Inference and Prolog

cs540 section 2
Louis Oliphant
oliphant@cs.wisc.edu
(some slides borrowed from Burr Settles)

Announcements

- HW 1
  - grading is done
  - If you have questions on grading please see the TA
  - Solution is on-line
- HW2 due today
- HW3
  - assigned today. Due next Thursday, 27th October
  - It has been shortened. No programming portion :-(
- Today's lecture is last used for the midterm

Reasoning in Subsets of FOL

- Subsets of FOL
  - First Order Definite Clauses
  - Datalog
- Reasoning in these Subsets
  - Forward Chaining
  - Backward Chaining
- Prolog

Subsets of FOL

- Definite Clauses
  - Disjunction of Literals with *exactly one positive*, where variables are always universally quantified.
  - Ex.

    $\neg$King(x) $\vee$ $\neg$Greedy(x) $\vee$ Evil(x).
    King(Father(John)).
    Greedy(y).
  - Usually written as a rule:

    King(x) $\wedge$ Greedy(x) $\Rightarrow$ Evil(x).
- Datalog
  - first-order definite clauses with no function symbols

Definite Clause Knowledge Base

"The law says that it is a crime for an American to sell weapons to hostile nations. The country, Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American."

American(x)∧Weapon(y)∧sells(x,y,z)∧Hostile(z) ⇒ Criminal(x)
Owns(Nono,M_1)
Missile(M_1)
Missile(x)∧Owns(Nono,x) ⇒ Sells(West,x,Nono)
Missile(x) ⇒ Weapon(x)
Enemy(x,America) ⇒ Hostile(x)
American(West)
Enemy(Nono,America)

Generalized Modus Ponens

- Modus Ponens

$$\frac{\alpha \Rightarrow \beta,\ \alpha}{\beta}$$

- Generalized Modus Ponens

$$\frac{p_1', p_2', \ldots, p_n',\ (p_1 \wedge p_2 \wedge \ldots \wedge p_n \Rightarrow q)}{SUBST(\theta, q)}$$

(where $SUBST(\theta, p_i') = SUBST(\theta, p_i)$ for all i)

- Example

$$\frac{\text{King(John)},\ \text{Greedy(y)},\quad \text{King(x)} \wedge \text{Greedy(x)} \Rightarrow \text{Evil(x)}}{\text{Subst}(\{\text{x/John, y/John}\}, \text{Evil(John)})}$$

Forward Chaining Algorithm

function FOL-FC-ASK(KB, α) **returns** a substitution or *false*
  repeat until *new* is empty
    new ← { }
    for each sentence r **in** KB **do**
      $(p_1 \wedge \ldots \wedge p_n \Rightarrow q) \leftarrow$ STANDARDIZE-APART(r)
      for each θ such that $(p_1 \wedge \ldots \wedge p_n)\theta = (p_1' \wedge \ldots \wedge p_n')\theta$
          for some $p_1', \ldots, p_n'$ in KB
        $q' \leftarrow$ SUBST(θ, q)
        if q' is not a renaming of a sentence already in KB or *new* **then do**
          add q' to *new*
          $\phi \leftarrow$ UNIFY(q', α)
          if ϕ is not *fail* **then return** ϕ
    add *new* to KB
  return *false*

Forward Chaining Example

American(x)∧Weapon(y)∧sells(x,y,z)∧Hostile(z) ⇒ Criminal(x)
Missile(x)∧Owns(Nono,x) ⇒ Sells(West,x,Nono)
Missile(x) ⇒ Weapon(x)
Enemy(x,America) ⇒ Hostile(x)

Criminal(West)

American(West) | Missile(M_1) | Owns(Nono,M_1) | Enemy(Nono,America)

Forward Chaining Example

American(x)∧Weapon(y)∧sells(x,y,z)∧Hostile(z) ⇒ Criminal(x)
Missile(x)∧Owns(Nono,x) ⇒ Sells(West,x,Nono)
Missile(x) ⇒ Weapon(x)
Enemy(x,America) ⇒ Hostile(x)

Criminal(West)

American(West) Missile(M_1) Owns(Nono,M_1) Enemy(Nono,America)

Forward Chaining Example

American(x)∧Weapon(y)∧sells(x,y,z)∧Hostile(z) ⇒ Criminal(x)
Missile(x)∧Owns(Nono,x) ⇒ Sells(West,x,Nono)
Missile(x) ⇒ Weapon(x)
Enemy(x,America) ⇒ Hostile(x)

Criminal(West)

American(West) Missile(M_1) Owns(Nono,M_1) Enemy(Nono,America)

Forward Chaining Example

American(x)∧Weapon(y)∧sells(x,y,z)∧Hostile(z) ⇒ Criminal(x)
Missile(x)∧Owns(Nono,x) ⇒ Sells(West,x,Nono)
Missile(x) ⇒ Weapon(x)
Enemy(x,America) ⇒ Hostile(x)

Criminal(West)

Sells(West,M_1,Nono)

American(West) Missile(M_1) Owns(Nono,M_1) Enemy(Nono,America)

Forward Chaining Example

American(x)∧Weapon(y)∧sells(x,y,z)∧Hostile(z) ⇒ Criminal(x)
Missile(x)∧Owns(Nono,x) ⇒ Sells(West,x,Nono)
Missile(x) ⇒ Weapon(x)
Enemy(x,America) ⇒ Hostile(x)

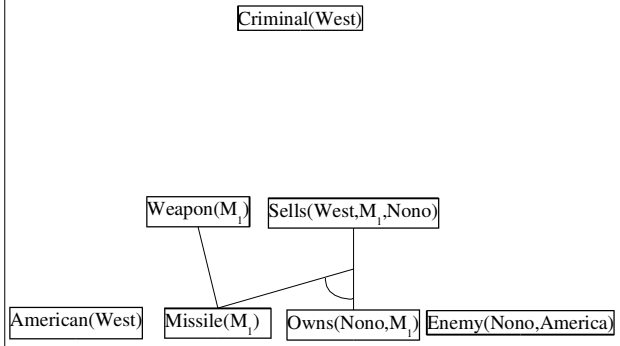
Criminal(West)

Sells(West,M_1,Nono)

American(West) Missile(M_1) Owns(Nono,M_1) Enemy(Nono,America)

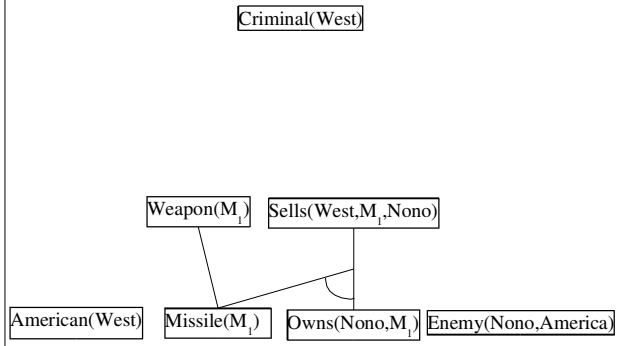
Forward Chaining Example

American(x)∧Weapon(y)∧sells(x,y,z)∧Hostile(z) ⇒ Criminal(x)
Missile(x)∧Owns(Nono,x) ⇒ Sells(West,x,Nono)
Missile(x) ⇒ Weapon(x)
Enemy(x,America) ⇒ Hostile(x)



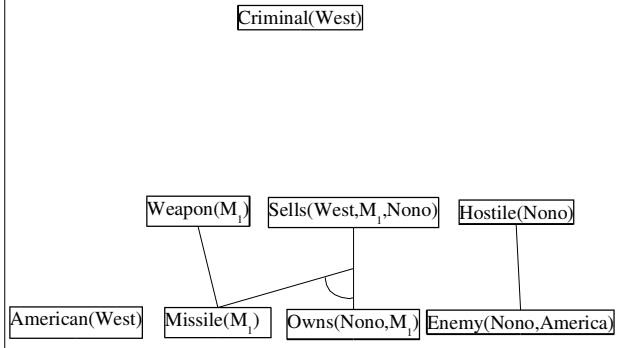
Forward Chaining Example

American(x)∧Weapon(y)∧sells(x,y,z)∧Hostile(z) ⇒ Criminal(x)
Missile(x)∧Owns(Nono,x) ⇒ Sells(West,x,Nono)
Missile(x) ⇒ Weapon(x)
Enemy(x,America) ⇒ Hostile(x)



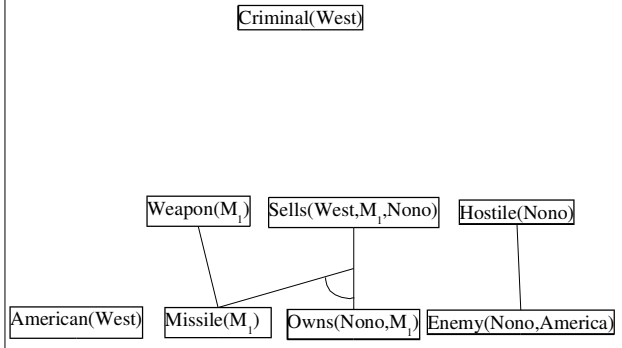
Forward Chaining Example

American(x)∧Weapon(y)∧sells(x,y,z)∧Hostile(z) ⇒ Criminal(x)
Missile(x)∧Owns(Nono,x) ⇒ Sells(West,x,Nono)
Missile(x) ⇒ Weapon(x)
Enemy(x,America) ⇒ Hostile(x)



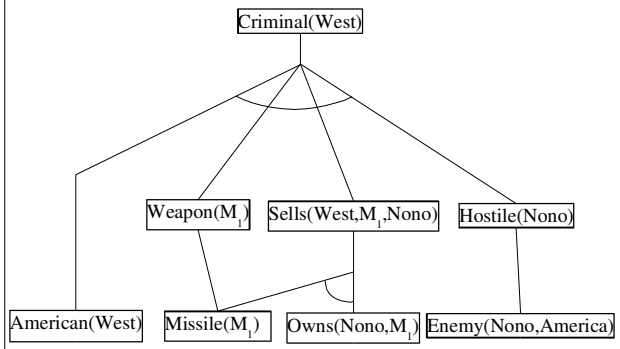
Forward Chaining Example

American(x)∧Weapon(y)∧sells(x,y,z)∧Hostile(z) ⇒ Criminal(x)
Missile(x)∧Owns(Nono,x) ⇒ Sells(West,x,Nono)
Missile(x) ⇒ Weapon(x)
Enemy(x,America) ⇒ Hostile(x)



Forward Chaining Example

American(x)∧Weapon(y)∧sells(x,y,z)∧Hostile(z) ⇒ Criminal(x)
Missile(x)∧Owns(Nono,x) ⇒ Sells(West,x,Nono)
Missile(x) ⇒ Weapon(x)
Enemy(x,America) ⇒ Hostile(x)



Analysis of Forward Chaining

- "inner loop" involves finding all possible unifiers
  - Pattern Matching problem
  - NP-hard
  - Good heuristics exist
- Recheck each rule on each iteration
  - Instead, ***Every new fact inferred on iteration t must be derived from at least one new fact from iteration t-1***.
- Generates many facts that are irrelevant to goal
  - Use Backward chaining instead
  - Use ***Magic sets***
    - From Database community
    - Change rules to take into consideration the goal

Backward Chaining Algorithm

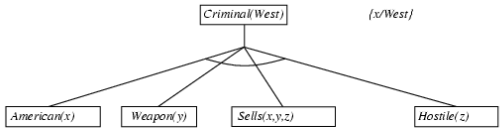
function FOL-BC-ASK(KB, goals, θ) **returns** a set of substitutions
  inputs: KB, a knowledge base
    goals, a list of conjuncts forming a query
    θ, the current substitution, initially the empty substitution { }
  local variables: *ans*, a set of substitutions, initially empty

  if *goals* is empty **then return** $\{\theta\}$
  $q' \leftarrow$ SUBST(θ, FIRST(*goals*))
  for each r **in** KB where STANDARDIZE-APART(r) = $(p_1 \wedge \ldots \wedge p_n \Rightarrow q)$
    and $\theta' \leftarrow$ UNIFY(q, q') succeeds
    ans $\leftarrow$ FOL-BC-ASK($KB, [p_1, \ldots, p_n | \text{REST}(goals)], \text{COMPOSE}(\theta, \theta')) \cup ans$
  return *ans*

Backward chaining example

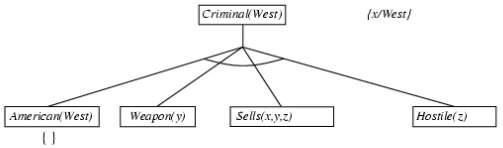
American(x)∧Weapon(y)∧sells(x,y,z)∧Hostile(z) ⇒ Criminal(x)
Owns(Nono,M_1)
Missile(M_1)
Missile(x)∧Owns(Nono,x) ⇒ Sells(West,x,Nono)
Missile(x) ⇒ Weapon(x)
Enemy(x,America) ⇒ Hostile(x)
American(West)
Enemy(Nono,America)

Backward chaining example



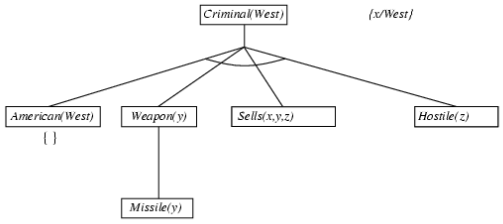
American(x)∧Weapon(y)∧sells(x,y,z)∧Hostile(z) ⇒ Criminal(x)
Owns(Nono,M_1)
Missile(M_1)
Missile(x)∧Owns(Nono,x) ⇒ Sells(West,x,Nono)
Missile(x) ⇒ Weapon(x)
Enemy(x,America) ⇒ Hostile(x)
American(West)
Enemy(Nono,America)

Backward chaining example



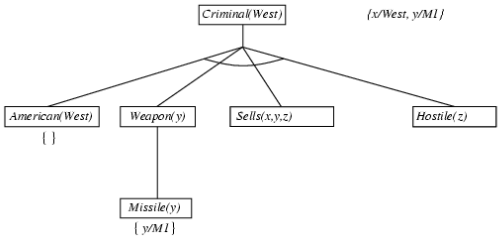
American(x)∧Weapon(y)∧sells(x,y,z)∧Hostile(z) ⇒ Criminal(x)
Owns(Nono,M_1)
Missile(M_1)
Missile(x)∧Owns(Nono,x) ⇒ Sells(West,x,Nono)
Missile(x) ⇒ Weapon(x)
Enemy(x,America) ⇒ Hostile(x)
American(West)
Enemy(Nono,America)

Backward chaining example



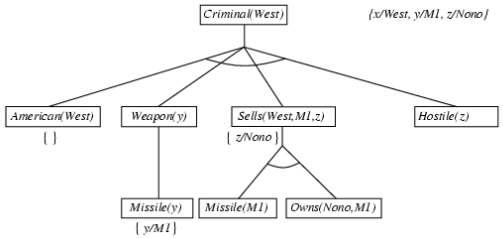
American(x)∧Weapon(y)∧sells(x,y,z)∧Hostile(z) ⇒ Criminal(x)
Owns(Nono,M_1)
Missile(M_1)
Missile(x)∧Owns(Nono,x) ⇒ Sells(West,x,Nono)
Missile(x) ⇒ Weapon(x)
Enemy(x,America) ⇒ Hostile(x)
American(West)
Enemy(Nono,America)

Backward chaining example



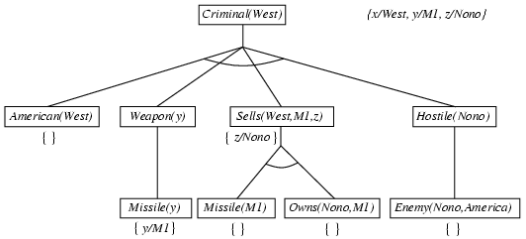
American(x)∧Weapon(y)∧sells(x,y,z)∧Hostile(z) ⇒ Criminal(x)
Owns(Nono,M_1)
Missile(M_1)
Missile(x)∧Owns(Nono,x) ⇒ Sells(West,x,Nono)
Missile(x) ⇒ Weapon(x)
Enemy(x,America) ⇒ Hostile(x)
American(West)
Enemy(Nono,America)

Backward chaining example



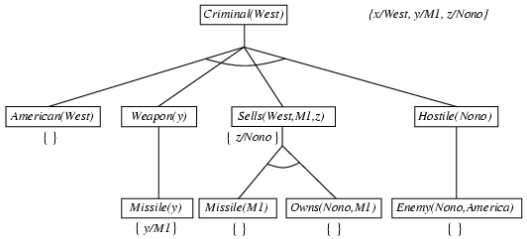
American(x)∧Weapon(y)∧sells(x,y,z)∧Hostile(z) ⇒ Criminal(x)
Owns(Nono,M_1)
Missile(M_1)
Missile(x)∧Owns(Nono,x) ⇒ Sells(West,x,Nono)
Missile(x) ⇒ Weapon(x)
Enemy(x,America) ⇒ Hostile(x)
American(West)
Enemy(Nono,America)

Backward chaining example



American(x)∧Weapon(y)∧sells(x,y,z)∧Hostile(z) ⇒ Criminal(x)
Owns(Nono,M_1)
Missile(M_1)
Missile(x)∧Owns(Nono,x) ⇒ Sells(West,x,Nono)
Missile(x) ⇒ Weapon(x)
Enemy(x,America) ⇒ Hostile(x)
American(West)
Enemy(Nono,America)

Backward chaining example



American(x)∧Weapon(y)∧sells(x,y,z)∧Hostile(z) ⇒ Criminal(x)
Owns(Nono,M_1)
Missile(M_1)
Missile(x)∧Owns(Nono,x) ⇒ Sells(West,x,Nono)
Missile(x) ⇒ Weapon(x)
Enemy(x,America) ⇒ Hostile(x)
American(West)
Enemy(Nono,America)

Prolog

- Programming = Logic + Control
- Prolog is an implementation of First Order Definite Clauses (plus some control)
- Queries are answered by Back Chaining

Characteristics of Prolog

- Descriptive Languages vs. Procedural Languages
  - Nouns vs. Verbs
  - Definitions vs. Actions
- Definite FOL clauses
  - All "procedures" are if-then rules
  - Additional non-logical pieces added (cuts, negation by failure)
- Prolog is an interpreted language (not compiled)
- We will be using the yap interpreter

Syntax of Prolog

- All clauses have the following form:

```
head(args) :- body1(args) , body2(args) , ... , bodyn(args).
```

:- means "if"
, means "and"
. ends a clause

- It is a rule in reverse
- Think of the clause as a "procedure" in procedural languages (like java or c)

Syntax of Prolog

- Variables Begin With Upper-Case Letters
  - All variables are universally quantified (∀)
- procedures, constants, and functions begin with lower case letters.

some example clauses:

```
male(john).
female(mary).
childOf(mary,john).

parentOf(X,Y):-childOf(Y,X).
fatherOf(X,Y):-male(X),childOf(Y,X).
```

A Simple Program

hello_world.pl

```
hello_world:-write('hello world'), nl.
```

To execute the program run:

```
%yap
?- [hello_world.pl].
yes
?- hello_world.
hello world
yes
?- halt.
%
```

Prolog Example

Example prolog DB that encodes our "criminal" example (note that variables are capitalized, and constants in lower-case):

```
missile(m).
owns(nono,m).
enemy(nono,america).
american(west).
weapon(X) :- missile(X).
sells(west,X,nono) :- missile(X),owns(nono,X).
hostile(X) :- enemy(X,america).
criminal(X) :-
      american(X), weapon(Y),
      sells(X,Y,Z), hostile(Z).
```

Prolog Example

- The implementation of prolog that we'll use on the TUX machines is called YAP
  - "Yet Another Prolog"
  - Freeware implementation, downloadable from **www.ncc.up.pt/~vsc/Yap/**
- To run prolog on a TUX machine, type: `% yap`
- To end prolog, type: `?- halt.`
- To load a file, type: `?- [file].`
- The prolog extension is `*.pl`
  - Try not to confuse it with perl programs
  - Note that you don't need the extension when loading a program into prolog, it knows to look for the file with a `*.pl` extension

Prolog Example

- Once YAP is running and the criminal KB is loaded, we can start by asking simple queries we clearly already know:

```
?- missile(m).        ?- american(west).
```

- Then we can move on to more complex queries:

```
?- weapon(X).         ?- owns(X,m).
?- criminal(west).
```

- To view the entire BC search, YAP has a debugging feature called "spy":
  - Type `spy(predicate).` to turn it on
  - And `nospy(predicate).` to turn it off

Another Prolog Example

- Let's consider a simple KB that expresses facts about a certain family:

```
father(tom,dick).     mother(tom,judy).
father(dick,harry).   mother(dick,mary).
father(jane,harry).   mother(jane,mary).
```

- Now let's also think about creating some FOL rules for defining family relations:
  - Parent?

```
parent(X,P) :- mother(X,P).
parent(X,P) :- father(X,P).
```

  - Grandmother?

```
granny(X,G) :- parent(X,Y), mother(Y,G).
```

Another Prolog Example

- How should we define the relation sibling?
  - Two people are siblings if they have the same mother and the same father (ignoring half-siblings, step-siblings, etc.)
- How about this:

```
sibling(X,Y) :- mother(X,M), mother(Y,M),
        father(X,F), father(Y,F).
```

- Let's run this and see what happens!
  - Oops! Need to make sure $X \neq Y$!

```
sibling2(X,Y) :- mother(X,M), mother(Y,M),
        father(X,F), father(Y,F), X\=Y.
```

More Prolog Syntax

- Prolog has built-in operators (predicates) for mathematical functions and equalities:
  - $x = 2\times(y+1)$ `X is 2*(Y+1).` *(y must already have a value)*
  - $d < 20$ `D < 20.`
  - $1 \leq 2$ `1 @=< 2.`
  - $x = y$ `X = Y.`
  - $x \neq y$ `X \= Y.`
- The major data structure for Prolog is the *list*
  - `[]` denotes an empty list
  - `[H|T]` denotes a list with a head (H) and tail (T)
    - The head is the first element of the list
    - The tail is the entire sublist after it
    - *e.g.* for the list [a,b,c,d]… H=[a] and T=[b,c,d]

List Processing in Prolog

- Suppose we want to define an "append" operator for lists… that is to take two lists *L1* and *L2*, and merges their elements together into a new list *L3*
  - In procedural languages this is done with a function
    - *e.g.* `L3 = append(L1,L2)`
  - Create make-shift functions by defining predicates with the return value included as a parameter
    - *e.g.* `append(L1,L2,L3)`
- How about defining a simple predicate that takes the first two L1 and L2, and returns a new list [L1|L2]?
  - *e.g.* `append(L1,L2,[L1|L2]).`
  - Nope! Let's try again…

List Processing in Prolog

- What we need to do is take one list and recursively add one element at a time from the other list, until we've added them all
- Let's assume that we start with *L2* and want to add the elements from *L1* one at a time to the front
  - Makes things easier: with [H|T], H is the front element
  - What is our base case?
    - `append([],L2,L2).`
  - Now how do we deal with the *recursive* aspect?
    - `append([H|T],L2,[H|L3]) :- append(T,L2,L3).`

List Processing in Prolog

- Now we can ask the queries:

```
?- append([1,2,3], [a,b,c], [1,2,3,a,b,c]).
```

  - Result: `yes`

```
?- append([1,2,3], [a,b,c], X).
```

  - Result: `X = [1,2,3,a,b,c]`

```
?- append(A, B, [1,2]).
```

  - Result:

```
A=[]    B=[1,2]
A=[1]   B=[2]
A=[1,2] B=[]
```

- Recall that, since prolog uses BC, we can try to find any *single* solution, or find *all* solutions
  - After each result, type "`;`" to view another

Partitioning Lists

- Another useful application might be how to recursively sort prolog lists
- Most sorting algorithms utilize some partitioning method, where the list *L* is split into two sublists *L1* and *L2* based on a particular element *E*
  - *e.g.* splitting list [1,5,3,9,7,4,1] on element [5] would yield the lists [1,3,4,1] and [5,9,7]
- This would be a useful method to define first

```
partition(E, L, L1, L2).
```

Partitioning Lists

- First, let's think of the base case for our partitioning predicate

```
partition(E,[],[],[]).
```

  - That is, an empty list gets split into two empty lists
- Second, we must consider the recursive aspect:
  - Upon considering a new element *H* at the head of the list, what conditions must we account for?
    - If $H < E$, or if $H \geq E$ (to determine which sublist)
  - Since we have two different cases, each with a different desired result, we need two recursive definitions

Partitioning Lists

- If $H < E$, then we want to add *H* to the first list *L1*:

```
partition(E,[H|T],[H|T1],L2) :-
        H < E,
        partition(E,T,T1,L2).
```

- However, if $H \geq E$ then we'll add it to the second list *L2*:

```
partition(E,[H|T],L1,[H|T2]) :-
        H @>= E,
        partition(E,T,L1,T2).
```

- These predicates, together with the base case, will partition all the list items less than *E* in the first list, and all greater or equal in the second list

Sorting in Prolog

- Now that we know how to partition one list into two, and also how to append two lists together, we have all the tools we need to sort a list!
- Let's consider insertion sort:
  - Walk through each position of the list
  - For each position, *insert* the list item *i* that belongs in that position, relative to other items in the list
  - Recursively, we can achieve the same effect by walking through each *i*, partitioning a pre-sorted list on *i*, and then appending the partitions on either side

Sorting in Prolog

- As always, we will need a base case for insertion sort (assume that an empty list is sorted):

```
isort([],[]).
```

- For the recursive aspect, we can walk through the whole list, and backtrack, inserting each element where it belongs in the pre-sorted list:

```
isort([H|T], F) :-
        isort(T,L),
        partition(H,L,L1,L2),
        append(L1,[H|L2],F).
```

- We can do something similar to implement quicksort, but I'll leave that up to you to work out on your own!

Ordering Prolog Rules

- The rules in a prolog program are searched depth-first, exploring the potential rules from top down
- So The order of your rules is very important:
  - Place base cases first
  - Place recursive cases last

Example – Segmentation Problem

- Remember the Segmentation problem from homework 1?
  - Given a string:
    "inwhichweseehowanagentcanfindasequenceofactionsthatachievesitsgoalswhennosingleactionwilldo"
  - Output string with spaces where each "word" is in the dictionary:
    "in which we see how an agent can find a sequence of actions that achieves its goals when no single action will do"
- Let's write this in prolog!

Segmentation Problem

- Designing The Program:
  add_spaces(InputList,OutputList).
    where InputList is a list of characters with no spaces
    and OutputList is a list of characters with spaces such that each set of characters between spaces is in the dictionary
- Methodology:
  Take the smallest "word" from the left of InputList
  Recursively call add_spaces on the remainder of the InputList
  Put the smallest "word" back together with the result of the recursive call, with a space in-between.
- So, first we will need some helper predicates

Helper Predicates

split_list_at_element_n(N,List,FirstN,Remainder)
  N – the number of elements from List that should be in FirstN
  List – the list to be split
  FirstN – the first N elements of List
  Remainder – the rest of List after the first N elements

```
split_list_at_element_n(0,Rest,[],Rest).
split_list_at_element_n(Num,[Head|Tail],[Head|Tail2],Rest):-
    Num2 is Num -1,
    split_list_at_element_n(Num2, Tail, Tail2,Rest).
```

Helper Predicates

merge(List1,List2,CombinedList)
  List1 – a list of characters
  List2 – another list of characters
  CombinedList – The concatination of List1and List2
  (just another implementation of append)

```
merge([],List,List).
merge([Head|Tail],List,[Head|List2]):-
    merge(Tail,List,List2).
```

Helper Predicate

run_add_spaces(Num,AsciiList,AsciiList2)
  Num – The next "word" to try from AsciiList
  AsciiList – The list of characters without spaces
  AsciiList2 – The list of characters with spaces added (this predicate will do almost all of the work)

```
run_add_spaces(_,[],[]).
run_add_spaces(Num,AsciiList,AsciiList2):-
    split_list_at_element_n(Num,AsciiList,FirstN,Rest),
    is_word(FirstN),
    run_add_spaces(1,Rest,RestWithSpaces),
    merge(FirstN,[32|RestWithSpaces],AsciiList2).  %32 is a space in ascii
run_add_spaces(Num,AsciiList,AsciiList2):-
    Num2 is Num + 1,
    length(AsciiList,Length),
    Num2 @=< Length,
    run_add_spaces(Num2,AsciiList,AsciiList2).
```

Final Predicate

```
add_spaces(AsciiList,AsciiList2):-
    run_add_spaces(1,AsciiList,AsciiList2),
    name(S2,AsciiList2),
    name(S,AsciiList),
    format("'~p'~nwith spaces added is~n'~p'~n",[S,S2]).
```

Conclusion

- First Order Definite Clauses, Datalog
- Generalized Modus Ponens
- Forward Chaining and Backward Chaining
- Yap is an implementation of Prolog
  - handles first order definite clauses
  - plus a few “non-logical” extensions
    - cuts, negation by failure
- Syntax symbols include :- , . [] |
- Order of clauses in file is important
  - place base cases first followed by recursive cases
- The web-site for yap prolog is:
  http://www.ncc.up.pt/~vsc/Yap/
- You can download yap for linux or windows
- It has an on-line users manual