PROLOG

References:

- Clocksin and Mellish, chapters 1–4, 6, and 8.

I will be using slides written by Professor Suzanne Stevenson, Rutgers University. Credits follow ...

Principles of Programming Languages: Computer Science 314 Lecture Notes

Spring 1998

Prof. Suzanne Stevenson

Lecture 21*

*Adapted with much gratitude from notes by Dr. Phil Stocks, who adapted the notes of Prof. Ellman and Prof. Patel-Schneider, originally from notes by Prof. Ryder

Logic Programming

Logic programming languages are not procedural or functional.

- Specify *relations* between objects
  - **`larger(3,2)`**
  - **`father(tom,jane)`**
- Separate logic from control:
  - Programmer declares **what** facts and relations are true
  - System determines **how** to use facts to solve problems
- Computation engine: theorem-proving and recursion (Unification, Resolution, Backward Chaining, Backtracking)
  - Higher-level than imperative languages

Basic Facts and Relations

A Prolog program starts with declarations of the basic facts

```
male(albert).  <------------------------- a fact
female(alice).          Facts are put in a file.
male(edward).
female(victoria).
parent(albert,edward).
parent(victoria,edward).
parent(albert,alice).
parent(victoria,alice).
| ?- [family].  <----------------------- loads file
yes
| ?- male(albert).  <-------------------- a query
yes
| ?- male(alice).
no
| ?- parent(albert,edward).
yes
| ?- parent(bullwinkle,edward).
no
```

Limited use: need variables and deductive rules.

Variables and Unification

```
| ?- female(X).
X = alice

| ?- female(X).
X = alice ; <------------ ';' means look further
X = victoria ;
no
```

X is **unified** to all possible values that make the query female(X) true.

```
| ?- parent(P,edward).
P = albert ;
P = victoria ;
no
```

P is unified to all possible values that make the query parent(P,edward) true.

$\Rightarrow$ search with pattern matching

In Prolog:

- All variables are capitalized
- All constants are in lower case
- All predicates are in lower case

Prolog Queries

A query is a proposed fact that is to be proven.

- If the query has no variables, returns yes/no.

- If the query has variables, returns appropriate values of variables (called a substitution).

Horn Clauses (Rules)

A Horn Clause is: $c \leftarrow h_1 \wedge h_2 \wedge h_3 \wedge \ldots \wedge h_n$

- Antecedents: conjunction of zero or more conditions which are atomic formulae in predicate logic

- Consequent: an atomic formula in predicate logic

Meaning of a Horn clause:

- "The consequent is true if the antecedents are all true"

- c is true if h_1, h_2, h_3, ..., and h_n are all true

Horn Clause Terminology

- Horn Clause = Clause
- Consequent = Goal = Head
- Antecedents = Subgoals = Tail
- Horn Clause with No Tail = Fact
- Horn Clause with Tail = Rule

In Prolog, a Horn clause

$$c \leftarrow h_1 \wedge \ldots \wedge h_n$$

is written

c :- h_1,   , h_n.

Syntax elements: ':-' ',' '.'

Prolog Horn Clause Examples

A Horn clause with no tail:

```
male(albert).
```

I.e., a fact: albert is a male dependent on no other conditions

A Horn clause with a tail:

```
father(albert,edward):-
  male(albert), parent(albert,edward).
```

I.e., a rule: albert is the father of edward if albert is male and albert is a parent of edward's.

Horn Clauses with Variables

Variables may appear in the antecedents **and** consequent of a Horn clause:

- c(X_1,...,X_n) :- h(X_1,...,X_m).

  "For all values of X_1,...,X_n, the formula c(X_1,...,X_n) is true if the formula h(X_1,...,X_n) is true"

- c(X_1,...,X_n) :- h(X_1,...,X_m,Y_1,...,Y_k).

  "For all values of X_1,...,X_n, the formula c(X_1,...,X_n) is true if there exist values of Y_1,...,Y_k such that the formula h(X_1,...,X_n,Y_1,...,Y_k) is true"

Examples:

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

(X is the father of Y if X is male and X is a parent of Y)

```
| ?- father(F,edward).
F = albert ;
no
```

```
child_of(C,P):- parent(P,C).
```

```
| ?- child_of(C,P).
C = edward
P = albert ;

C = edward
P = victoria ;

C = alice
P = albert ;

C = alice
P = victoria ;

no
```

Example Prolog Clauses

```
sibling(X,Y):- parent(P,X), parent(P,Y).

| ?- sibling(alice,A).
A = edward ;
A = alice ;
A = edward ;
A = alice ;
no
```


Rule Ordering and Unification

1. rule ordering used in search
2. unification requires two instances of the same variable to get the same value
3. unification does not require differently named variables to get different values: hence, sibling(edward,edward).
4. all rules searched if requested by ';'

```
| ?- sibling(A,alice).
A = edward ;
A = edward ;
A = alice ;
A = alice ;
no
```

Note: arguments are interchangeable, but ordering affects order of search.


```
brother(X,Y):- male(X), sibling(X,Y).

sister(X,Y):- female(X), sibling(X,Y).

uncle(X,Y):- brother(X,Z), parent(Z,Y).

aunt(X,Y):- sister(X,Z), parent(Z,Y).

nephew(X,Y):- male(X), uncle(Y,X).
nephew(X,Y):- male(X), aunt(Y,X).

niece(X,Y):- female(X), uncle(Y,X).
niece(X,Y):- female(X), aunt(Y,X).
```

Note: “disjunction” (logical-or) is achieved by defining multiple clauses.

Transitive Relations

```
parent(sally,jane).     parent(bob,jane).
parent(sally,john).     parent(bob,john).
parent(mary,sally).     parent(al,sally).
parent(ann,bob).        parent(mike,bob).
parent(jean,al).        parent(joe,al).
parent(ruth,mary).      parent(jim,mary).
parent(esther,ruth).    parent(mick,ruth).

grandparent(X,Y) :- parent(X,Z), parent(Z,Y).

ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).

| ?- grandparent(Y,jane).
Y = mary ;
Y = al ;
Y = ann ;
Y = mike ;
no

| ?- ancestor(jane,X).
X = sally ;
X = bob ;
X = mary ;
X = al ;
X = ruth ;
X = jim ;
X = ester ;
X = mick ;
X = jean ;
X = joe ;
X = ann ;
X = mike ;
no
```


Procedural Semantics of Prolog

```
uncle(X,jane) :- male(X), sibling(X,Y),
                 parent(Y,jane).
```

In order to find an X to make $uncle(X, jane)$ true:

1. First find an X to make $male(X)$ true
2. Then find a Y such that $sibling(X, Y)$ is true
3. Then check that $parent(Y, jane)$ is true

Procedural Semantics of Prolog

Notice the recursion in this algorithm: “find” calls “find”. This reasoning is recursively applied until we reach rules that are facts.

This process is called **Backward Chaining**.

Logic Programming vs. Prolog

```
cousin(X,jane) :- parent(Y,jane), sister(Y,Z),
                  parent(Z,X).

cousin(X,jane) :- parent(Y,jane), brother(Y,Z),
                  parent(Z,X).
```

Rule and Goal Ordering:

- There are two rules for cousin
- Which rule do we try first?
- Each rule for cousin has several subgoals
- Which subgoal do we try first?

Logic Programming vs. Prolog

Logic Programming: *Nondeterministic*

- Arbitrarily choose rule to expand first
- Arbitrarily choose subgoal to to explore first
- Results don't depend on rule and subgoal ordering

Prolog: *Deterministic*

- Expand first rule first
- Explore first subgoal first
- Results may depend on rule and subgoal ordering

(Minimal) Prolog Syntax

<rule> ::= <head> :- <body> . |
<fact> .

<head> ::= <predicate>

<fact> ::= <predicate>

<body> ::= <predicate> { , <predicate> }

<predicate> ::=

<functor> '(' <term> { , <term> } ')'

<term> ::= <integer> | <atom> |
<variable> | <predicate>

<query> ::= '|' ?- <predicate>

Lists in Prolog

- '[' List Elems Separated by Commas ']'
- '[' First Elem '|' Rest of List ']'

List	head	tail	equivalent
[a,b,c]	a	[b,c]	
[a]	a	[]	
[X,[cat],Y]	X	[[cat],Y]	
[[a,b],c,d]	[a,b]	[c,d]	

Unifying Lists

[X,Y,Z]	[john,likes,fish]
[cat]	[X | Y]
[1,2]	[X | Y]
[1 | 2]	[X , Y]
[a,b,c]	[X|Y]
[a,b|Z]	[X|Y]
[X,abc,Y]	[X,abc|Y]
[[the | Y] | Z]	[[X,hare] | [is,here]]

"Appending" Lists

Definition of **`append`**...

Build a list:

| ?- append([a],[b],Y).
Y=[a,b]

Break a list up:

| ?- append(X,[b],[a,b]).
X=[a]
| ?- append([a],Y,[a,b]).
Y=[b]
| ?- append(X,Y,[a,b]).
X=[],Y=[a,b];
X=[a],Y=[b];
X=[a,b],Y=[];
no

Generate lists:

| ?- append(X,[b],Z).
X=[],Z=[b] ;
X=[_169],Z=[_169,b] ;
X=[_169,170],Z=[_169,_170,b] ;
...

List Membership

Definition of **`member`**...

| ?- member(a,[a,b]).
yes
| ?- member(a,[b,c]).
no
| ?- member(X,[a,b,c]).
X=a ;
X=b ;
X=c ;
no
| ?- member(a,[c,b,X]).
X=a ;
no
| ?- member(X,Y).
X=_123, Y=[X|_124] ;
X=_123, Y=[_125,X|_126] ;
X=_123, Y=[_127,_128,X|_129] ;
...

Lazy evaluation of potentially infinite data structures