LOGIC PROGRAMMING AND PROLOG

Reading:

- Sebesta, chapter 16

References:

- Clocksin and Mellish, Ch. 1-4, 6, 8 (on hold in library)
- Online Resources (tutorials, SWI page, etc.)

Logic Programming and Prolog

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.
  - System **instantiates** variables in order to make relations true!
- Computation engine: theorem-proving and recursion (Unification, Resolution, Backward Chaining, Backtracking)
  - Higher-level than imperative languages

Jumping Right In

Suppose we state these facts:

```
male(albert).           parent(albert,edward).
female(alice).          parent(victoria,edward).
male(edward).           parent(albert,alice).
female(victoria).       parent(victoria,alice).
```

We can then make queries:

```
?- male(albert).
Yes

?- male(victoria).
No

?- female(Person).
Person = alice;
Person = victoria;
No

?- parent(Person, edward).
Person = albert;
Person = victoria;
No

?- parent(Person, edward), female(Person).
Person = victoria;
No
```


We can also state rules, such as this one:

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

Then the queries become more interesting:

```
?- sibling(albert, victoria).
No

?- sibling(edward, Sib).
Sib = edward;
Sib = alice;
Sib = edward;
Sib = alice;
No
```


Prolog vs Scheme

In Scheme, we program with **functions** ("procedures").

- A function's arguments are different from the function's value.
- Give a single Scheme function, we can only ask one kind of question:

  Here are the argument values; tell me what is the function's value.

In Prolog, we program with **relations**.

- There is no bias; all arguments are the same.
- Given a single Prolog predicate, we can ask many kinds of question:

  Here are some of the argument values; tell me what the others have to be in order to make a true statement.

Logic Programming

- A program consists of facts and rules.
- Running a program means asking queries.
- The language tries to find one way (or more) to prove that the query is true.
- This may have the side effect of freezing variable values.
- The language determines how to do all of this, *not* the program.
- How does the language do it? Using unification, resolution, and backtracking.

The swi Interface on cdf

```
cdf% ls
family.pl

cdf% pl
Welcome to SWI-Prolog (Multi-threaded, Version 5.2.11)
Copyright (c) 1990-2003 University of Amsterdam.
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free sof
and you are welcome to redistribute it under certain conditior
Please visit http://www.swi-prolog.org for details.

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

?- ['family'].   <------------- load file family.pl
% family compiled 0.00 sec, 5,264 bytes
Yes


?- parent(Person, edward).

Person = albert;   <----------- ";" to get more
Person = victoria;
No


?- parent(Person, edward).

Person = albert   <--------------"a", CR, space to break
Yes
```



```
?- trace.
Yes
[trace]

[trace] ?- parent(Person, edward).

[trace]  ?- parent(Person, edward).
   Call: (7) parent(_G283, edward) ? creep  <- CR to continue
   Exit: (7) parent(albert, edward) ? creep

Person = albert ;
   Redo: (7) parent(_G283, edward) ? creep
   Exit: (7) parent(victoria, edward) ? creep

Person = victoria

Yes
[debug]  ?-


   (0) Call: parent(_57,edward) ?
   (0) Exit: parent(albert,edward) ?
Person = albert;
   (0) Redo: parent(albert,edward) ?
   (0) Exit: parent(victoria,edward) ?
Person = victoria
Yes
[trace]
```

```
?- notrace.
Yes

?- parent(Person, edward).

Person = albert;
Person = victoria;
No


?- halt.
cdf%
```

```
cdf% pl
Welcome to SWI-Prolog (Multi-threaded, Version 5.2.11) ...

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

?- [family].
[family loaded]
Yes

?- parent(Person, edward).

Person = albert;
Person = victoria;
No

--- edit family.P and remove parent(albert,edward). ---

?- ['family'].

% family compiled 0.00 sec, 5,200 bytes
Yes

?- parent(Person, edward).

Person = victoria;
No

?- halt.
cdf%
```


Some Prolog Syntax

Lexical Rules:

- Variables are capitalized.
- Constants begin with a lower case letter.
- Predicate names begin with a lower case letter.

Simplified Grammar:

```
<clause> ::= <pred> |

             <pred> :- <pred> { , <pred> } .


<pred>   ::= <pname>'(' <term> { , <term> } ')'


<term>   ::= <int> | <atom> | <var>
```

Note: No blank between predicate name and opening bracket.

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.

Meaning of Prolog Rules Without Variables

A prolog rule must have this form:

$$c\ :-\ a_1,\ \ a_2,\ \ a_3,\ \ \cdots,\ \ a_n.$$

which means in logic:

$$a_1 \wedge a_2 \wedge a_3 \wedge \cdots \wedge a_n \rightarrow c.$$

Restrictions

- There can be zero or more antecedents, but they are conjoined; we cannot disjoin them.
- There cannot be more than 1 consequent.

Bending the Restrictions

Getting disjoined antecedents

Example: $a_1 \vee a_2 \vee a_3 \vee \to c$.

Solution:

Getting more than 1 consequent, conjoined

Example: $a_1 \wedge a_2 \wedge a_3 \to c_1 \wedge c_2$.

Solution:

Getter more than 1 consequent, disjoined

Example: $a_1 \wedge a_2 \wedge a_3 \to c_1 \vee c_2$.

Solution:

Why Can't We Disjoin Consequents?

Why did the designers of Prolog disallow this?

Logic Review

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_n).

  "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_n,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,.. is true"

Meaning of Prolog Rules With Variables

Example:

```
isaMother(X) :- female(X), parent(X, Y).
```

Logic:

$$parent(X, Y) \wedge female(X) \supset isaMother(X).$$

But this is meaningless without quantifiers for the variables.

The rule

A Prolog rule of this form $(n \geq 0, m \leq n, k \geq o)$:

$$c(X_1, \cdots X_n)\ :-\ a(X_1, \cdots X_m, Y_1, \cdots Y_k).$$

means:

$\forall X_1, \cdots X_n$

$[\exists Y_1, \cdots Y_k \quad [a(X_1, \cdots X_m, Y_1, \cdots Y_k) \supset c(X_1, \cdots X_n)] \quad]$

Sample run

```
cdf% pl
Welcome to SWI-Prolog (Multi-threaded, Version 5.2.11)...

?- ['family'].

Warning: (./family.pl:50):
        Singleton variables: [Y]
% family compiled 0.00 sec, 5,528 bytes

?- isaMother(X).
X = victoria;
X = victoria;
No
```