**Prolog I**

**Facts, rules, and queries**

- Fact: Socrates is a man.
  \[ \text{man(socrates).} \]
- Rule: All men are mortal.
  \[ \text{mortal(X)} :\text{- man(X).} \]
- Query: Is Socrates mortal?
  \[ \text{mortal(socrates).} \]

**Running Prolog I**

- Create your "database" (program) in any editor
- Save it as *text only*, with a `.pl` extension
- Here's the complete "program":
  \[ \text{man(socrates).} \]
  \[ \text{mortal(X)} :\text{- man(X).} \]

**Syllogisms**

- “Prolog” is all about **programming in logic**.
  - Socrates is a man.
  - All men are mortal.
  - Therefore, Socrates is mortal.
Running Prolog II

• Prolog is completely interactive.
• Begin by invoking the Prolog interpreter.
  – sicstus
• Then load your program.
  – consult('mortal.pl')
• Then, ask your question at the prompt:
  - mortal(socrates).
• Prolog responds:
  - Yes

On gl.umbc.edu

> sicstus
SICStus 3.7.1 ... Licensed to umbc.edu
| ?- consult('mortal.pl').
{consulting /home/faculty4/finin/cmsc/331/fall00/prolog/mortal.pl...}
{/home/faculty4/finin/cmsc/331/fall00/prolog/mortal.pl consulted, 0 msec
624 bytes}
yes
| ?- mortal(socrates).
yes
| ?- mortal(X).
X = socrates ? yes
| ?-

Syntax I: Structures

• Example structures:
  - sunshine
  - man(socrates)
  - path(garden, south, sundial)
• <structure> ::= <name> | <name> ( <arguments> )
• <arguments> ::= <argument> | <argument> , <arguments>

Syntax II: Base Clauses

• Example base clauses:
  - debug_on.
  - loves(john, mary).
  - loves(mary, bill).
• <base clause> ::= <structure> .
Syntax III: Nonbase Clauses

- Example nonbase clauses:
  - mortal(X) :- man(X).
  - mortal(X) :- woman(X)
  - happy(X) :- healthy(X), wealthy(X), wise(X).

- \(<\text{nonbase clause}>::=\)
  \(<\text{structure}>::=\text{<structures> .}\)
- \(<\text{structures}>::=\)
  \(<\text{structure}> | <\text{structures}> , <\text{structure}>\)

Syntax IV: Predicates

- A predicate is a collection of clauses with the same \textit{functor} and \textit{arity}.
  - loves(john, mary).
  - loves(mary, bill).
  - loves(chuck, X) :- female(X), rich(X).

- \(<\text{predicate}>::=\)
  \(<\text{clause}> | <\text{predicate}> <\text{clause}>\)
- \(<\text{clause}>::=\)
  \(<\text{base clause}> | <\text{nonbase clause}>\)

Syntax V: Programs

- A \text{program} is a collection of predicates.
- Predicates can be in any order.
- Predicates are used in the order in which they occur.

Syntax VI: Assorted details

- Variables begin with a capital letter:
  \(X, \text{ Socrates}, \_\text{result}\)
- Atoms do not begin with a capital letter:
  \(x, \text{ socrates}\)
- Other atoms must be enclosed in single quotes:
  - 'Socrates'
  - 'C:/My Documents/examples.pl'
Syntax VII: Assorted details

- In a quoted atom, a single quote must be quoted or backslashed: 'Can't, or won't?'
- /* Comments are like this */
- Prolog allows some infix operators, such as :- (turnstile) and , (comma). These are syntactic sugar for the functors ':-' and ',',.
- Example: 
  ':-'(mortal(X), man(X)).

Backtracking

- loves(chuck, X) :- female(X), rich(X).
- female(jane).
- female(mary).
- rich(mary).
- --------- Suppose we ask: loves(chuck, X).
- female(X) = female(jane), X = jane.
- rich(jane) fails.
- female(X) = female(mary), X = mary.
- rich(mary) succeeds.

Additional answers

- female(jane).
- female(mary).
- female(susan).
- ?- female(X).
- X = jane ;
- X = mary
- Yes

Readings

- loves(chuck, X) :- female(X), rich(X).
- Declarative reading: Chuck loves X if X is female and rich.
- Approximate procedural reading: To find an X that Chuck loves, first find a female X, then check that X is rich.
- Declarative readings are almost always preferred.
Nonmonotonic logic

- Prolog’s facts and rules can be changed at any time.
  - `assert(man(plato)).`
  - `assert((loves(chuck,X) :- female(X), rich(X))).`
  - `retract(man(plato)).`
  - `retract((loves(chuck,X) :- female(X), rich(X))).`

Common problems

- Capitalization is extremely important!
- No space between a functor and its argument list: `man(socrates), not man (socrates).`
- Don’t forget the period! (But you can put it on the next line.)

A Simple Prolog Model

- Imagine prolog as a system which has a database composed of two components:
  - FACTS - statements about true relations which hold between particular objects in the world. For example:
    - `parent(adam,able): adam is a parent of able`
    - `parent(eve,able): eve is a parent of able`
    - `male(adam): adam is male.`
  - RULES - statements about true relations which hold between objects in the world which contain generalizations, expressed through the use of variables. For example, the rule
    - `father(X,Y) :- parent(X,Y), male(X).`
    - might express:
      - for any X and any Y, X is the father of Y if X is a parent of Y and X is male.

Nomenclature and Syntax

- A prolog rule is called a **clause**.
- A clause has a head, a neck and a body:
  - `father(X,Y) :- parent(X,Y), male(X).`
    - head neck body
  - the **head** is a rule’s conclusion.
  - The **body** is a rule’s premise or condition.
  - note:
    - read `:-` as IF
    - read `,` as AND
    - a . marks the end of input
Prolog Database

**Facts** comprising the “extensional database”

parent(adam,able)
parent(adam,cain)
male(adam)
...

father(X,Y) :- parent(X,Y),
male(X).
sibling(X,Y) :- ...

**Rules** comprising the “intensional database”

Extension vs. Intensional

The terms *extensional* and *intensional* are borrowed from
the language philosophers use for *epistemology*.

- **Extension** refers to whatever extends, i.e., “is quantifiable in space as well as in time”.
- **Intension** is an antonym of extension, referring to “that class of existence which may be quantifiable in time but not in space.”
- NOT intensional with a “t”, which has to do with “will, volition, desire, plan, …”

For KBs and DBs we use

- extensional to refer to that which is explicitly represented (e.g., a fact), and
- intensional to refer to that which is represented abstractly, e.g., by a rule of inference.

A Simple Prolog Session

| ?- assert(parent(adam,able)).
| yes |
| ?- assert(parent(eve,able)).
| yes |
| ?- assert(male(adam)).
| yes |
| ?- parent(adam,able).
| yes |
| X = able |

A Prolog Session

| ?- mother(eve,Who).
| Who = cain |
| yes |
| ?- trace, mother(Who,cain).
| 2 Exit: parent(0,cain) |
| 2 Call: female(0) |
| 2 Fail: female |
| 2 Back to: parent(0,cain) |
| 5 Call: female(eve) |
| 5 Exit: female(eve) |
| 2 Exit: mother(eve,cain) Who = eve |
| yes |
?- sibling(X,Y).
X = able
Y = cain ;
X = cain
Y = able ;

How to Satisfy a Goal

Here is an informal description of how Prolog satisfies a goal (like father(adam,X)). Suppose the goal is G:

- if G = P, Q then first satisfy P, carry any variable bindings forward to Q, and then satisfy Q.
- if G = P; Q then satisfy P. If that fails, then try to satisfy Q.
- if G = not(P) then try to satisfy P. If this succeeds, then fail and if it fails, then succeed.
- if G is a simple goal, then look for a fact in the DB that unifies with G look for a rule whose conclusion unifies with G and try to satisfy its body.

Terms

- The term is the basic data structure in Prolog.
- The term is to Prolog what the s-expression is to Lisp.
- A term is either:
  - a constant - e.g.
    - john, 13, 3.1415, +, 'a constant'
  - a variable - e.g.
    - X, Var, __, _foo
  - a compound term - e.g.
    - part(arm, body)
    - part(arm(john), body(john))

Note

- two basic conditions are true, which always succeeds, and fail, which always fails.
- A comma (,) represents conjunction (i.e. and).
- A semi-colon represents disjunction (i.e. or), as in:
  - grandParent(X,Y) :- grandFather(X,Y); grandMother(X,Y).
- there is no real distinction between RULES and FACTS. A FACT is just a rule whose body is the trivial condition true. That is parent(adam,cain) and parent(adam,cain) :- true. are equivalent
- Goals can usually be posed with any of several combination of variables and constants:
  - parent(cain,able) - is Cain Able's parent?
  - parent(cain,X) - Who is a child of Cain?
  - parent(X,cain) - Who is Cain a child of?
  - parent(X,Y) - What two people have a parent/child relationship?
Compound Terms

- A compound term can be thought of as a relation between one or more terms:
  - part_of(finger, hand)
and is written as:
  - the relation name (called the principle functor) which must be a constant.
  - An open parenthesis
  - The arguments - one or more terms separated by commas.
  - A closing parenthesis.
- The number of arguments of a compound terms is called its arity.

<table>
<thead>
<tr>
<th>Term</th>
<th>arity</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>f(a)</td>
<td>1</td>
</tr>
<tr>
<td>f(a,b)</td>
<td>2</td>
</tr>
<tr>
<td>f(g(a),b)</td>
<td>2</td>
</tr>
</tbody>
</table>