

---

## Overview

---

⇒ Motion World

- Recursion

---

## Explained Motion Example

---

- If we see motion in a room within a certain amount of time of previous motion in that room, it's fine  
*explained(Room, Now) ← hasdetector(Room)*  
     $\wedge$  *lastmotion(Room, Prev)*  
     $\wedge$  *subtract(Now, Prev, Diff)*  
     $\wedge$  *motionlessinroom(Room, Time)*  
     $\wedge$  *less(Diff, Time)*  
  
- *subtract(X, YZ)* is true for all  $X, Y, Z$  where  $Z = X - Y$

---

## Numbers

---

- We could define subtract
- First, pick a simple representation for numbers
  - Represent  $X$   $Y$   $Z$  as integers with the successor function
  - 0 is a number
  - if  $X$  is a number, so is  $s(X)$
- Define subtract by using successor function
  - $subtract(X, 0, X)$ .
  - $subtract(X, s(Y), Z) \leftarrow$   
 $subtract(X, Y, s(Z))$
- Could just have easily used some other notation
  - binary:  $b(1, b(0, b(1, b(1, null))))$
  - Would need a corresponding definition of subtract

---

## Cheating on Numbers

---

- Silly to define arithmetic
- Why not just build it in
  - $is(X, Y)$ 
    - Evaluate expression  $Y$  using Tcl **expr** command and unify it with  $X$
    - But will require that  $Y$  is a ground expression
      - + Reasoning procedure needs to substitute values in before evaluating it using the **expr** command, as the variables are not Tcl variables
    - Need to be careful about when atom is evaluated
  - This is sort of cheating
    - But very handy!

---

## Other Motion Rules

---

- If motion was previous seen in an adjacent room  

$$\text{explained}(\text{Room}, \text{Now}) \leftarrow \text{connected}(\text{Room}, \text{From})$$

$$\quad \wedge \text{hasdetector}(\text{From})$$

$$\quad \wedge \text{lastmotion}(\text{From}, \text{Prev})$$

$$\quad \wedge \text{is}(1, \{10 \leq (\text{Now} - \text{Prev})\})$$
  - (Leave whitespace around Unification variables so algorithm to substitute their values in doesn't have to know Tcl's syntax)
- If there is room with no detector, could have walked through it  

$$\text{explained}(\text{Room}, \text{Now}) \leftarrow \text{connected}(\text{Room}, \text{From})$$

$$\quad \wedge \text{nodetector}(\text{From})$$
- Pretty weak rule, can we do better?

---

## Recursive Rule

---

- If the motion is from someone in the house, that person would have set off a motion detector previously
- So, must be a path from the room with motion now, to some previous room with motion, going through rooms that do not have motion detectors  

$$\text{explained}(\text{Room}, \text{Now}) \leftarrow \text{connected}(\text{Room}, \text{From})$$

$$\quad \wedge \text{nodetector}(\text{From})$$

$$\quad \wedge \text{stayinroom}(\text{From}, \text{Time})$$

$$\quad \wedge \text{is}(\text{New}, \{ \text{Now} - \text{Time} \})$$

$$\quad \wedge \text{explained}(\text{From}, \text{New})$$
- Recursive rule
  - Base cases given earlier

---

## Problems with Connected

---

```
connected(foyer;library)
connected(library;parlor)
connected(foyer;parlor)
connected(foyer;diningroom)
...
connected(X,Y) ← connected(Y,X)
```

- With top-down resolution, could keep applying rule

```
connectedSub(foyer;library)
connectedSub(library;parlor)
connectedSub(foyer;parlor)
connectedSub(foyer;diningroom)
...
connected(X,Y) ← connectedSub(Y,X)
connected(X,Y) ← connectedSub(X,Y)
```

---

## Why is Example Neat?

---

- Represented knowledge about the domain with just rules and a bunch of facts
- Very concise representation: three rules enough to reason about motion
- Reasoning procedure is separated from knowledge about domain
- We can use an *ordinary* theorem prover

---

## Overview

---

- Motion World
- ⇒ Recursion

---

## Recursion and Mathematical Induction

---

- Idea: define a predicate in terms of simpler instances of itself
  - $live(X) \leftarrow$
  - $connected\_to(X, Y) \wedge$
  - $live(Y).$
  - $live(a).$
  - $connected\_to(a, b).$
  - $connected\_to(b, c).$
- Recursion works by having
  - a well-founded ordering of instances of relations
  - such that each element is defined in terms of elements lower in the ordering
  - and each decreasing chain eventually reaches an element that is simplest in the ordering—defined by a clause with no body

## Recursion in Motion World

---

- **Base Cases**

*explained(Room, Now)* ← *hasdetector(Room)*

∧ *lastmotion(Room, Prev)*

∧ *motionlessinroom(Room, Time)*

∧ *is(1, {Time < (Now - Prev)})*

*explained(Room, Now)* ← *connected(Room, From)*

∧ *hasdetector(From)*

∧ *lastmotion(From, Prev)*

∧ *is(1, {10 < (Now - Prev)})*

- **Recursion**

*explained(Room, Now)* ← *connected(Room, From)*

∧ *nodetector(From)*

∧ *stayinroom(From, Time)*

∧ *is(New, {Now - Time})*

∧ *explained(From, New)*

## Recursive Data Structure

---

- Recursion is only 'looping construct' in Datalog
- Useful to have recursive data structure
  - Process top element of data structure on each level of the recursion and pass rest of the structure during recursion
  - Build up a data structure on way out of recursion on way out of recursion, add top element to structure

## Example: Path through a Maze

- path of 1 2 6 10 could be represented
  - $p(1,p(2,p(6,p(10,nil))))$
  - or  $p(10,p(6,p(2,p(1,nil))))$
  - $p$ : arbitrary function name to glue path together
  - $nil$ : arbitrary symbol to denote empty path
- To get first part of path and remaining, unify with  $p(Top, Rest)$
- Recursive rule to check if a cell is used in the path
  - $member(S;p(S,X))$
  - $member(S;p(X,R)) \leftarrow member(S,R)$

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

## Toward Building Recursive Structures

- Example: path through a maze
  - connectedSub(1,2).
  - connectedSub(2,3).
  - connectedSub(2,6).
  - connectedSub(4,8).
  - ...
  - connected(X,Y) :- connectedSub(X,Y).
  - connected(X,Y) :- connectedSub(Y,X).
- Finding out if there is a path
  - path(X,Y) :- connected(X,Y).
  - path(X,Y) :- connected(X,Z), path(Z,Y).
  - ?- path(1,16).

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

## Proof Derivation

---

```
?path(1,16)
yes ← path(1,16)
   use: path( $X_1, Y_1$ ) ← connected( $X_1, Z_1$ ) ∧ path( $Z_1, Y_1$ )
   sub:  $X_1/1 Y_1/16$ 
yes ← connected(1, $Z_1$ ) ∧ path( $Z_1, 16$ )
   use: connected( $X_2, Y_2$ ) ← connectedSub( $X_2, Y_2$ )
   sub:  $X_2/1 Y_2/Z_1$ 
yes ← connectedSub(1, $Z_1$ ) ∧ path( $Z_1, 16$ )
   use: connectedSub(1,2)
   sub:  $Z_1/2$ 
yes ← path(2,16)
...
```

## Building Recursive Structures

---

- How can we remember the path?
- Building path on way out of recursion
 

```
path( $X, X, p(X, nil)$ ).
path( $X, Y, p(X, PathZioY)$ ) ←
  connected( $X, Z$ )
  path( $Z, Y, PathZioY$ )
?- path(1,16,Path) Path = p(1,p(2,p(6,p(10,p(11,p(7,p(8,p(12,p(16,nil))))))))
```
- Does code really work?
- Might end up in an endless cycle
  - Need to make sure we do not visit a cell already in the path
  - Will come back to this issue

## Proof Derivation

---

```

?path(1,3,Path)
yes(Path) ← path(1,3,Path)
  use: path(X1,Y1,p(X1,PathZY1)) ← connected(X1,Z1) ∧ path(Z1,Y1,PathZY1)
  sub: X1/1 Y1/3 Path/p(1,PathZY1)
yes(p(1,PathZY1)) ← connected(1,Z1) ∧ path(Z1,3,PathZY1)
  use: connected(X2,Y2) ← connectedSub(X2,Y2)
  sub: X2/1 Y2/Z1
yes(p(1,PathZY1)) ← connectedSub(1,Z1) ∧ path(Z1,3,PathZY1)
  use: connectedSub(1,2)
  sub: Z1/2
yes(p(1,PathZY1)) ← path(2,3,PathZY1)
  use: path(X3,Y3,p(X3,PathZY3)) ← connected(X3,Z3) ∧ path(Z3,Y3,PathZY3)
  sub: X3/2 Y3/3 PathZY1/p(2,PathZY3)
yes(p(1,p(2,PathZY3))) ← connected(2,Z3) ∧ path(Z3,3,PathZY3)
...

```

## Why is this building the path on way **out**?

- Path is being accumulated in the answer atom, but not available to the path clause
 
$$\text{path}(X, Y, p(X, \text{PathZtoY})) \leftarrow \text{path}(X, Y, \text{Path}) \leftarrow \text{connected}(X, Z)$$

$$\text{path}(Z, Y, \text{PathZtoY})$$

$$\text{Path} = p(X, \text{PathZtoY})$$
- ‘=’ is for explicit unification
  - Really just an infix version of the 2-ary predicate ‘unify’, which can be defined as ‘unify(X,X)’
  - Not adding any power to Datalog

## Proof Derivation (2nd Version)

---

```

?path(1,3,Path)
yes(Path) ← path(1,3,Path)
use: path(X1,Y1,P1) ← connected(X1,Z1) ∧ path(Z1,Y1,PathZY1) ∧ P1=path(X1,PathZY1)
sub: X1/1 Y1/3 P1/Path
yes(Path) ← connected(1,Z1) ∧ path(Z1,3,PathZY1) ∧ Path=path(1,PathZY1)
use: connected(X2,Y2) ← connectedSub(X2,Y2)
sub: X2/1 Y2/Z1
yes(Path) ← connectedSub(1,Z1) ∧ path(Z1,3,PathZY1) ∧ Path=path(1,PathZY1)
use: connectedSub(1,2)
sub: Z1/2
yes(Path) ← path(2,3,PathZY1) ∧ Path=path(1,PathZY1)
use: path(X3,Y3,P3) ← connected(X3,Z3) ∧ path(Z3,Y3,PathZY3) ∧ P3=path(X3,PathZY3)
sub: X3/2 Y3/3 P3/PathZY1
yes(Path) ← connected(2,Z3) ∧ path(Z3,3,PathZY3) ∧ PathZY1=path(X3,PathZY3)
  ∧ Path=path(1,PathZY1)
...

```

## Building Path on Way In

---

- Build the current path before the recursive call
  - On way out
 

<pre> path(X, Y, Path) ←   connected(X, Z) path(Z, Y, PathZtoY) Path = p(X, PathZtoY)           </pre>	<pre> - On way in path(X, Y, PathToX) ←   connected(X, Z) PathToZ = p(Z, PathToX) path(Z, Y, PathToZ)           </pre>
--------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------
  - For 'way in' version, how is answer returned to user?
  - Do both versions build the same path?

---

## More Recursion

---

- $\text{length}(\text{List}, \text{Len})$  true if  $\text{List}$  has length  $\text{Len}$

```
length(nil, 0).
length(p(E, Rest), Len) ←
  length(Rest, RestLen)
  is(Len, RestLen + 1).
```

---

## Appending lists

---

- Might want a predicate  $\text{append}(L1, L2, L3)$ 
  - This is similar to Tcl's concat command, not its lappend
  - True if  $L3$  is list of elements of  $L1$  followed by elements of  $L2$
  - Useful for verifying whether  $L3$  is  $L1$  appended in front of  $L2$
  - But also should be useful for appending two lists together:
    - +  $\text{append}(p(\text{tim}, \text{nil}), p(\text{john}, p(\text{phil}, p(\text{ted}, \text{nil}))), L3)$
    - + Or finding the prefix of a list of given ending
      - +  $\text{append}(L1, p(c, p(d, \text{nil})), p(a, p(b, p(c, p(d, \text{nil}))))))$
- How do we define  $\text{append}(L1, L2, L3)$ ?
  - Only tricky part is which of  $L1$  and  $L2$  to attack
  - This is determined by how lists are represented

---

## Reversing a list

---

- $rev(L, R)$  true if  $R$  is the reversal of list  $L$

*# Good place to start is with list represented as head and tail*

$rev(p(Head, Tail), R) \leftarrow$

*# now to reverse the tail*

$rev(Tail, TR)$

*# now to stick head after the reversal of the tail*

$append(TR, p(Head, nil), R).$

$rev(nil, nil).$

- Not very efficient though, as we append after each step

- Number of steps in proof  $O(n^2)$

- Can we do better?

---

## Efficiently Reversing a List

---

- Let's take advantage of what we learned from building paths on way *in* versus *out* of the recursion
- To reverse a list, we can
  - pull off elements of the list on way into recursion
  - and put them onto another list on way into recursion
- Like washing a stack of dishes
  - Take top one off, wash it, and put it on the top of the clean dishes

---

## Definition

---

- Define  $\text{rev3}(J,K,L)$

- where  $J$  is remainder of list to be reversed
- $K$  is the reversal of the list that has been reversed so far
- $L$  will be used to return the list at the bottom of the recursion

- Definition:

$\text{reverse}(J,L) \leftarrow \text{rev3}(J,\text{nil},L).$

*# now define rev3 recursively in terms of a smaller version of itself*

$\text{rev3}(p(\text{Head},\text{Tail}),K,L) \leftarrow$

$\text{rev3}(\text{Tail},p(\text{Head},K),L)$

*# now the base case*

$\text{rev3}(\text{nil},L,L).$

---

## Building Lists

---

- Append:

$\text{append}(p(\text{Head},\text{Tail}),L,p(\text{Head},R)) \leftarrow$

$\text{append}(\text{Tail},L,R)$

$\text{append}(\text{nil},L,L).$

- on way into recursion, tear off top element
- on way out, put at top of list

- rev3:

$\text{rev3}(p(\text{Head},\text{Tail}),L2,L3) \leftarrow$

$\text{rev3}(\text{Tail},p(\text{Head},L2),L3)$

$\text{rev3}(\text{nil},L2,L2).$

- on way into recursion, tear off top element and put on top of new list
- pass back completed list on way out