Prolog

Burr H. Settles
CS-540, UW-Madison
www.cs.wisc.edu/~cs540-1
Summer 2003

Announcements

- Read Sections 10.2-10.3, and Chapter 11 in *AI: A Modern Approach* for Monday
- Homework #2 is due on Monday
  - Handin directories are already set up, in case you're ready to turn it in
- Homework #3 will go out on Monday (but it won't be due until after the midterm)

Logic Programming

- Computation as inference on logical KBs

	Ordinary Programming	Logic Programming
1.	Identify problem	Identify problem
2.	Assemble information	Assemble information
3.	Plan out your algorithms	Go have a beer!
4.	Program the solution	Encode information in the KB
5.	Encode problem as data	Encode problem as logical facts
6.	Run program on data	Ask queries
7.	Debug errors	Find/correct false facts

Should be easier to debug `capital(NewYork,USA)` than find that pesky `x += 2`!

Logic Programming

- Logic programs are also called expert systems
  - Problem domain experts sit down and encode lots of information into the KB
  - System then reasons using that "expert" knowledge
  - Used for medical diagnosis, Q/A systems, etc.
  - KB must be in datalog format: FOL with no functions
- Fall loosely under the "think rationally" quadrant of AI research

Prolog

- Prolog is probably the most common logic programming language
- A program is:
  - A set of logic sentences in HNF (definite clauses)
    - Called the database (DB, basically the KB)
    - Ordered by programmer (top to bottom)
  - Executed by specifying a query to be proved
    - Backward-chaining with GMP
    - Uses DFS on the ordered facts and rules
    - Searches until a solution is found (or times out)
  - Can find multiple solutions

Prolog Execution

To Solve a Goal (*i.e.* answer a query)

- Try to unify:
  - First with each of the ground facts
  - When all facts fail, then with each of the consequents of the rules in the order in which they occur in DB
- Successful unification with a fact:
  - Solved, pop goal from stack
- Successful unification with a rule:
  - Solve the sub-goals in DFS manner (*i.e.* recursively attempt to solve each of the rule's premises)

Prolog Execution

Backtracking During DFS

- While solving a rule:
  - If antecedent (premise) fails to be proved true
  - Then try to re-prove it using *different* facts or rules
- When a rule fails:
  - If an antecedent can't be solved at all, the rule fails
  - Go on to the next rule in the program and try again (try to unify current goal with a different consequent)

7

Prolog Execution

- Efficient implementation
  - Unification use "open coding"
  - Retrieval and matching of clauses by direct linking
  - Sophisticated memory management
- Uses a closed world assumption
  - Negation as failure
  - *e.g.* given `¬dead(X) ⇒ alive(X)`
    `alive(elvis)` succeeds if `dead(elvis)` fails
- Widely used in Europe and Japan

8

Basic Prolog Syntax

- Database:
  - **Fact:** a positive literal (atom)
    FOL: `F(x)` *Prolog:* `f(X).`
    FOL: `¬F(x)` *Prolog:* `not f(X).`
    variables are capitalized and universally quantified
  - **Rules:** 1 positive literal (the consequent, or head), and at least 1 negative (the antecedents)
    FOL: $A_1 \wedge A_2 \wedge \ldots \wedge A_n \Rightarrow C$ *Prolog:* `C:- A1, A2, … , An.`
- Query:
  - *FOL:* $Q_1 \wedge Q_2 \wedge \ldots \wedge Q_n$ *Prolog:* `?- Q1, Q2, … , Qn.`

9

Prolog Example

Example prolog DB that encodes our "criminal" example from last time (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).
```

10

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

11

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

12

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).
```

13

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,M), father(Y,M).
```

- 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,M), father(Y,M), X\=Y.
```

14

More Prolog Syntax

- Prolog has built-in operators (predicates) for mathematical functions and equalities:
  - $x = 2\times(y+1)$ `X is 2*(Y+1).`
  - $d < 20$ `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]

15

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*
  - Usually this is done with a function
    - *e.g.* `L3 = append(L1,L2)`
  - But prolog programs are *datalog*: no functions allowed!
    - 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.* `app(L1,L2,[L1|L2]).`
  - Nope! Let's try again…

16

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).`

17

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

18

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).
```

19

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

20

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

21

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

22

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!

23

Parsing with Prolog

- A lot of early natural language processing (NLP) research was historically done using logic systems, because HNF rules are analogous to grammar productions
  - e.g. A simple English grammar: $S \rightarrow NP\ VP$ (*NP*)
    - S means "sentence," NP means "noun phrase," and VP means "verb phrase"
  - In prolog:

```
s(Input) :- np(Input, Mid), vp(Mid, []).
s(Input) :- np(Input, Mid1), vp(Mid1, Mid2), np(Mid2, []).
```

- Once we add definitions for **`np`** and **`vp`** (and ultimately **`noun`**, **`verb`**, **`prep`**, **`det`**, etc.), we will have a full-blown deterministic English parser!

24

Ordering Prolog Rules

- The rules in a prolog program are searched depth-first, exploring the potential rules from top down
- Imagine that we are designing a knowledge-based reflex agent that has multiple rules which which it can unify
  - We want to make more specific rules toward the top, and more general rules toward the bottom
  - For recursive "functions," that means making sure the base case comes before the recursive case(s)

Other Logic Systems

- Production Systems
  - Proposed by E.L. Post in 1943
  - Equivalent in computational power to a Turing machine.
    - Rules are unordered, unlike Prolog
  - Have been developed for a wide variety of problems, ranging from algebra word problems, mathematical and logical proofs, physics problems, and games.
  - Newell and Simon (1960s) used production systems to define model human cognition
    - Production rules represent problem-solving skills stored in a person's long-term memory.
  - Many other groups have tried to develop similar models of human cognition using production systems
  - Harder to do inference than BC in Prolog

Other Logic Systems

- Semantic Networks
  - Used widely in computational linguistics
  - Developed to aid in machine translation and natural language understanding ("WordNet" is a famous SN)
  - Represent knowledge in a hierarchy of semantic classes to be able to deduce and disambiguate meaning
    - *e.g.* "Jane looked for her keys. She needed milk."
    - The query: Why was Jane looking for her keys?

Summary

- Logic programs are a agent programs that use facts and rules in a KB to answer questions about a particular domain
- Such programs arrive at conclusions (or decide on actions) in a logical way
- Prolog is one of the most common logic programming languages
  - Uses first-order definite clauses to encode the KB
  - Searches for proofs recursively with BC and GMP
  - Can answer yes/no queries, or find bindings