted (removed) if it is present in the set

- Operations:

1. test whether a set is empty
2. test for membership in a set
3. add element to a set
4. remove element
5. find the union of two sets
6. find the intersection of two sets
7. find the set difference of two sets
8. determine if one set is a subset of another
9. test whether two sets are equal

Set Implementation

set operations

1. test whether a set is empty
2. test for membership in a set
3. add element to a set
4. remove element
5. find the union of two sets
6. find the intersection of two sets
7. find the set difference of two sets
8. determine if one set is a subset of another
9. test whether two sets are equal

- Set predicates:

1. empty_set([ ]).

2. member_set(E,S) :- member(E,S).

3. add_to_set(X, S, S) :- member(X, S), !.
   add_to_set(X, S, [X|S]).

4. remove_from_set(_, [], []).
   remove_from_set(E, [E|T], T) :- !.
   remove_from_set(E, [H|T], [H|T_new]) :- remove_from_set(E, T, T_new), !.

Set Implementation (cont'd)

Set predicates (cont'd):

5. union([], S, S).
   union([H|T], S, S_new) :- union(T, S, S2), add_to_set(H, S2, S_new).
6. intersection([], _, []).
   intersection([H|T], S, [H|S_new]) :- member_set(H, S), intersection(T, S, S_new),!.
   intersection([_|T], S, S_new) :- intersection(T, S, S_new),!.
7. set_diff([], _, []).
   set_diff([H|T], S, T_new) :- member_set(H, S), set_diff(T, S, T_new),!.
   set_diff([H|T], S, [H|T_new]) :- set_diff(T, S, T_new), !.
8. subset([], _).
   subset([H|T], S) :- member_set(H, S), subset(T, S).
9. equal_set(S1, S2) :- subset(S1, S2), subset(S2, S1).

Set Predicate Examples

- Basic idea: Bind all variables except the one you wish to find or generate

- add_to_set(E, A, B) where E = element, A = set to add to, B = result set

```
[?- add_to_set( 3, [ 4,3,5 ], X ).
X = [4, 3, 5].

[?- add_to_set( 3, [ 4,5 ], X ).
X = [3, 4, 5].
```

- remove_from_set(E, A, B)) where E = element, A = set to remove from, B = result set

```
[?- remove_from_set( 3, [ 4,5 ], X ).
X = [4, 5].

[?- remove_from_set( 3, [ 4,5,3 ], X ).
X = [4, 5].
```

More Set Predicate Examples

- intersection(A, B, C) find the set of all elements that are in both sets A and B

```
?- intersection([b,a,d,e],[e,f,a,g,d],X).
X = [a, d, e].

?- intersection([b,a,d,e],[e,f,a,g,b,d],X).
X = [b, a, d, e].
```

- set_diff(A, B , C) find the set of all elements of set A that are not in set B

```
?- set_diff([h,b,a,d,e],[e,f,a,g,d],X).
X = [h, b].

?- set_diff([b,a,d,e],[e,f,a,g,b,d],X).
X = [].
```

Still More Set Predicate Examples

- subset(A, B) determine whether set A is a subset of set B

```
?- subset([3,4],[6,2,3,5,4]).
true .

?- subset([3,4],[6,2,5,4]).
false.
```

- equal_set(A, B) determine whether sets A and B contain the same elements

```
?- equal_set([3,4,5],[4,3,5]).
true .
```

- union(A, B, U) where U is the union of sets A and B

```
?- union([3,4],[4,5,6],X).
X = [3, 4, 5, 6].
```