Prolog

Reading: Sethi, Chapter 11.

- Overview
- Predicate Calculus
- Substitution and Unification
- Introduction to Prolog
- Prolog Inference Rules
- Programming in Prolog
  - Recursion
  - List Processing
  - Arithmetic
  - Higher-order programming
  - Miscellaneous functions
- Conclusion

Prolog

Programming in **Log**ic

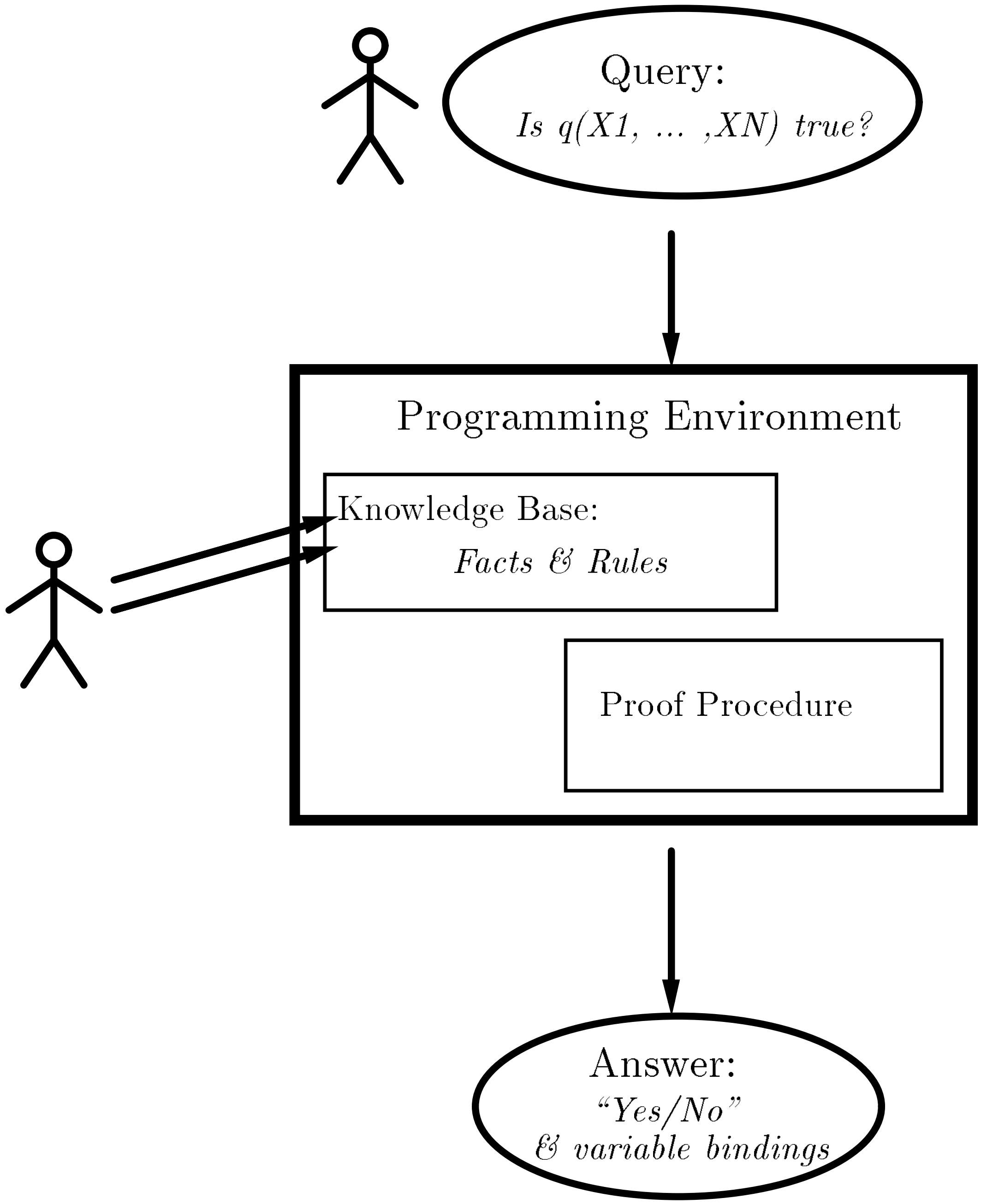
- Idea emerged in early 1970's;
  most work done at Univ. of Edinburgh.

- Based on a subset of first-order logic.
  - Feed it theorems and pose queries, system does the rest.

- main uses:
  - Originally, mainly for natural language processing.
  - Now finding uses in database systems and even rapid prototyping systems of industrial software.

- Popular languages: Prolog, XSB, LDL, Coral, Datalog, SQL.

Logic Programming Framework



Declarative Languages

In its purest form, Logic programming is an example of *declarative programming*.

Popular in database systems and artificial intelligence.

Declarative specifications: Specify what you want, but not how to compute it.

Example. Find `X` and `Y` such that

```
3 X + 2 Y = 1
X - Y = 4
```

A method (program) for solving these is *how* to get values for `X` and `Y`. But all we gave was a *specification*, or *declaration* of what we want. Hence the name.

Examples

• "Retrieve the telephone number of the person whose name is Tom Smith" (easy)

• "Retrieve the telephone number of the person whose address is 13 Black St" (hard)

• "Retrieve the name of the person whose telephone number is 123-3445" (hard)

Each command specifies *what* we want but not *how* to get the answer. A database system would use a different algorithm for each of these cases.

Can also return multiple answers:
• "Retrieve the names of *all* people who live on Oak St."

Algorithm = Logic + Control

• Users specify "logic" — *what* the algorithm does — using logical rules and facts.

• "Control" —*how* the algorithm is to be implemented — is built into Prolog.

i.e., Search procedures are built into Prolog. They apply logical rules in a particular order to answer user questions.

<u>Example</u>. `P if` $\mathtt{Q}_1$ `and` $\mathtt{Q}_2$ `and ... and` $\mathtt{Q}_k$
can be read as

to deduce `P`:
  deduce $\mathtt{Q}_1$
  deduce $\mathtt{Q}_2$
  ...
  deduce $\mathtt{Q}_k$

Users specify what they want using classical first-order logic (predicate calculus).

Classical First-Order Logic

- The simplest kind of logical statement is an <u>*atomic formula*</u>. *e.g.*,

  `man(tom)` (tom is a man)

  `woman(mary)` (mary is a woman)

  `married(tom,mary)`
  (tom and mary are married)

- More complex formulas can be built up using <u>*logical connectives*</u>: $\wedge$, $\vee$, $\sim$, $\forall X$, $\exists X$. *e.g.*,

  `smart(tom)` $\vee$ `dumb(tom)`

  `smart(tom)` $\vee$ `tall(tom)`

  $\sim$ `dumb(tom)`

  $\exists$`X married(tom, X)`
  (tom is married to something)

  $\forall$`X loves(tom, X)`
  (tom loves everything)

  $\exists$`X [married(tom, X)` $\wedge$ `female(X)` $\wedge$ `human(X)]`
  (tom is married to a human female)

Logical Implication

$\texttt{rich(tom)} \vee \sim\texttt{smart(tom)}$

This implies that if tom is smart, then he must be rich. So, we often write this as

$\texttt{rich(tom)} \leftarrow \texttt{smart(tom)}$

In general, $P \leftarrow Q$ and $Q \rightarrow P$ are abbreviations for $P \vee \sim Q$.

For example,

$\forall\texttt{X [(person(X)} \wedge \texttt{smart(X))} \rightarrow \texttt{rich(X)]}$
(every person who is smart is also rich)

$\exists\texttt{X mother(john,X)}$
(john has a mother)

$\exists\texttt{X [mother(john,X)} \wedge$
$\forall\texttt{Y mother(john,Y)} \rightarrow \texttt{Y = X]}$
(john has *exactly* one mother)

Horn Rules

Logic programming is based on formulas called Horn rules. These have the form

$$\forall x_1 ... x_k \; [A \leftarrow B_1 \wedge B_2 ... \wedge B_j]$$

where $k, j \geq 0$.

For example,

```
∀X,Y [A(X) ← B(X,Y) ∧ C(Y)]
∀X [A(X) ← B(X)]
∀X [A(X,d) ← B(X,e)]
A(c,d) ← B(d,e)
∀X A(X)
∀X A(X,d)
A(c,d)
```

Note that atomic formulas are also Horn rules, often called facts.

A *set* of Horn rules is called a *Logic Program*

Logical Inference with Horn Rules

Logic Programming is based on a simple idea:
From rules and facts derive more facts

Example 1. Given the facts `A`, `B`, `C`, `D`, and these rules:

(1) $\mathtt{E} \leftarrow \mathtt{A} \wedge \mathtt{B}$
(2) $\mathtt{F} \leftarrow \mathtt{C} \wedge \mathtt{D}$
(3) $\mathtt{G} \leftarrow \mathtt{E} \wedge \mathtt{F}$

From (1), derive `E`
From (2), derive `F`
From (3), derive `G`

Example 2. Given these facts:

`man(plato)` ("plato is a man")
`man(socrates)` ("socrates is a man")

and this rule:

$\forall \mathtt{X}\ [\mathtt{man(X)} \rightarrow \mathtt{mortal(X)}]$
("all men are mortal")

derive: `mortal(plato)`, `mortal(socrates)`.

Recursive Inference

Example.

Given:

```
∀X [mortal(X) → mortal(son_of(X))]
mortal(plato)
```

Derive:

```
mortal(son_of(plato))
    (using X = plato)
mortal(son_of(son_of(plato)))
    (using X = son_of(plato))
mortal(son_of(son_of(son_of(plato))))
    (using X = son_of(son_of(plato)))
...
```

This kind of inference simulates recursive programs (as we shall see).

Logic Programming

Horn rules correspond to programs, and a form of Horn inference corresponds to execution.

For example, consider the following rule:

$$\forall \texttt{X,Y}\ \texttt{p(X)} \leftarrow \texttt{q(X,Y)} \wedge \texttt{r(X,Y)} \wedge \texttt{s(X,Y)}$$

Later, we shall see that this rule can be interpreted as a program, where

`p` is the program name,

`q,r,s` are subroutine names,

`X` is a parameter of the program, and

`Y` is a local variable.

Non-Horn Formulas

The following formulas are *not* Horn:

$\mathtt{A} \rightarrow \sim \mathtt{B}$

$\mathtt{A} \vee \mathtt{B}$

$\mathtt{A} \vee \mathtt{B} \leftarrow \mathtt{C}$

$\exists\mathtt{X}\ [\mathtt{A(X)} \leftarrow \mathtt{B(X)}]$

$\mathtt{A} \leftarrow (\mathtt{B} \leftarrow \mathtt{C})$

$\forall\mathtt{X}\ [\mathtt{flag(X)} \rightarrow [\mathtt{red(X)} \vee \mathtt{white(X)}]]$
("every flag is red or white")

$\forall\mathtt{X}\ \exists\mathtt{Y}\ [\mathtt{wife(X)} \rightarrow \mathtt{married(X,Y)}]$
("every wife is married to someone")

Non-Horn Inference

Inference with non-Horn formulas is more complex than with Horn rules alone.

Example.

```
A ← B
A ← C
B ∨ C
```
(non-Horn)

We can infer `A`, but must do case analysis:

```
either B or C is true.
  if B then A
  if C then A
Therefore, A is true in all cases.
```

Non-Horn formulas do not correspond to programs, and non-Horn inference does not correspond to execution.

Logical Equivalence

Many non-Horn formulas can be put into Horn form using two methods:

(1) logical equivalence
(2) skolemization

Example 1. Logical Equivalance.

$$
\begin{array}{lcl}
\sim\mathtt{A} \leftarrow \sim\mathtt{B} & \equiv & \sim\mathtt{A} \vee \sim(\sim\mathtt{B}) \\
 & \equiv & \sim\mathtt{A} \vee \mathtt{B} \\
 & \equiv & \mathtt{B} \vee \sim\mathtt{A} \\
\text{(Horn)} & \equiv & \mathtt{B} \leftarrow \mathtt{A}
\end{array}
$$

Logical Laws:

$$
\begin{array}{lcl}
\sim\sim\mathtt{A} & \equiv & \mathtt{A} \\
\sim (\mathtt{A} \vee \mathtt{B}) & \equiv & \sim\mathtt{A} \wedge \sim\mathtt{B} \\
\mathtt{A} \vee (\mathtt{B} \wedge \mathtt{C}) & \equiv & (\mathtt{A} \vee \mathtt{B}) \wedge (\mathtt{A} \vee \mathtt{C}) \\
\mathtt{A} \leftarrow \mathtt{B} & \equiv & \mathtt{A} \vee \sim\mathtt{B}
\end{array}
$$

Example 2. Logical Equivalance.

$$
\begin{array}{lcl}
\mathtt{A} \leftarrow (\mathtt{B} \vee \mathtt{C}) & \equiv & \mathtt{A} \vee \sim(\mathtt{B} \vee \mathtt{C}) \\
 & \equiv & \mathtt{A} \vee (\sim\mathtt{B} \wedge \sim\mathtt{C}) \\
 & \equiv & (\mathtt{A} \vee \sim\mathtt{B}) \wedge (\mathtt{A} \vee \sim\mathtt{C}) \\
\text{(Horn)} & \equiv & (\mathtt{A} \leftarrow \mathtt{B}) \wedge (\mathtt{A} \leftarrow \mathtt{C})
\end{array}
$$

Example 3. Logical Equivalence.

$$
\begin{array}{lcl}
\texttt{A} \leftarrow (\texttt{B} \leftarrow \texttt{C}) & \equiv & \texttt{A} \vee \sim(\texttt{B} \leftarrow \texttt{C}) \\
 & \equiv & \texttt{A} \vee \sim(\texttt{B} \vee \sim\texttt{C}) \\
 & \equiv & \texttt{A} \vee (\sim\texttt{B} \wedge \sim\sim\texttt{C}) \\
 & \equiv & \texttt{A} \vee (\sim\texttt{B} \wedge \texttt{C}) \\
 & \equiv & (\texttt{A} \vee \sim\texttt{B}) \wedge (\texttt{A} \vee \texttt{C}) \\
\text{(non Horn)} & \equiv & (\texttt{A} \leftarrow \texttt{B}) \wedge (\texttt{A} \vee \texttt{C})
\end{array}
$$

In general, rules of the following form *cannot* be converted into Horn form:

$$\forall x[(A_1 \vee ... \vee A_n) \leftarrow (B_1 \wedge ... \wedge B_m)]$$

For example,

$(\texttt{A} \vee \texttt{B}) \leftarrow (\texttt{C} \wedge \texttt{D})$

$(\texttt{A} \vee \texttt{B}) \leftarrow \texttt{C}$

$(\texttt{A} \vee \texttt{B})$

$\forall\texttt{X}\ [\texttt{A(X)} \vee \texttt{B(x)}] \leftarrow [\texttt{C(X)} \wedge \texttt{D(X)}]$

i.e., if it is possible to infer a *non-trivial* disjunction from a set of formulas, then the set is inherently non-Horn.

(A rule like $p \vee q \leftarrow q$ infers a *trivial* disjunction, since the rule is a logical tautology. Such rules can simply be ignored.)

Skolemization

Non-Horn formulas like $\exists x \; A(x)$ can be converted to Horn form.

Example 1.

Replace (1)	$\exists$`X mother(john,X)`	(non-Horn)
with (2)	`mother(john,m)`	(Horn)

Here, `m` is a new constant symbol, called a *skolem constant*, that stands for the (unknown) mother of john.

Note: (1) $\not\equiv$ (2), but they say (almost) the same thing. In particular, (1) can sometimes be replaced by (2) during inference, as we shall see.

Skolemization (Cont'd)

<u>Example 2</u>. A non-Horn formula:

(3) $\forall$X [person(X) $\rightarrow$ $\exists$Y mother(X,Y)]
("every person has a mother")

Let m(x) stand for the (unknown) mother of X. Then, we can replace (3) by a Horn rule:

(4) $\forall$X [person(X) $\rightarrow$ mother(X,m(X))]

m(X) is called a <u>*skolem function*</u>.

It is an artificial name we have created.

e.g., m(mary) denotes the mother of mary.
m(tom) denotes the mother of tom.
m(jfk) denotes the mother of jfk.

So, we only need $\exists$X because we don't have a *name* for X. By creating artifical names (skolem symbols), we can eliminate many $\exists$'s, and convert many formulas to Horn rules, which Prolog can then use.

Skolemization is a technical device for doing inference.

Inference with Skolemization

(1) $\forall$`X [man(X)` $\rightarrow$ `person(X)]`
("every man is a person")

(2) $\forall$`X` $\exists$`Y [person(X)` $\rightarrow$ `mother(X,Y)]`
("every person has a mother"—non Horn)

(3) $\forall$`X,Y [mother(X,Y)` $\rightarrow$ `loves(Y,X)]`
("every mother loves her children")

(4) `man(plato)` ("plato is a man")

<u>Question</u>. $\exists$`Y loves(Y,plato)`
("does someone love plato?")

<u>Step 1</u>. Skolemize (2) to get a Horn rule:
(2') $\forall$`X [person(X)` $\rightarrow$ `mother(X, m(X))]`

<u>Step 2</u>. Use Horn inference:

`person(plato)`	from (1)
`mother(plato,m(plato))`	from (2')
`loves(m(plato),plato)`	from (3)

<u>Thus</u>. $\exists$`Y loves(Y,plato)`
i.e., `Y = m(plato)`. So, answer is YES.

Skolem Dependencies

(1) $\exists$X $\forall$Y p(X,Y)

skolemizes to $\forall$Y p(a,Y),
where a is a skolem constant.

(2) $\forall$Y $\exists$X p(X,Y)

skolemizes to $\forall$Y p(b(Y),Y),
where b is a skolem function.

i.e., in (2), X depends on Y.
But in (1), X is independent of Y.

(3) $\forall$X $\forall$Y $\exists$Z q(X,Y,Z)

skolemizes to $\forall$X $\forall$Y q(X,Y,c(X,Y)),
where c is a skolem function of both X and Y.

i.e., in (3), Z depends on both X and Y.

Skolem Dependencies — Concrete Examples

$\exists$`X` $\forall$`Y loves(X,Y)` (“someone loves everybody”)

$\Rightarrow$ $\forall$`Y loves(p,Y)` (“p loves everybody”)

$\forall$`X` $\exists$`Y mother(X,Y)` (“everyone has a mother”)

$\Rightarrow$ $\forall$`X mother(X,m(X))`

(“m(X) is the mother of X”)

$\forall$`X` $\forall$`Y` $\exists$`Z owns(X,Y)` $\rightarrow$ `document(Z,X,Y)`

(“if X owns Y, then there is a document, Z, saying that X owns Y”)

$\Rightarrow$ $\forall$`X` $\forall$`Y owns(X,Y)` $\rightarrow$ `document(d(X,Y),X,Y)`

(“d(X,Y) is a document saying that X owns Y”)