Logic as a Programming Language

Dr Lutz Hamel

Dept. of Computer Science & Statistics

Logic as a Programming Language

- Here we take a look at First Order Logic as a computational model.
- Resources:
  - https://github.com/lutzhamel/phl-prolog
  - https://www.ida.liu.se/~ulfni53/lpp/
  - http://www.doc.ic.ac.uk/~rak/papers/LogicForProblemSolving.pdf
  - http://www.swi-prolog.org

Logic as a Programming Language

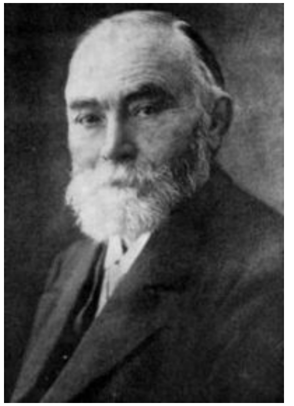
- Fundamental Question:
  - Can First Order Logic be considered a computational model?
- Another way of asking the same question:
  - Is First Order Logic *Turing Complete?*

Turing Completeness: A system of data-manipulation rules is said to be *Turing complete* if it can be used to simulate any Turing machine (*i.e.* compute any algorithm).

-- Wikipedia

Quantification

- In 1879 Gottlob Frege introduced the predicate calculus (‘Begriffsschrifft’)
- Today predicate calculus is more commonly known as First Order Logic (FOL).
- This logic is characterized by three structures:
  - predicates,
  - universal quantification, and
  - existential quantification.



Friedrich Ludwig Gottlob Frege
Philosopher and Logician

First-Order Logic

- Quantified Variables
  - Universally quantified variables

    $\forall$X – for All objects X

  - Existentially quantified variables

    $\exists$Y – there Exists an object Y

First-Order Logic

- Predicates
  - Predicates are functions that map their arguments into true/false
  - The signature of a predicate p(X) is

    p: Objects $\rightarrow$ { true, false }

  - Example: human(X)
    - human: Objects $\rightarrow$ { true, false }
    - human(tree) = false
    - human(paul) = true
  - Example: mother(X,Y)
    - mother: Objects $\times$ Objects $\rightarrow$ { true, false }
    - mother(betty,paul) = true
    - Mother(giraffe,peter) = false

First-Order Logic

- We can combine predicates and quantified variables to make statements on sets of objects
  - $\exists X[mother(X,paul)]$
    - there exists an object X such that X is the mother of Paul
  - $\forall Y[human(Y)]$
    - for all objects Y such that Y is human

First-Order Logic

- Logical Connectives: and, or, not
  - $\exists F \forall C$[parent(F,C) and male(F)]
    - There exists an object F for all object C such that F is a parent of C and F is male.
  - $\forall X$[day(X) and (wet(X) or dry(X))]
    - For all objects X such that X is a day and X is either wet or dry.

First-Order Logic

- If-then rules: $A \rightarrow B$
  - $\forall X \forall Y[\text{parent}(X,Y) \text{ and female}(X) \rightarrow \text{mother}(X,Y)]$
    - For all objects X and for all objects Y such that if X is a parent of Y and X is female then X is a mother.
  - $\forall Q[\text{human}(Q) \rightarrow \text{mortal}(Q)]$
    - For all objects Q such that if Q is human then Q is mortal.

Horn Clause Logic

In Horn clause logic the form of the WFF's is restricted:

$$P_1 \wedge P_2 \wedge \ldots \wedge P_{n-1} \wedge P_n \rightarrow P_0$$

Single predicate in consequent

Conjunctions only!

Where P_0 , P_1 , P_2, … P_{n-1}, P_n are predicates.

Horn Clause Logic is a *Turing complete* subset of First Order Logic.

Horn Clause Logic

- Proof outline that Horn Clause Logic (HCL) is Turing complete:
  - $\text{HCL} \Leftrightarrow \mu\text{−recursive functions} \Leftrightarrow \text{Turing Machine}$

Computational Logic

- Horn Clauses
  - Restricted FOL
- Unification
  - *Pattern matching* where both pattern and subject term can have variables.
  - *E.g.* f(X,g(b)) ~ f(a,g(Y)) : { X = a, Y = b }
- (SLD) Resolution
  - Modus Ponens kind of reasoning.
  - Think: search top to bottom, left to right, *backtrack* if you can't find what you are looking for.
  - **Closed World Assumption:** assumes false if the entity Prolog is searching for cannot be found/deduced.

→ Prolog

Proving things is computation!

Advantage: All this can be done mechanically (Alan Robinson, 1965)

$\boxed{\text{“Deduction is Computation”}}$

J. Alan Robinson: A Machine-Oriented Logic Based on the Resolution Principle. J. ACM 12(1): 23-41 (1965)

Basic Prolog Programs

Facts - a fact constitutes a declaration of a truth; in Prolog it has to to be a positive assertion.

Prolog Programs - a Prolog program is a collection of facts (…and rules, as we will see later).

Example: a simple program

```
male(phil).
male(john).
female(betty).
```

Facts, Prolog will treat these as true and enters them into its knowledgebase.

We execute Prolog programs by posing queries on its knowledgebase:

Prompt

```
?- male(phil).
   true - because Prolog can use its knowledgebase to prove true.
?- female(phil).
   false - this fact is not in the knowledgebase.
```

Prolog ≡ Programming in Logic

Prolog - Queries & Goals

A query is a way to extract information from a logic program.

Given a query, Prolog attempts to show that the query is a logical consequence of the program; of the collection of facts.

In other words, a query is a goal that Prolog is attempting to satisfy (prove true).

When queries contain variables they are existentially quantified, consider

```
?- parent(X,liz).
```

The interpretation of this query is: prove that there is at least one object X that can be considered a parent of liz, or formally, prove that

$$\exists x[parent(x,liz)]$$

holds.

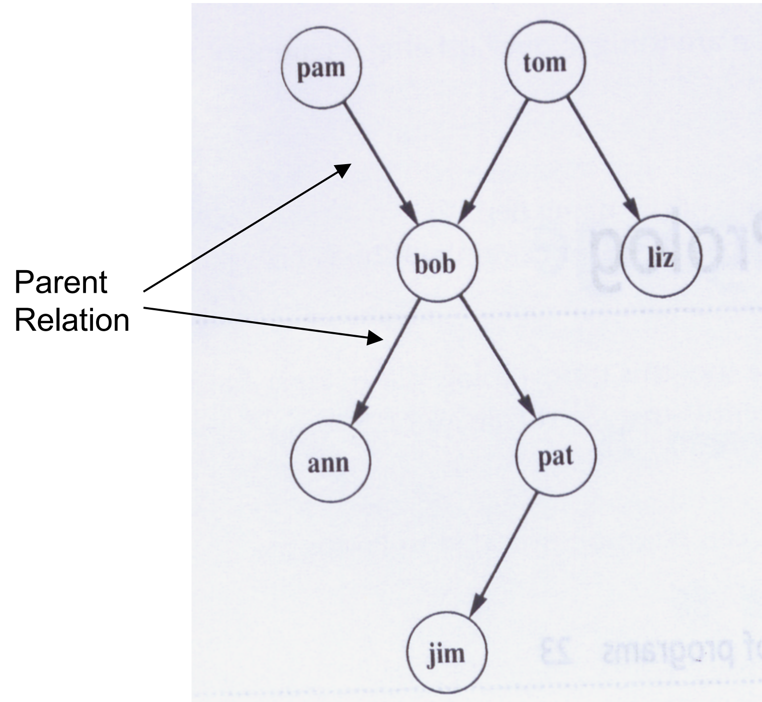
NOTE: Prolog will return all objects for which a query evaluates to true.

A Prolog Program

```
% a simple prolog program
female(pam).
female(liz).
female(ann).
female(pat).

male(tom).
male(bob).
male(jim).

parent(pam,bob).
parent(tom,bob).
parent(tom,liz).
parent(bob,ann).
parent(bob,pat).
parent(pat,jim).
```



A Family Tree

Example Queries:

```
?- female(pam).
?- female(X).        ∃X[female(X)]?
?- parent(tom,Z).
?- father(Y).
```

Compound Queries

A compound query is the conjunction of individual simple queries.

Stated in terms of goals: a compound goal is the conjunction of individual subgoals each of which needs to be satisfied in order for the compound goal to be satisfied. Consider:

?- parent(X,Y) , parent(Y,ann).

or formally,

$\exists X,Y[\text{parent}(X,Y) \wedge \text{parent}(Y,\text{ann})]$

When Prolog tries to satisfy this compound goal, it will make sure that <u>the two Y variables always have the same values</u>.

Prolog uses <u>unification</u> and <u>backtracking</u> in order to find all the solutions which satisfy the compound goal.

Prolog Rules

Prolog rules are Horn clauses, but they are written "backwards", consider:

$$\forall X,Y[female(X) \wedge parent(X,Y) \rightarrow mother(X,Y)]$$

is written in Prolog as

Implies ("think of $\leftarrow$")

$$\underbrace{mother(X,Y)}_{\text{head}} \text{ :- } \underbrace{female(X) , parent(X,Y)}_{\text{body}} \text{ .}$$

"and"

Prolog rules a implicitly universally quantified!

You can think of a rule as introducing a new "fact" (the head), but the fact is defined in terms of a compound goal (the body). That is, predicates defined as rules are only true if the associated compound goal can be shown to be true.

Prolog Rules

```
% a simple prolog program
female(pam).
female(liz).
female(ann).
female(pat).

male(tom).
male(bob).
male(jim).

parent(pam,bob).
parent(tom,bob).
parent(tom,liz).
parent(bob,ann).
parent(bob,pat).
parent(pat,jim).

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

Queries:

```
?- mother(pam,bob).
?- mother(Z,jim).
?- mother(P,Q).
```

Prolog Rules

The same predicate name can be defined by multiple rules:

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

Use backtracking to find alternative definitions.

Another Simple Prolog Program

Consider the program relating humans to mortality:

```
mortal(X) :- human(X).
human(socrates).
```

We can now pose the query:

```
?- mortal(socrates).
```

True or false?

Exercise

- See GitHub Exercise #1

Declarative vs. Procedural Meaning

When interpreting rules purely as Horn clause logic statement → <u>declarative</u>

When interpreting rules as "specialized queries" → <u>procedural</u>

<u>Observation</u>: We design programs with declarative meaning in our minds, but the execution is performed in a procedural fashion.

Consider:

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

Lists & Pattern Matching

- The unification operator: =/2 ← arity
  - The expression A=B is true if A and B are terms and unify (look identical)

```
?- a = a.
 true
?- a = b.
 false
?- a = X.
 X = a
?- X = Y.
 true
```

Lists & Pattern Matching

- Lists – a convenient way to represent abstract concepts
  - Prolog has a special notation for lists.

[a]
[a,b,c]
[ ]

Empty
List

[bmw, vw, mercedes]
[chicken, turkey, goose]

Lists & Pattern Matching

- Pattern Matching in Lists

```
?- [ a, b ] = [ a, X ].
X = b

?- [ a, b ] = X.
X = [ a, b ]
```

But:

```
?- [ a, b ] = [ X ].
no
```

- The Head-Tail Operator: [H|T]

```
?- [a,b,c] = [X|Y];
X = a
Y = [b,c]

?- [a] = [Q|P];
Q = a
P = [ ]
```

Lists - the First Predicate

The predicate first/2: accept a list in the first argument and return the first element of the list in second argument.

```
first(List,E) :- List = [H|T], E = H.
```

or

```
first([H|_],H).
```

Lists - the Last Predicate

The predicate last/2: accept a list in the first argument and return the last element of the list in second argument.

Recursion: there are always two parts to a recursive definition; the base and the recursive step.

```
last([A],A).
last([ _ |L],E) :- last(L,E).
```

Member Predicate

Write a predicate member/2 that takes a list as its first argument and an element as its second element. This predicate is to return true if the element appears in the list otherwise it returns false.

```
member([E|_],E).
member([_|T],E) :- member(T,E).
```

Exercise

- Write a predicate that compares two lists and returns true if the lists are the same and false otherwise.

Prolog – Arithmetic

- Prolog is a programming language, therefore, arithmetic is implemented as expected.
- The only difference to other programming languages is that assignment is done via the predicate <u>is</u> rather than the equal sign, since the equal sign has been used for the unification operator.

<u>Examples:</u>

```
?- X is 10 + 5;
X = 15

?- X is 10 + 5 * 6 / 3;
X = 20
```

Precedence and associativity of operators are respected.

Prolog – Arithmetic

Example: write a predicate definition for length/2 that takes a list in its first argument and returns the length of the list in its second argument.

```
length([ ], 0).
length(L, N) :- L = [H|T], length(T,NT), N is NT + 1.
```

Prolog – I/O

- write(term)
  - is true if term is a Prolog term, writes term to the terminal.
- read(X)
  - is true if the user types a term followed by a period, X becomes unified to the term.
- nl
  - is always true and writes a newline character on the terminal.

☞ Extra-logical predicates due to the side-effect of writing/reading to/from the terminal.

Prolog – I/O

```
?- write(tom).
tom

?- write([1,2]).
[1, 2]

?- read(X).
|: boo.
X = boo

?- read(Q).
|: [1,2,3].
Q = [1, 2, 3]
```

Prolog I/O Prompt

Exercises

(1) Define a predicate max/3 that takes two numbers as its first two arguments and unifies the last argument with the maximum of the two.

(2) Define a predicate maxlist/2 takes a list of numbers as its first argument and unifies the second argument with the maximum number in the list. The predicate should fail if the list is empty.

(3) Define a predicate ordered/1 that takes a list of numbers as its argument and succeeds if and only if the list is in non-decreasing order.

A Translation Program

Write a program that takes simple English statements and translates them into German. The sentences are given as lists of words.

```
% the dictionary
lookup(logic,logik).
lookup(is,macht).
lookup(fun,spass).


% the translation procedure
translate([],[]).
translate([W|S],G):-
         lookup(W,GW),
         translate(S,GS),
         G=[GW|GS].
```

Interaction Loops

Write a program that prompts a user for a list, then reads the list, reverses the elements of the list and then prints out the reversed list to the terminal. It then returns to prompting the user for a new list, etc.

```
interact:-
    nl,
    write('gimme a list> '),
    read(X),
    reverse(X,Y),
    write('this is the reverse: '),
    write(Y),
    nl,
    !,interact.
```

A Cut

Exercise

- Prolog Exercise 2
  - Prolog excels at natural language processing. This exercise lets you explore this a bit.