Rule-based Programming, Logic Programming and Prolog

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 = 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 is one in which we specify 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 the 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).

(note: `% is Prolog's comment character)

**Nomenclature and Syntax**

- A prolog rule is called a **clause**
- A clause has a head, a neck and a body:

  ```prolog
  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
  - `.` marks the end of input

**Prolog Database**

- **Facts** comprising the "extensional database"

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

- **Rules** comprising the "intensional database"

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

**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)`

**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"
  ```bash
  finni@linux2 ~> sicstus
  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
?- parent(adam,cain).
yes
?- parent(adam,cain).
yes

Two basic conditions are true, which always succeed, and fail, which always fails.

- if G = P, Q then first satisfy P, carry any variable bindings forward to Q, and then satisfy Q.
- if G = P, Q then fail. 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, or look for a rule whose conclusion unifies with G and try to satisfy its body.

A Prolog Session

?- female(eve).
- female(eve).
- father(X,Y) :- parent(X,Y), male(X).
- mother(X,Y) :- parent(X,Y), female(X).

No real distinction between rules and facts. A fact is just a rule whose body is the trivial condition true. These are equivalent:

- father(adam, cain).
- parent(adam, cain).
- parent(adam, cain) :- true.

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 fail. 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, or look for a rule whose conclusion unifies with G and try to satisfy its body.

Note

- Two basic conditions are true, which always succeed, 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).
- Program files

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

% 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).

% facts
male(adam).
male(eve).
female(adam).
female(eve).
female(adam).
female(adam).
male(adam).

Program files

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

% 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).

% facts
male(adam).
male(eve).
female(adam).
female(eve).
female(adam).
female(adam).
male(adam).

Program files

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

% 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).

% facts
male(adam).
male(eve).
female(adam).
female(eve).
female(adam).
female(adam).
male(adam).

Program files

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

% 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).

% facts
male(adam).
male(eve).
female(adam).
female(eve).
female(adam).
female(adam).
male(adam).
**Note**

- Goals can usually be posed with any of several combinations 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 principal 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>

**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 | Tail]` results in `Head` being bound to `1` and `Tail` to `[2, 3, 4]`
  - If `L = [4]` then `L = [Head | 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),
  \( \lor \) (member(Y,L), Y>X).
Computations

- Numerical computations can be done in logic, but it's messy and inefficient
- Prolog provides a simple limited way to do computations
- `<variable>` is `<expression>` succeeds if `<variable>` can be unified with the value produced by `<expression>`

```
?- 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 ≠ 22
- Relations are more general than functions
- The typical way to define a function f with inputs i₁…iₙ and output o is as: `f(i₁,i₂,…,iₙ,o)`

A numerical example

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

```prolog
fact(1,1).
fact(N,M) :-
    N > 1,
    N1 is N-1,
    fact(N1,M1),
    M is M1 * N.
```

Another example:

```prolog
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