

*Aside: as mentioned earlier, this topic is really out of order -- so we can spread the programming assignments out better*

## Logic Programming & Prolog

Prolog (Programming in Logic) is the prime example of a logic programming language

- only one with real wide spread use

These languages are declarative

- They let the programmer indicate what is to be computed, but leave (some/many/most of) the details of how it is to be computed to the system
- Not based on state modification (e.g., not imperative)
- Also not procedural in nature
  - Procedure or function is not the primary programming construct
  - Does not have control flow (as we are used to thinking of it)

=> Quite different

Why do we want to do this?

Declarative semantics are simpler

Will make simple statements in logic

Can understand their meaning without knowing the "state" of the program at the point of execution (e.g., the value of all the variables)

- which implies understanding a lot about the execution history of the program up to that point

If system can pull it off, its just plain a lot easier to say what, but not how

=> downside of logic programming, may be quite inefficient

Example:

describing what constitutes sorted is easy (2 lines, p. 542 in the book) does not allow system to infer an efficient algorithm to do it

Often have to compromise declarative semantics some for efficiency

But if it is well suited to the problem get a big win

So if we don't have procedures or control flow, what do we have?

Programming in a logic programming language (we will stick to Prolog) is based on statements in predicate calculus (logic)

As we will see, you can also look at this several different ways (e.g., searching in a relational database)

Review of logic and the predicate calculus

Proposition

- logical statement that may or may not be true
- made up of "objects" and their relationships to each other
- combined with logical operators (and, or, etc.)

Will program in clauses that represent propositions

## Objects

- in logic programming made up of simple terms (constants or variables)
- in Prolog these will end up being: variables (ids starting in upper case), numeric and string literals, and symbolic literals (ids starting in lower case)

## Atomic Propositions

represents a relation

two parts:

functor            - names the relation

parameters       - objects which take part in the relation

examples

facultyMember(scott)

teaches(scott, cs3411)

Note: functors do not supply meaning  
just ids, we interpret them  
logic would work the same with "foo" and "bar"

Functors simply name a relation

Atomic propositions either indicates members of  
that relation (facts)

faculty(scott)  
faculty(spencer)  
faculty(raja)

faculty
scott
spencer
raja

teaches(scott, cs3411)  
teaches(spencer, cs3411)  
teaches(raja, cs3411)  
teaches(scott, cs6395)

teaches	
scott	cs3411
spencer	cs3411
raja	cs3411
scott	cs6395

Or a question about membership in a relation  
(query)

teaches(X, cs3411)

## Compound Propositions

two or more atomic propositions combined with logical operators

### Notation(s)

Name	Book	Ex	Alt	Meaning
negation	$\neg$	$\neg a$	! ~	not a
conjunction	$\cap$	$a \cap b$	$\wedge$ & ,	a and b
disjunction	$\cup$	$a \cup b$	$\vee$	a or b
equivalence	$\equiv$	$a \equiv b$	=	a equiv to b
implication	$\supset$	$a \supset b$	$\Rightarrow$	a implies b
		$a \leftarrow b$	$\Leftarrow$ :-	b implies a

Precedence:  $\neg$  then  $\cap \cup \equiv$  then  $\supset$

Examples:

$$a \cap b \supset c$$

$$a \wedge b \Rightarrow c$$

$$a \cup (b \cap c) \supset d$$

$$a \vee (b \wedge c) \Rightarrow d$$

## Quantifiers

Universal  $\forall X.P$  For all  $X$ ,  $P$  is true

Existential  $\exists X.P$  There exists a value of  $X$   
such that  $P$  is true

(often leave out the ".")

Examples:

$\forall X.(\text{teachingFaculty}(X) \Rightarrow \text{facultyMember}(X))$

$\forall X.(\text{teachingFaculty}(X) \Rightarrow \exists Y.\text{teaches}(X,Y))$

(Won't see much in the way of quantifiers, but they are implicit in a number of places)

## Predicate Calculus and Making Inferences

Would like to infer new propositions (e.g., facts)  
from some existing set of propositions  
Automatic theorem proving

An inference rule that can be applied automatically  
is "Resolution"

If we have:

$$P1 \leftarrow P2 \quad \text{and} \quad Q1 \leftarrow Q2$$

and it turns out that P1 is identical to Q2, then we  
can rename P1 and Q2 to T to get:

$$T \leftarrow P2 \quad \text{and} \quad Q1 \leftarrow T$$

from which we can deduce:

$$Q1 \leftarrow P2$$

(this is resolution)

Resolution gets more complex if variables are involved

To use resolution in the presence of variables, we need to find values for variables that allow matching to proceed

Example

$f(X, Y) \leq P2(Y, X) \ \& \ Q1(\text{foo}) \leq f(\text{foo}, \text{bar})$   
if we rename  $X = \text{foo}$  and  $Y = \text{bar}$  we can conclude  
 $Q1(\text{foo}) \leq P2(\text{bar}, \text{foo})$

Process of finding these suitable "assignments" of values to variables is called "unification"

Temporarily "assigning" a value to a variable for unification is called "instantiation" of the variable

Resolution process often requires backtracking  
instantiate a variable with a value  
matching fails  
backtrack by instantiating with another value  
and trying match again

Prolog operates on the basis of unification

Consider propositions with variables

=> queries

Unification becomes a search process

=> find a set of assignments of values to variables that make this proposition true

Prolog uses limited form of logical expression:

Horn Clauses:

$Q_1 \wedge Q_2 \wedge \dots \wedge Q_n$  (fact)

- or -

$P \leftarrow Q_1 \wedge Q_2 \wedge \dots \wedge Q_n$  (implication)

Note: you can't express all propositions this way  
lack of negation is particularly troubling as  
we will see later

Prolog notation for this:

$Q_1, Q_2, \dots, Q_n .$  (fact or query)

$P :- Q_1, Q_2, \dots, Q_n .$  (rule)

Two ways to read a rule:  $P :- Q1, Q2.$   
If Q1 and Q2 then conclude P  
To show P, first show Q1, then Q2

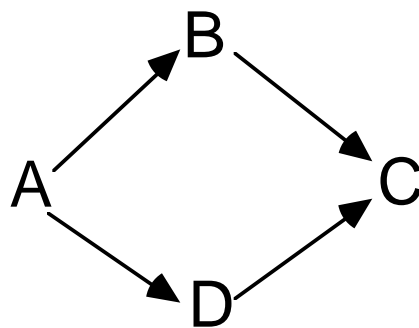
Each proposition is a term

Terms can consist of

constants	integer
	atom (literal id or quoted string)
variable	id starting with upper case
structure	functorid (term, ..., term)

(plus list construct we will see later: [... | ...])

Example (a graph):



$\text{link}(a,b), \text{link}(b,c), \text{link}(a,d), \text{link}(d,c).$   
 $\text{path}(N, N).$   
 $\text{path}(L, M) :- \text{link}(L, X), \text{path}(X, M).$

Based on our logical encoding of the graph, we can then write queries:

?- path(a,c)  
yes

?- path(c,a)  
no

?- path(a,X), path(X,c)  
X = a  
X = b  
X = c  
X = d

Notice we don't write a graph traversal algorithm  
- and don't hard code the set of questions we  
can ask in advance

We just define what a graph is