Predicate Logic

Proposition logic deals with propositions (self-contained things that can be true or false). Individuals are not entities.

Predicate logic constructs truth values out of predicates that apply to individuals.

A predicate with all argument individuals specified is like a proposition, in that it has a true or false value.

Quantifiers

In addition to truth function operators of proposition logic, predicate logic introduces quantifiers for expressing variation over individuals:

\[ (\forall x) p(x) : \text{for all } x, \ p(x) \]

universal quantifier

\[ (\exists x) p(x) : \text{for some } x, \ p(x) \]

existential quantifier

Interpretation of Formulas with Quantifiers

An interpretation of the symbols in a formula assumes a set of individuals over which we are quantifying, called the domain.

The domain may be finite or infinite.

An interpretation also assumes a specific predicate for each predicate symbol.

Fun with Specific Interpretations

| Triangle(x) means x is a triangle |
| Square(x) means x is a square |
| Circle(x) means x is a circle |
| Red(x) means x is red, etc. |

above(x, y) means x is above y |
left(x, y) means x is to the left of y

Which are true in the interpretation shown?

red(3)
square(2)
above(2, 1)
left(3, 1)
left(1, 1) \land (red(2) \lor yellow(3))

Specific Interpretations that quantify over individuals

\[ (\forall x) (\square(x) \lor \circle(x)) \]
\[ (\forall x, y) ((\square(x) \land \circle(y)) \rightarrow \above(x, y)) \]
\[ (\exists x, y) ((\square(y) \land \circle(x)) \land \left(x, y)) \]
\[ (\exists x) (\left(x, x)) \]
\[ (\forall x) \square(x) \rightarrow (\red(x) \lor \green(x)) \]
Give Formulas that Characterize the Interpretation
(without referring to specific individuals)

1
3
4
2

Give Formulas that Characterize the Interpretation

1
3
4
6
5

Give Formulas that Characterize the Interpretation

1

Give Formulas that Characterize the Interpretation

1
2
4
5
6

Formulas Valid for the Interpretation
Natural Numbers

- \((\forall x) (x = 0) \lor (\exists y) (x = S(y))\)
- \((\forall x) -S(x) = 0\)
- \((\forall x, y) S(x) = S(y) \rightarrow x = y\)
- \((0) \land (\forall x) (p(x) \rightarrow p(S(x)))) \rightarrow (\forall x) p(x)\)

Here

- \(S\) represents the successor function \(S(x) = x+1\)
- 0 is an "individual" symbol
- \(p\) is any predicate
- These form a variant on the Peano axioms (Giuseppe Peano, 1889).

Universally Valid Formulas
are true regardless of interpretation

- We always assume the domain is non-empty.
- \((\forall x) p(x) \rightarrow (\exists x) p(x)\)
- \(((\forall x) p(x) \lor (\forall x) q(x)) \rightarrow (\forall x) (q(x) \lor p(x))\)
- \((\exists x) (p(x) \land q(x)) \rightarrow ((\exists x) p(x)) \land ((\exists x) q(x))\)
- \((\forall x) (p(x) \land q(x)) \rightarrow ((\forall x) p(x)) \land ((\forall x) q(x))\)
- \((\exists x) p(x) \leftrightarrow (\forall x) \neg p(x)\)
- \((\forall x)(\forall y) p(x, y) \rightarrow (\forall y)(\exists x) p(x, y)\)
- not valid: \((\forall x)(\exists y) p(x, y) \rightarrow (\exists y)(\forall x) p(x, y)\)
Uses of Predicate Logic

- Querying databases
- Program specification
- Program verification
- Advanced properties of logic circuits (time dependence, etc.)

Relational Database Example

<table>
<thead>
<tr>
<th>Name</th>
<th>Dorm</th>
<th>Name</th>
<th>Dept</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nama</td>
<td>South</td>
<td>Nama</td>
<td>CS</td>
<td>60</td>
</tr>
<tr>
<td>Alice</td>
<td>West</td>
<td>Naima</td>
<td>CS</td>
<td>5</td>
</tr>
<tr>
<td>Toshiko</td>
<td>East</td>
<td>Toshiko</td>
<td>CS</td>
<td>50</td>
</tr>
<tr>
<td>Roy</td>
<td>North</td>
<td>Roy</td>
<td>CS</td>
<td>60</td>
</tr>
<tr>
<td>Albert</td>
<td>South</td>
<td>Albert</td>
<td>CS</td>
<td>55</td>
</tr>
</tbody>
</table>

Three relations:
- `lives`: name x dorm
- `takes`: name x dept x number
- `tutors`: name x dept x number

Relational Database Example

Sample Queries:
- Who lives in South dorm?
  - `x: lives(x, South)`
- Who lives in East dorm and takes CS 5?
  - `x: lives(x, East) ^ takes(x, CS, 5)`
- Who takes a CS course?
  - `x: (\exists y) takes(x, CS, y)`

Predominant Database Languages

- SQL (sometimes pronounced "sequel")
  - Structured Query Language
  - The standard query language used in most commercial relational database systems (Oracle, Informix, Sybase, etc.)
  - Invented by Don Chamberlin, HMC '66
- Prolog (sometimes pronounced "prolog")
  - Programming in Logic
  - A complete programming language
  - Used in AI and rapid prototyping
Prolog Tutorial

- Relations can be expressed in two ways:
  - Enumeration
  - Rules
  - Combinations of both are possible
- Highly case-sensitive
  - Predicates and constants are in lower-case, unless quoted with single quotes '…'
  - Variables begin with upper-case
  - _ is a variable (does not match others)
  - ‘_’ is not used for quoting; it means something else.

Prolog Rule Syntax

- A clause such as
  `lives(john,east).`
  is called a unit clause or fact; it refers to one piece of information.
- Non-unit clauses typically are in the form of reverse implications:
  `Consequent :- Antecedent.`
  which stand for
  `Antecedent implies Consequent.`

Prolog Tutorial

- Typical developmental execution (as opposed to complete application) scenario:
  - Knowledge Base (= Database+Rules) is loaded ("consulted").
  - Queries are posed based on loaded database.

Prolog on Turing

(text in red is typed by user)

```
turing ~:1> prolog
Quintus Prolog Release 3.2 (Sun 4, SunOS 5.3)
Copyright (C) 1994, Quintus Corporation. All rights reserved.
Licensed to Harvey Mudd College, CS Dept.
| ?- consult(tutors). % tutors.pl contains the database
  % compiling file /home/keller/tutors.pl
| ?- lives(john, X). % Where does john live?
  X = east
| ?- lives(X, east). % Who lives in east?
  X = john ;
  X = toshiko ;
  no
```

Prolog Tutorial

Enumeration of the lives relation in Prolog:

```
lives(john,east).
lives(naima, south).
lives(alice, west).
lives(toshiko, east).
lives(roy, north).
lives(albert, south).
```

Prolog Tutorial

Enumeration of the tutors relation in Prolog:

```
tutors(john, cs, 5).
tutors(naima, cs, 5).
tutors(roy, math, 3).
tutors(alice, math, 55).
tutors(albert, math, 4).
```

Prolog Tutorial

Typical developmental execution (as opposed to complete application)
scenario:

- Knowledge Base (= Database+Rules) is loaded ("consulted").

Prolog Rule Syntax

Consequent :- Antecedent.

- Can be read as any of the following:
  - Antecedent implies Consequent
  - Consequent is implied by Antecedent
  - Consequent if Antecedent
  - Consequent provided Antecedent
- These are called the logical interpretation of a clause.
### Non-Unit Clause Examples

- **livesInEast(X) :- lives(X, east).**
  - *variable* → individual

- **canTutor(X, Y) :-
  tutors(X, Dept, Number),
  takes(Y, Dept, Number).**
  - Comma in this context is *and*.

Variables occurring in the consequent are implicitly universally quantified.

### Existential Variables

- **knows(X, Y) :-
  lives(X, Z),
  lives(Y, Z).**

A variable occurring on the right but not on the left is implicitly existentially quantified.

Means "(For all X, Y), if there is some Z such that lives(X, Z) and lives(Y, Z),
then knows(X, Y)."

### Multiple Clauses for One Predicate

- **knows(X, Y) :-
  lives(X, Z),
  lives(Y, Z).**

- **knows(X, Y) :-
  takes(X, Dept, Number),
  tutors(Y, Dept, Number).**

Expressing two different ways for X to know Y:

### Logic of Passing an Exam

- **There are two ways for a person X to pass an exam:**
  - X is adequately prepared, or
  - the exam is extremely easy

- **pass_exam(X) :- prepared_for_exam(X).**
- **pass_exam(X) :- easy_exam, person(X).**

This is like a proposition variable.

It is equivalent to a predicate with no arguments, and parens are not used.

### Preparing for an Exam (1)

- **There are three ways for X to be prepared:**
  - X knows it all

- **prepare_for_exam(X) :-
  knows_it_all(X).**

### Preparing for an Exam (2)

- **X read the book, attended classes (without sleeping), and worked the problems:**

- **prepare_for_exam(X) :-
  read_book(X),
  attended_classes(X),
  \(+ slept_during_lectures(X),
  worked_problems(X).**

Note: \(+ is Prolog’s version of not.
Preparing for an Exam (3)

- X was tutored by Y, who was prepared for the exam:
  
  prepared_for_exam (X) :-
  tutored_by (X, Y),
  prepared_for_exam (Y).

Simulating a Logic Circuit

Goal-Oriented (Procedural)

Interpretation of Prolog

- The execution of Prolog is actually a form of depth-first search.
- A Prolog query is composed of a list of goals (the individual predicate expressions).
- Prolog tries to solve these goals by finding individuals that satisfy the predicates, as determined by the knowledge base.
- During the solving process, a goal is replaced by other goals, according to the rules, until there are no unsolved goals left.

Who passes the exam?

person (mary).
person (john).
person (tom).
person (sally).
person (fred).

read_book (fred).
read_book (mary).
work_prob (fred).
work_prob (mary).

attended_lectures (fred).
attended_lectures (mary).
slept_during_lectures (fred).
knows_it_all (tom).
tutored_by (john, mary).
tutored_by (sally, john).

Previous Example Prolog KB

Goal Succession:

Depth-First Execution in Prolog

canTutor(alice, Y).

variable, since starts with upper-case
**Backtracking in Depth-First Search**

```
canTutor(alice, Y).
tutors(alice, Dept, Number), takes(Y, Dept, Number).
take(naia, math, 55).
take(naia, math, 55).
Y = naia
result
binding
Dept = math
Number = 55

tutors(alice, math, 55).
takes(Y, math, 55).
```

**Deeper Backtracking**

```
canTutor(X, Y).
canTutor(X, Y) :-
tutors(X, Dept, Number), takes(Y, Dept, Number).
tutors(alice, math, 55).
takes(Y, math, 55).
X = alice
Dept = math
Number = 55

etc.
```

**Summary of Backtracking**

- Given a goal, Prolog tries rules in order of occurrence (“top-to-bottom”), using the first rule, the consequent of which matches the goal.
- If the rule has sub-goals, the sub-goals are satisfied in order of occurrence (“left-to-right”), resulting in bindings at each stage.
- If a goal sub-goal fails completely, Prolog retries to satisfy it using the next available option (e.g. the next rule).
Rule and Sub-Goal Ordering

Suppose the goal is `knows(john, Y, R).`

This rule is tried first.

This rule is tried after the first rule is exhausted.

In effect, we have **disjunction** (or) among rules, and **conjunction** (and) within rules. Remember that Prolog execution is **depth-first search**.

And-Or Trees

- In AI, problem-solving trees are typically "And-Or" trees.
- This applies to Prolog's goals.

Lists in Prolog

- The rex list notation was derived from Prolog's list notation.
- A list can contain logical variables, which are already bound, or may get bound later.
- The process of binding is known as **unification** (which means "make the same").

"Logical Variables" in Prolog

- A variable in Prolog is like an object that can have one of two states:
  - unbound
  - bound, to some Prolog term, e.g., an individual

- Once the variable is bound, it only gets re-bound in **backtracking**, which results in removing the former binding first.

Unification

- Unification causes two logical variables to denote the same thing.
- The symbol for unification in Prolog is `=.`
- Examples:
  - `X = a` Variable unified with constant
  - `X = [a, b, c]` Variable unified with a list
  - `[X, b, c] = [a | Y]` `X = a, Y = [b, c]`
  - `[X, b] = [a, c]` NOT UNIFIABLE

Unification (1)

- Unification takes place implicitly when a rule is used for a goal:
  ```prolog
  knows(john, U, R).
  knows(X, Y, living) :-
  lives(X, Z),
  lives(Y, Z).
  ```
  - `X = john`
  - `Y = U`
  - `R = living`
  - The rule is usable iff the goal and the rule consequent variables unify.

Unification (2)
Special handling of _ in Prolog

- The variable _ is special.
- It is called a "throw-away" variable.
- _ unifies with anything, but different occurrences of _ are not identified, unlike other variables.

Other variables beginning with _

- A variable that occurs only once in a clause is called a "singleton variable".
- Often singleton variables are the result of a typing error, and certain compilers will warn about them.
- To prevent the warning, when this is the intention, use a variable that begins with _, such as _Name rather than Name.

Example: Towers of Hanoi

- Move only one disk at a time. Never place a larger disk on a smaller one.

Solving Towers of Hanoi

- Some approaches:
  - Pre-programmed solution
    - Recursive solution is easy in most languages
  - Let Prolog find solution using depth-first search
    - Trickier, but shows off Prolog's capabilities
    - May not find shortest solution
  - Program breadth-first search in Prolog
    - Still trickier

Pre-Programmed Towers of Hanoi (1)

- To move N disks from stack From to stack To:

Pre-Programmed Towers of Hanoi (2)

- To move N disks from stack From to stack To:
  - Move N-1 disks from stack From to stack Other (the stack other than From and To)

A key point throughout is that the N-1 disk moves can be done without violating the constraint that a larger disk not be put atop a smaller one.
Pre-Programmed Towers of Hanoi (3)
- To move N disks from stack \textbf{From} to stack \textbf{To}:
  - Move N-1 disks from stack \textbf{From} to stack \textbf{Other} (the stack other than \textbf{From} and \textbf{To})
  - Move 1 disk from stack \textbf{From} to stack \textbf{To}

Pre-Programmed Towers of Hanoi (4)
- To move N disks from stack \textbf{From} to stack \textbf{To}:
  - Move N-1 disks from stack \textbf{From} to stack \textbf{Other} (the stack other than \textbf{From} and \textbf{To})
  - Move 1 disk from stack \textbf{From} to stack \textbf{To}
  - Move N-1 disks from stack \textbf{Other} to stack \textbf{To}

The \textit{is} operator in Prolog
- Unlike a functional language, expressions that look like arithmetic are not automatically evaluated.
- Example: \(X - 1\) is kept as its syntax tree \(-(X, 1)\).
- The \textit{is} operator is used to force evaluation.
- Example: \(Y\) is \(X-1\) means: unify \(Y\) with the arithmetic value of \(X-1\)

Depth-First Towers of Hanoi (1)
Does not require a human to solve the puzzle first

First characterize the possible moves.

This is a move from stack 1 to stack 2:

\[
\begin{array}{c|c|c|c}
\text{from/to} & \text{stack 1 before} & \text{stack 2 after} & \text{condition} \\
\hline
1, 2 & (F1, R1, S2, S3) & (R1, F1, S2, S3) & \text{ok(F1, S2)} \\
\end{array}
\]

provided that it is ok to move disk F1 onto stack S2

Depth-First Towers of Hanoi (2)
All the possible moves in six rules:

\[
\begin{array}{c}
\text{move(1, 2), } (\{F1 : R1, S2, S3\}, \{R1, F1 : S2, S3\}) \leftarrow \text{ok(F1, S2)} . \\
\text{move(1, 3), } (\{F1 : R1, S2, S3\}, \{R1, S2, F1 : S3\}) \leftarrow \text{ok(F1, S3)} . \\
\text{move(2, 1), } (\{S1, F2 : R2, S3\}, \{F2 : R1, S2, S3\}) \leftarrow \text{ok(F2, S1)} . \\
\text{move(2, 3), } (\{S1, F2 : R2, S3\}, \{S1, F2, R2 : S3\}) \leftarrow \text{ok(F2, S3)} . \\
\text{move(3, 1), } (\{S1, S2, F1 : R3\}, \{F1 : S1, R2, S3\}) \leftarrow \text{ok(F1, S3)} . \\
\text{move(3, 2), } (\{S1, S2, F1 : R3\}, \{S1, F3, S2 : R3\}) \leftarrow \text{ok(F3, S2)} . \\
\end{array}
\]
When is it ok to move a disk onto a stack?
Assume the disks are represented by numbers 1, 2, 3, ... with smaller numbers representing smaller disks.

\[
\text{ok}(\_, [\_]) \quad \text{empty target stack}
\]
\[
\text{ok}(A, [B | \_]) \leftarrow \text{smaller}(A, B).
\]
\[
\text{smaller}(A, B) : - A < B.
\]

**Depth-First Towers of Hanoi (4)**

towers([S1, S2, S3], Moves) will mean that Moves is a valid move sequence that results in S1 and S2 being empty (so all disks are on S3).
towers([S1, S2, S3], Seen, Moves) means the same, except that Seen will be a list of all previous states (to prevent infinite looping).
towers(InitialState, Moves) :- towers(InitialState, [\], Moves).
towers([\], [\], \_ | Moves) :- \% final state, no more moves

\[
towers(\text{Before}, \text{Seen}, [\text{Move} | \text{Moves}]) \leftarrow\neg \text{member}(\text{Before}, \text{Seen}),
\]
\[
\text{move}(\text{Move}, \text{Before}, \text{After}),
\]
\[
towers(\text{After}, \text{After} | \text{Seen}, \text{Moves}). \quad \text{only consider if Before not already seen}
\]
\[
towers(\text{After}, \text{After} | \text{Seen}, \text{Moves}). \quad \text{recursive}
\]

**Exercise**

Reverse the pegs by moving peg “forward” or jumping forward over a peg of either color.
Work out a depth-first solution in Prolog.
(You don't have to check for cycles, because there can't be any.)

**Bi-Directional Execution (1)**
Consider:

\[
\text{member}(X, [X | \_]).
\]
\[
\text{member}(X, [\_ | Y]) :- \text{member}(X, Y).
\]

This predicate can be viewed as a member test.
It can also be viewed as a member generator.

**Bi-Directional Execution (2)**

\[
test \leftarrow \neg \text{member}(3, [1, 2, 3, 4, 5]).
\]
\[
\text{yes}
\]
\[
\neg \text{member}(6, [1, 2, 3, 4, 5]).
\]
\[
\text{no}
\]
\[
generate \leftarrow \text{member}(X, [1, 2, 3, 4, 5]).
\]
\[
X = 1;
\]
\[
X = 2;
\]
\[
X = 3;
\]
\[
X = 4;
\]
\[
X = 5;
\]
\[
\text{no}
\]
Generating with append

### relational

\[
\text{append } ([1, 2, 3], [4, 5], Z).
\]

\[
Z = [1, 2, 3, 4, 5] ;
\]

\[
\text{no}
\]

### functional

\[
\text{append } ([1, 2, 3], [4, 5], Z).
\]

\[
Z = [1, 2, 3, 4, 5] ;
\]

\[
\text{no}
\]

Generator/Test Example: Map Coloring

A map

Corresponding graph

**Map Coloring (2)**

**Map Coloring (3)**

Prolog Clause

\[
\text{map}([A, B, C, D, E, F, G]) :-
\]

Graph

1. next(A, B),
2. next(A, C),
3. next(A, D),
4. next(A, E),
5. next(B, E),
6. next(C, D),
7. next(C, E),
8. next(C, F),
9. next(D, E),
10. next(D, F),
11. next(E, B),
12. next(E, F),
13. next(E, G),
14. next(F, G).

**Map Coloring (4): Color Constraints**

\[
\text{next}(X, Y) :- \text{color}(X), \text{color}(Y), X \neq Y.
\]

\[
\text{color(red), color(blue), ...}
\]

\[
\text{colors to be used}
\]

\[
\text{means individuals are not equal}
\]

These and the preceding clause are the entire program.

Constraint Solving with Generate & Test: “Zebra” problem (1 of 2)

There are five consecutive houses, each of a different color and inhabited by men of different nationalities. They each own a different pet, have a different favorite drink and drive a different car.

1. The Englishman lives in the red house.
2. The Spaniard owns the dog.
3. Coffee is drunk in the green house.
4. The Ukrainian drinks tea.
5. The green house is immediately to the right of the ivory house.
6. The Porsche driver owns snails.
7. The Masserati is driven by the man who lives in the yellow house.
8. Milk is drunk in the middle house.
9. The Norwegian lives in the first house on the left.
10. The man who drives a Saab lives in the house next to the man with the fox.
11. The Masserati is driven by the man in the house next to the house where the horse is kept.
13. The Norwegian lives next to the blue house.
14. The Norwegian lives in the first house on the left.

The problem is: Who owns the Zebra? Who drinks water?

---

Prolog Solution to “Zebra”

```prolog
left_right(L, R, [L, R, _, _, _]).
left_right(L, R, [_, L, R, _, _]).
left_right(L, R, [_, _, L, R, _]).
left_right(L, R, [_, _, _, L, R]).
next_to(X, Y, L) :- left_right(X, Y, L).
next_to(X, Y, L) :- left_right(Y, X, L).

zebra(S) :-
  S = [[_, norwegian, _, _, _], [blue, _, _, _, _]],
  next_to([_, norwegian, _, _, _], [blue, _, _, _, _], S),
  member([green, coffee, _], S),
  left_right([ivory, _, _, _, _], [green, coffee, _], S),
  member([red, englishman, tea, _], S),
  member([yellow, masserati, _, _], S),
  member([red, englishman, tea, _], S),
  member([red, englishman, tea, _], S),
  member([spaniard, _, tea, _], S),
  next_to([_, _, masserati, _, _], [_, englishman, tea, _], S),
  next_to([_, _, saab, orange_juice, _], S).
```

---

Built-in Arithmetic Not Reversible

- Consider a clause
  \( p(X, Y) :- Y \text{ is } X + 1. \)
- A possible use of this clause is:
  \[ ?- p(5, Z). \]
  \[ Z = 6 \]
- This clause cannot be used in reverse because the is predicate is not reversible:
  \[ ?- p(X, 6). \]
  \[ \text{will not give } X = 5; \text{ it will fail.} \]

---

Prolog’s Origins

(see [http://mox.cs.ku.edu/~acarra/prolog.html](http://mox.cs.ku.edu/~acarra/prolog.html))

- Prolog was invented at the University of Montreal around 1970 by Alain Colmerauer, who since has been professor at the École Supérieure d’Ingénieurs de Luminy in Marseille, France.
- The original work was a grammar-based language for natural language translation.

---

Prolog Perspective

- A complete programming language
- Not a complete logic language
  - Restricted to “Horn Clauses”
  - Restricted form of negation
  - Quantifiers not completely general
  - Builtin arithmetic not reversible
- More powerful logic systems exist, e.g.
  - Otter (see CS 80 or 151)
Contemporary Extensions of Prolog

- Constraint logic programming
- Inductive logic programming
- Lambda-prolog
- Goedel
- Parallel prologs
- Prolog++
- ... (The list is quite long.)