# Topic #16: Logic Programming

CSE 413, Autumn 2004

Programming Languages

http://www.cs.washington.edu/education/courses/413/04au/



## References

- Slides from CSE 341 – S. Tanimoto
- See Chapter 16 of the text
  - Read 16.1, 16.4, 16.5, 16.6 (skip 16.6.7)
  - Skim 16.7, 16.8



## Motivation

**1. Reduce the programming burden.**

**2. System should simply accept the necessary information and the objective (goal), and then figure out its own solution.**

**3. Have a program that looks more like its own specification.**

**4. Take advantage of logical inference to automatically get many of the consequences of the given information.**



## What is a program?

•Whatever it is, it must be __________
e.g., interpretable by some identified "computer language processor."



Q: Is a logical description of a problem actually a program?



## Prolog Program Structure

**A Prolog program consists of an ordered collection of logical statements. These usually represent:**

**• background information**

**•specific information for a given problem**

**•a hypothesis or a statement containing a free variable.**

## Sample Problem

**For someone (call him or her X) to be the *grandmother* of someone else (call him or her Y), X must be the mother of someone (call him or her Z) who is a parent of Y.**

**Someone is a *parent* of another person, if that someone is either the mother or the father of the other person.**

**Mary is the mother of Stan.**
**Gwen is the mother of Alice.**
**Valery is the mother of Gwen.**
**Stan is the father of Alice.**

**The question: Who is a grandmother of Alice?**



## Sample Prolog Program: `grandmother.pl`

```
grandmother(X, Y) :- mother(X, Z), parent(Z, Y).
parent(X, Y) :- mother(X, Y).
parent(X, Y) :- father(X, Y).

mother(mary, stan).
mother(gwen, alice).
mother(valery, gwen).
father(stan, alice).


grandmother(X, alice).
```



## Sample Session

```
Welcome to SWI-Prolog (Version 3.3.2)
Copyright (c) 1990-2000 University of Amsterdam. All
rights reserved.

For help, use ?- help(Topic). or ?- apropos(Word).

?- [grandmother].
% grandmother compiled 0.00 sec, 1,312 bytes

Yes
?- grandmother(X, alice).

X = mary ;

X = valery ;

No
?-
```



## How does this work?

**Propositional Logic**

**Clauses**

**Resolution**

**Predicate Logic**

**Unification**



## Propositional Logic

*"If the butler had a motive and the butler was alone with the victim then the butler is guilty."*

Define our atomic formulas:

P: The butler had a motive.
Q: The butler was alone.
R: The butler is guilty.

Express the compound formula:

(P & Q) -> R    a typeable version of $(P \wedge Q) \rightarrow R$



## Clause Form

**Any boolean formula can be put into conjunctive normal form (CNF). From there, it's easy to get "clause form." Automatic inference using resolution requires statements be in clause form.**

```
If not Y then X and not Z.
Y or (X & not Z)
(Y or X) & (Y or not Z)
(X V Y) & (Y V ~Z)
```

**clauses:** `(X V Y), (Y V ~Z)`
`X, Y,` **and** `~Z` **are called literals.** **(A literal is a variable or a negated variable.)**
**Each clause is a disjunction of literals.**
**A formula in CNF is a conjunction of clauses.**

For our example with the butler, we get the following clause:
~ P V ~ Q V R

# Resolution

Resolution is a way to make a new clause from two existing clauses. If the two original clauses were true, then the new one is, too.

The two existing clauses must be compatible, in order to use resolution. There must exist some literal in one clause that occurs negated in the other clause.

Example: Clause 1: A V ~ B V C
Clause 2: B V D

The resolvent is formed by disjoining (i.e., combining with "V" all the literals of clause 1 and clause 2 except the ones involving B.

This results in:



# Example of Resolution

*"If the butler had a motive and the butler was alone with the victim then the butler is guilty. If the butler needed money, then the butler had a motive. The butler needed money. The butler was alone with the victim."*

Prove that the butler was guilty.

Premises: c1: ~ P V ~ Q V R
c2: ~ S V P *If the butler needed money, then the butler had a motive.*
c3: S *The butler needed money.*
c4: Q *The butler was alone with the victim.*
Assume to the contrary that the butler was not guilty: c5: ~ R

Show a contradiction by resolving to obtain the null clause [].
c1 and c4 resolve to give c6:
c6 and c2 resolve to give c7:
c7 and c5 resolve to give c8:
c8 and c3 resolve to give c6:



# Going from Propositional Logic to Predicate Logic

**Propositional logic is too limited in its expressive power to help us do real-world programming.**

**We need to extend it to a system that uses domain variables, functions, and constants.**

**We will still be able to use resolution to perform inference mechanically.**



# Predicate Logic

**"Every Macintosh computer uses electricity."**

**(all x) (Macintosh(x) implies UsesElectricity(x))**

**variables: x, y, z, etc.**
**constants: a, b, c, etc.**
**function symbols: f, g, etc.**
**Predicate symbols: P, Q,**
**Macintosh, UsesElectricity**
**quantifiers: all, exists**

**Logical connectives: not, implies, and, or.**
**~, ->, &, V.**



# Clause Form

```
(all x) (Macintosh(x) -> UsesElectricity(x)

UsesElectricity(x) V Not Macintosh(x).
```

**Clause form for predicate calculus expressions is similar to that for propositional calculus expressions.**

**To achieve clause form, several steps may be required.**
**a. Rewrite X -> Y expressions as ~ X V Y.**
**b. Reduce scopes of negation.**
**c. Using strict rules, eliminate quantifiers.**
**Universal: drop the quantifier (implicit)**
**Existential: use "skolem constants"**
**d. Convert to conjunctive normal form**



# How to extend resolution to clauses with variables?

**Literals are matched using a method called unification.**

**Unification involves substituting "terms" for variables.**

## Unification of Literals

**A *substitution* is a set of term/variable pairs.**
`{ f(a)/x, b/y, z/w }`

**A *unifier* for a pair of literals is a substitution that when applied to both literals, makes them identical.**

`P(x, a)`, `P(f(a), y)` **have the unifier**
`{ f(a)/x, a/y }`

`P(x)`, `P(y)` **have the unifier** `{ a/x, a/y }`,
**but they also have the unifier** `{ x/y }`.

19

## Unification Example

```
mother(jill).
father(bob).
man(X) :- father(X).
```

Goal:

```
man(X).
```

Unifier?

20

## Issues for Prolog Solvers

- Solving direction
  - Forward chaining:
  - Backward chaining:
- Solving multiple clause goals:
  - Depth-first:
  - Breadth first:
- Backtracking

21

## Backtracking Example

```
male(bob).
male(john).
male(fred).
...
parent(fred, shelley).
```

**<u>Goal:</u>**

```
male(X), parent(X, shelley).
```

22

## 'Rithmetic

```
speed(ford, 100).
speed(chevy, 105).
speed(dodge, 95).
speed(volvo, 80).
time(ford, 20).
time(chevy, 21).
time(dodge, 24).
time(volvo, 24).
distance(X, Y), :- speed(X, Speed),
                   time (X, Time),
                   Y is Speed * Time.
```

**Goal:**

```
distance(chevy, Chevy_Distance).
```

23

## Inside solving: `trace`

```
trace.
distance(chevy, Chevy_Distance).

Call: distance(chevy, _0)?
Call: speed(chevy, _5)?
Exit: speed(chevy, 105)
Call: time(chevy, _6)?
Exit: time(chevy, 21)
Call: _0 is 105*21?
Exit: 2205 is 105*21
Exit: distance(chevy, 2205)

Chevy_Distance = 2205
```

24

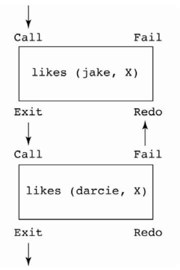
## `trace` with backtracking

```
likes(jake, chocolate).
likes(jake, apricots).
likes(darcie, licorice).
likes(darcie, apricots).

trace.
likes(jake, X), likes(darcie, X).

Call: likes(jake, _0)?
Exit: likes(jake, chocolate)
Call: likes(darcie, chocolate)
Fail: likes(darcie, chocolate)
Redo: likes(jake, _0)?
Exit: likes(jake, apricots)
Call: likes(darcie, apricots)
Exit: likes(darcie, apricots)

X = apricots
```






## Problems with Prolog

1. Resolution Order control
   In theory, programmer shouldn't care.
   In practice, greatly affects efficiency

2. Problems with handing negation

3. Converting spec to execution

```
sort(old, new) :- permute(old,new), sorted(new).

sorted ([]).
sorted ([x]).
sorted([x,y | list]) :- x <= y, sorted([y | list]).
```



## Applications of Logic Programming

1. Databases
   Express facts and queries with one language
   Deduction capability built in
2. AI
   - Expert systems – need facts, heuristics, inferencing engine
     Legal, medical, financial helpers
   - Natural language processing
     Parse sentences described by CFG

   Prolog allows very concise specification of these apps
3. Other uses of "declarativism?"



## An Example: E-Agents

[roll animation here]

Based on McDowell, Etzioni, and Halevy. "The Specification of Agent Behavior by Ordinary People: A Case Study." ISWC 2004, November, 2004.

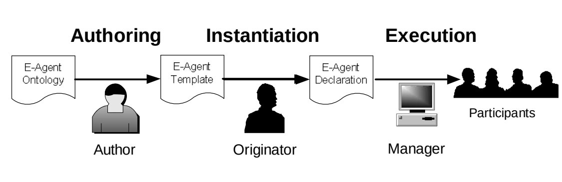


## Overview of E-Agents

- General pattern of an *E-Agent*
  - *Ask* people some set of questions
  - *Collect* their responses
  - *Ensure* that results satisfy some constraints
- Questions have logical interpretation
  - Enables automated responses, re-use of data



## E-Agent Lifecycle



## E-Agent Template Authoring

- Components
  - **Questions** to ask
  - **Constraints** to enforce
  - **Notifications** to send

31

## Template Part 1: Questions

Ask the participants what dish they will bring:

```
[a              :StringQuestion;
 :name          "Bring";
 :enumeration   "$Choices$"; ]
```

(Maybe) Ask how many guests they will bring:

```
[a              :IntegerQuestion;
 :guard         "$AskForNumGuests$";
 :name          "NumGuests";
 :minInclusive  "0"; ]
```

32

## Template Part 2: Constraints

Ensure that responses are pair-wise balanced:

```
[a           :MustConstraint
 :forAll     ( [:name   "x";
                :range "$Choices$" ]

               [:name   "y";
                :range "$Choices$}"] );

 :enforce    "abs($Bring.x.count()$ -
                 $Bring.y.count()$ )
              <= $MaxImbalance$";   ]
```

33

## Template Part 3: Notifications

Notify the originator when the total number of guests hits a threshold value:

```
[a           :OnConditionSatisfied;
 :guard      "$GuestThreshold$ > 0";
 :condition "$NumGuests.sum()$ >=
               $GuestThreshold$";
 :notify     :Originator;
 :message
   "Currently, $NumGuests.sum()$
    guests are expected.";]
```

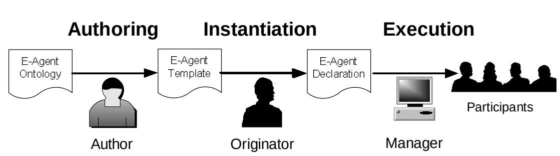
34

## Results: Declarative vs. Procedural Specification

| Name | Proc. size | Decl. Size | Savings |
|---|---|---|---|
| Potluck | 1680 | 170 | 90% |
| FCFS | 536 | 99 | 82% |
| Meeting | 743 | 82 | 89% |
| Approval | 1058 | 109 | 90% |
| Auction | 503 | 98 | 81% |

35

## E-Agent Lifecycle



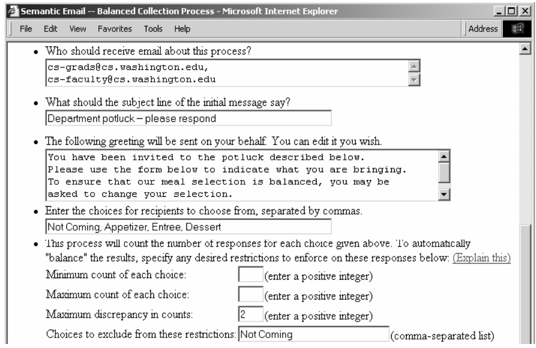

36

## E-Agent Instantiation

- Tool generates HTML form from details in template
  - Parameters needed plus restrictions
- Originator enters parameters, e.g.,
  - **`Participants`** (who to ask?)
  - **`Choices`** (what options to offer?)
  - **`AskForNumGuests`** (should we ask about guests?)
  - **`GuestThreshold`** (when to notify the originator?)



## Web form for Instantiation




## Instantiation Safety

- Possible instantiation:
  - **`AskForNumGuests`** = *False*
  - **`GuestThreshold`** = *50*
  - …
- Problem:
  - **`NumGuests`** question not defined →
    ERROR: notification references undefined symbol
  - Very confusing for originator
- Solutions?
  - Manual checking, or
  - Automated *instantiation safety* testing



## Instantiation Safety - Results

- Template is instantiation-safe if **every possible** instantiation results in a legal declaration (well-formed, symbols defined)
- Result: testing instantiation safety is co-NP-complete in the # of parameters
  - Need to consider all possible combinations
- Result: polynomial time in common cases:
  - Restrict guards, quantifications to use only small number of parameters
  - Ensure that parts of template can be can be checked independently



## Logic Programming Summary

- Programs specify desired outcomes, not how to get there
  - May be inefficient
  - But enables concise specification
  - Very useful for certain domains
- Prolog most widely used
  - Adds support for I/O, lists, arithmetic
  - Enables some control over efficiency