

**Rule-based
Programming, Logic
Programming
and Prolog**

What is Logic Programming?

There are many (overlapping) perspectives on logic programming

- Computations as Deduction
- Theorem Proving
- Non-procedural Programming
- Algorithms minus Control
- A Very High Level Programming Language
- A Procedural Interpretation of Declarative Specifications

The Paradigm

- An important programming paradigm is to express a program as a set of rules
- The rules are independent and often unordered
- CFGs can be thought of as a rule based system
- We'll take a brief look at a particular sub-paradigm, [Logic Programming](#)
- And at Prolog, the most successful of the logic programming languages

History

- Logic Programming has roots going back to early AI researchers like John McCarthy in the 50s & 60s
- [Alain Colmerauer](#) (France) designed [Prolog](#) as the first LP language in the early 1970s
- [Bob Kowalski](#) and colleagues in the UK evolved the language to its current form in the late 70s
- It's been widely used for many AI systems, but also for systems that need a fast, efficient and clean rule based engine
- The prolog model has also influenced the database community – see [datalog](#)

Computation as Deduction

- Logic programming offers a slightly different paradigm for computation: *computation is logical deduction*

- It uses the language of logic to express data and programs.

For all X, Y : X is the father of Y if X is a parent of Y and X is male

- Current logic programming languages use first order logic (FOL) which is often referred to as first order predicate calculus (FOPC).

- The *first order* refers to the constraint that we can quantify (i.e. generalize) over objects, but not over functions or relations. We can express "*All elephants are mammals*" but not

"for every continuous function f , if $n < m$ and $f(n) < 0$ and $f(m) > 0$ then there exists an x such that $n < x < m$ and $f(x) = 0$ "

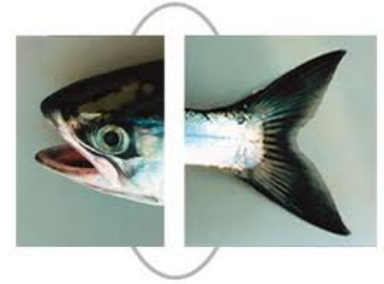
Theorem Proving

- Logic Programming uses the notion of an *automatic theorem prover* as an interpreter.
- The theorem prover derives a desired solution from an initial set of axioms.
- The proof must be a "constructive" one so that more than a true/false answer can be obtained
- E.G. The answer to
exists x such that $x = \text{sqrt}(16)$
- should be
 $x = 4$ or $x = -4$
- rather than
true

Non-procedural Programming

- Logic Programming languages are non-procedural programming languages
- A non-procedural language one in which one specifies **what** needs to be computed but not **how** it is to be done
- That is, one specifies:
 - the set of objects involved in the computation
 - the relationships which hold between them
 - the constraints which must hold for the problem to be solved
- and leaves it up to the language interpreter or compiler to decide **how** to satisfy the constraints

A Declarative Example



- Here's a simple way to specify what has to be true if X is the smallest number in a list of numbers L
 1. X has to be a member of the list L
 2. There can't be list member X_2 such that $X_2 < X$
- We need to say how we determine that some X is a member of a list
 1. No X is a member of the empty list
 2. X is a member of list L if it is equal to L 's head
 3. X is a member of list L if it is a member of L 's tail.

A Simple Prolog Model

Think of 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 relations between objects in the world which use variables to express generalizations

```
% X is the father of Y if X is a parent of Y and X is male
```

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

```
% X is a sibling of Y if X and Y share a parent
```

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

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 single predicate -- the rule's conclusion
- The **body** is a a sequence of zero or more predicates that are the rule's premise or condition
- An empty body means the rule's head is a fact.
- note:
 - read :- as IF
 - read , as AND between predicates
 - a . marks the end of input

Prolog Database

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

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

Facts comprising the
“extensional database”

Rules comprising the
“intensional database”

Queries

- We also have queries in addition to having facts and rules
- The Prolog REPL interprets input as queries
- A simple query is just a predicate that might have variables in it:
 - `parent(adam, cain)`
 - `parent(adam, X)`

Extensional 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 *intentional* 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.

Prolog Database

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

...

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

Facts comprising the
“extensional database”

Rules comprising the
“intensional database”

Epistemology is “a branch of philosophy that investigates the origin, nature, methods, and limits of knowledge”

Running prolog

- A good free version of prolog is [swi-prolog](#)
- GL has a commercial version ([sicstus prolog](#))
you can invoke with the command “sicstus”

```
[finin@linux2 ~]$ sicstus
```

```
SICStus 3.7.1 (Linux-2.2.5-15-i686): Wed Aug 11 16:30:39 CEST 1999
```

```
Licensed to umbc.edu
```

```
| ?- assert(parent(adam,able)).
```

```
yes
```

```
| ?- parent(adam,P).
```

```
P = able ?
```

```
yes
```

```
| ?-
```

A Simple Prolog Session

| ?- assert(parent(adam,able)).

yes

| ?- assert(parent(eve,able)).

yes

| ?- assert(male(adam)).

yes

| ?- parent(adam,able).

yes

| ?- parent(adam,X).

X = able

yes

| ?- parent(X,able).

X = adam ;

X = eve ;

no

| ?- parent(X,able) , male(X).

X = adam ;

no

A Prolog Session

```
| ?- [user].  
| female(eve).  
| parent(adam,cain).  
| parent(eve,cain).  
| father(X,Y) :- parent(X,Y), male(X).  
| mother(X,Y) :- parent(X,Y), female(X).  
|^Zuser consulted 356 bytes 0.0666673  
  sec.  
yes  
| ?- mother(Who,cain).  
Who = eve  
yes
```

```
| ?- mother(eve,Who).  
Who = cain  
yes  
| ?- trace, mother(Who,cain).  
  (2) 1 Call: mother(_0,cain) ?  
  (3) 2 Call: parent(_0,cain) ?  
  (3) 2 Exit: parent(adam,cain)  
  (4) 2 Call: female(adam) ?  
  (4) 2 Fail: female(adam)  
  (3) 2 Back to: parent(_0,cain) ?  
  (3) 2 Exit: parent(eve,cain)  
  (5) 2 Call: female(eve) ?  
  (5) 2 Exit: female(eve)  
  (2) 1 Exit: mother(eve,cain)  
Who = eve  
yes
```

```
| ?- [user].
| sibling(X,Y) :-
|   father(Pa,X),
|   father(Pa,Y),
|   mother(Ma,X),
|   mother(Ma,Y),
|   not(X=Y).
```

^Zuser consulted 152 bytes 0.0500008 sec.

```
yes
| ?- sibling(X,Y).
X = able
Y = cain ;
X = cain
Y = able ;
```

```
trace,sibling(X,Y).
(2) 1 Call: sibling(_0,_1) ?
(3) 2 Call: father(_65643,_0) ?
(4) 3 Call: parent(_65643,_0) ?
(4) 3 Exit: parent(adam,able)
(5) 3 Call: male(adam) ?
(5) 3 Exit: male(adam)
(3) 2 Exit: father(adam,able)
(6) 2 Call: father(adam,_1) ?
(7) 3 Call: parent(adam,_1) ?
(7) 3 Exit: parent(adam,able)
(8) 3 Call: male(adam) ?
(8) 3 Exit: male(adam)
(6) 2 Exit: father(adam,able)
(9) 2 Call: mother(_65644,able) ?
(10) 3 Call: parent(_65644,able) ?
(10) 3 Exit: parent(adam,able)
(11) 3 Call: female(adam) ?
(11) 3 Fail: female(adam)
(10) 3 Back to: parent(_65644,able) ?
(10) 3 Exit: parent(eve,able)
(12) 3 Call: female(eve) ?
(12) 3 Exit: female(eve)
(9) 2 Exit: mother(eve,able)
(13) 2 Call: mother(eve,able) ?
(14) 3 Call: parent(eve,able) ?
(14) 3 Exit: parent(eve,able)
(15) 3 Call: female(eve) ?
(15) 3 Exit: female(eve)
(13) 2 Exit: mother(eve,able)
(16) 2 Call: not able=able ?
(17) 3 Call: able=able ?
(17) 3 exit: able=able
(16) 2 Back to: not able=able ?
(16) 2 Fail: not able=able
(15) 3 Back to: female(eve) ?
(15) 3 Fail: female(eve)
(14) 3 Back to: parent(eve,able) ?
(14) 3 Fail: parent(eve,able)
(13) 2 Back to: mother(eve,able) ?
(13) 2 Fail: mother(eve,able)
(12) 3 Back to: female(eve) ?
(12) 3 Fail: female(eve)
(10) 3 Back to: parent(_65644,able) ?
(10) 3 Fail: parent(_65644,able)
(9) 2 Back to: mother(_65644,able) ?
(9) 2 Fail: mother(_65644,able)
(8) 3 Back to: male(adam) ?
(8) 3 Fail: male(adam)
(7) 3 Back to: parent(adam,_1) ?
(7) 3 Exit: parent(adam,cain)
(18) 3 Call: male(adam) ?
(18) 3 Exit: male(adam)
(6) 2 Exit: father(adam,cain)
(19) 2 Call: mother(_65644,able) ?
(20) 3 Call: parent(_65644,able) ?
(20) 3 Exit: parent(adam,able)
(21) 3 Call: female(adam) ?
(21) 3 Fail: female(adam)
(20) 3 Back to: parent(_65644,able) ?
(20) 3 Exit: parent(eve,able)
(22) 3 Call: female(eve) ?
(22) 3 Exit: female(eve)
(19) 2 Exit: mother(eve,able)
(23) 2 Call: mother(eve,cain) ?
(24) 3 Call: parent(eve,cain) ?
(24) 3 Exit: parent(eve,cain)
(25) 3 Call: female(eve) ?
(25) 3 Exit: female(eve)
(23) 2 Exit: mother(eve,cain)
(26) 2 Call: not able=cain ?
(27) 3 Call: able=cain ?
(27) 3 Fail: able=cain
(26) 2 Exit: not able=cain
(2) 1 Exit: sibling(able,cain)
```

```
X = able
Y = cain
yes no
| ?-
```

Program files

Typically you put your assertions (fact and rules) into a file and load it

```
| ?- [genesis].
{consulting /afs/umbc.edu/users/f/i/finin/home/genesis.pl...}
{/afs/umbc.edu/users/f/i/finin/home/genesis.pl consulted, 0 msec 2720
bytes}
yes
| ?- male(adam).
yes
| ?- sibling(P1, P2).
P1 = cain,
P2 = cain ? ;
P1 = cain,
P2 = able ? ;
P1 = cain,
P2 = cain ? ;
P1 = cain,
P2 = able ? ;
P1 = able,
P2 = cain ? ;
P1 = able,
P2 = able ? ;
P1 = able,
P2 = cain ? ;
P1 = able,
P2 = able ? ;
no
| ?-
```

```
[finin@linux2 ~]$ more genesis.pl
% prolog example

% facts
male(adam).
female(eve).
parent(adam,cain).
parent(eve,cain).
parent(adam,able).
parent(eve,able).

% rules
father(X,Y) :-
    parent(X,Y),
    male(X).
mother(X,Y) :-
    parent(X,Y),
    female(X).
sibling(X,Y) :-
    parent(P, X),
    parent(P, Y).
child(X, Y) :- parent(Y, X).
```

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 $G = \text{not}(P)$ 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

Note

- Two basic conditions are true, which always succeeds, and fail, which always fails.
- Comma (,) represents conjunction (i.e. and).
- Semi-colon represents disjunction (i.e. or):
grandParent(X,Y) :-
 grandFather(X,Y);
 grandMother(X,Y).
- No real distinction between rules and facts. A fact is just a rule whose body is the trivial condition true. These are equivalent:
–*parent(adam,cain)*.
–*parent(adam,cain) :- true*.

Note

- 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?

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))

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.

Term	arity
f	0
f(a)	1
f(a,b)	2
f(g(a),b)	2

Lists

- Lists are so useful there is special syntax to support them, tho they are just terms
- It's like Python: `[1, [2, 3], 4, foo]`
- But matching is special
 - If $L = [1, 2, 3, 4]$ then $L = [Head \mid Tail]$ results in Head being bound to 1 and Tail to $[2, 3, 4]$
 - If $L = [4]$ then $L = [Head \mid Tail]$ results in Head being bound to 4 and Tail to $[\]$

member

% member(X,L) is true if X is a member of list L.

member(X, [X|Tail]).

member(X, [Head|Tail]) :- member(X, Tail).

min

% min(X, L) is true if X is the smallest member
% of a list of numbers L

min(X, L) :-

member(X, L),

\+ (member(Y,L), Y>X).

- \+ is Prolog's negation operator
- It's really "negation as failure"
- \+ G is false if goal G can be proven
- \+ G is true if G can not be proven
- i.e., assume its false if you can not prove it to be true

Computations

- Numerical computations can be done in logic, but its messy and inefficient
- Prolog provides a simple limited way to do computations
- $\langle \text{variable} \rangle$ is $\langle \text{expression} \rangle$ succeeds if $\langle \text{variable} \rangle$ can be unified with the value produced by $\langle \text{expression} \rangle$

?- X=2, Y=4, Z is X+Y.

X = 2,

Y = 4,

Z = 6.

?- X=2, Y=4, X is X+Y.

false.

From Functions to Relations

- Prolog facts and rules define *relations*, not *functions*
- Consider age as:
 - A function: calling *age(john)* returns 22
 - As a relation: querying *age(john, 22)* returns true, *age(john, X)* binds *X* to 22, and *age(john, X)* is false for every $X \neq 22$
- Relations are more general than functions
- The typical way to define a function **f** with inputs $\mathbf{i}_1 \dots \mathbf{i}_n$ and output **o** is as: **f(i₁,i₂,...i_n,o)**

A numerical example

- Here's how we might define the factorial relation in Prolog.

fact(1,1).

fact(N,M) :-

N > 1,

N1 is N-1,

fact(N1,M1),

M is M1*N.

```
def fact(n):  
    if n==1:  
        return 1  
    else:  
        n1 = n-1  
        m1 = fact(n1)  
        m = m1 * n  
        return m
```

Another example:

square(X,Y) :- Y is X*X.

Prolog = PROgramming in LOGic

- Prolog is as much a programming language as it is a theorem prover
- It has a simple, well defined and controllable reasoning strategy that programmers can exploit for efficiency and predictability
- It has basic data structures (e.g., Lists) and can link to routines in other languages
- It's a great tool for many problems