Logic Programming

Greg Smith

KCDCC, 2002

In this lecture

- Briefly review logic
- Describe declarative programming using a subset of logic: Prolog
- Examples of places where Prolog could be used
  - Expert System or Knowledge System
  - Inference layer on a database
  - Reasoning engine of an applet: Hanoi's Tower

Imperative Programming

- Some procedural statements:
  - To get to the library, go to the stairs ...
  - To tie your shoelaces, first take the left lace ...
- Programming in this way is called imperative
- Is the usual computing paradigm
- Such as C, Basic, Fortran ...
- Object Oriented languages Java, Javascript, C/C++, ... mostly imperative

Declarative Programming

- Some non-procedural statements:
  - The library is on the 4th floor
  - Workers are supervised by the manager of the department in which they work
- These are not statements of how to do things
- They are declarative statements that can be applied to multiple differing tasks
- We can use logic to program such statements

Propostions

- "Socrates is a man" is a proposition
- It is either `true` or `false`

- Let `p,q,...` denote propositions
- Then use connectives:
  ~p denotes "it is not the case that p"

p	~p
true	false
false	true

Propostions

p∧q denotes "both p and q"

p	q	p∧q
false	false	false
false	true	false
true	false	false
true	true	true

p∨q denotes "either p or q"

p	q	p∨q
false	false	false
false	true	true
true	false	true
true	true	true

Propostions

p⇒q denotes "if p then q"

p	q	p⇒q
false	false	true
false	true	true
true	false	false
true	true	true

p ⇔ q denotes "p if and only if q"

p	q	p ⇔ q
false	false	true
false	true	false
true	false	false
true	true	true

Example

"If the world is round then the ancient Greeks were clever"

- Denote "the world is round" by p
- Denote "the ancient Greeks were clever" by q
- So the above rule is p⇒q
- If we are told that p is true
- Then we can deduce that q must be true
- Thus we can reason with propositions

Nonsense in - nonsense out

- A reasoner just applies the rules you give it
- If you put nonsense in you will get nonsense out
- For example, we can encode 2 rules:
  `"wet grass" ⇒ "it rained"`
  `"we break this bottle" ⇒ "wet grass"`
- From which we can legally deduce
  `"we break this bottle" ⇒ "it rained"`
- The problem is with the 1st rule - it should have been:
  `"wet grass" ⇐ "it rained"`
- The reason does not understand the rules - it uses them blindly!

Predicates

- How do we denote the following?
  - Socrates is wise∴Someone is wise

- Everyone is happy ∴ Plato is happy

- We need to be able to say things like
  - There exists at least 1 person x that x is wise
  - For every person x, x is happy
- To do this we quantify our propositions by saying what objects they apply to

Predicates

- A predicate P(a) is true or false
- where
  - P is a property
  - a is an object
- E.g. wise(Socrates) is true if Socrates is wise
- Reasoning with predicates is Predicate Calculus or 1st Order Logic
- and Prolog is programming with (certain kinds of) predicates

Predicates

- Constants

  a,b,c
- Variables

  x,y,z
- Predicates

  P(a,b),Q(x,y,c)

Predicates

- Clauses

  Finite, possibly empty, set of predicates and/or negated predicates
- Sentences

  P(x,y)

  P(x,y)∧Q(a)

  P(x,y)⇒Q(a)

  ~P(x,y)

Predicates

- Quantifiers

  To express "for all x, P(x) is true" write:

  ∀ x • P(x)

  To express "there is at least 1 x where P(x) is true" write:

  x • P(x)

Example

- "Socrates is a man"

  Denoted by man(socrates)

- "All men are mortal"

  Denoted by ∀ x • man(x)⇒mortal(x)

- If we are told that man(socrates) is true and that ∀ x • man(x)⇒mortal(x) is true
- Then we can deduce that mortal(socrates) must be true

- Thus we can reason with predicates

Horn Clauses

- In order that computation is tractable we restrict predicate logic to sentences of this form:

  $Q \wedge R \wedge ... \wedge S \Rightarrow P$

- Which for convenience can be rewritten as:

  $P \Leftarrow Q, R, ..., S.$

Horn Clauses

- Also,
  $Q \vee R \Rightarrow P$

- can be rewritten as:

  $P \Leftarrow Q. \quad P \Leftarrow R.$

- Called Horn clauses - the basis of Prolog!

Prolog

- Can easily prove program correctness
- Not goal dependent & non-deterministic
- High level - simple but powerful
- No distinction between program and data
- No distinction between input and output
- No assignment statement
- Negation as failure: assume a statement is false if an attempt to prove it fails

1st Prolog Example

- Consider previous example:
  "All men are mortal"
  "Socrates is a man"

- In prolog this looks like

```
mortal(A) :- man(A).
man(Socrates).
```

- We run the program and it waits for a goal ...

1st Prolog Example

```
1 ?- man(_).
Yes
2 ?- man(X).
X = Socrates
Yes
3 ?- mortal(X).
X = Socrates
Yes
4 ?- man(X),not(mortal(X)).
No
```

Syntax of Prolog

- A Prolog program is composed of:
  - constants → atoms and numbers
  - variables
  - predicates
  - fact clauses
  - query clauses
  - rule clauses
- Knowledge is supplied to Prolog as facts and rules
- Goals are expressed as queries
- Prolog searches for knowledge that can satisfy goals

Constants: atoms & numbers

- An atom is a string of characters that
  - Starts with a...z
    or is in single quotes
  - Includes the following characters:
    ```
    a..z
    A..Z
    0..9
    +-*/<>:.&_~
    ```

Constants: Examples

- Examples
  ```
  x25
  bob
  'Australia'
  ::=
  ```
- There are other predefined atoms, e.g.
  ```
  :-
  ```
- Range of numbers depends on implementation
- Examples
  ```
  1000
  -0.035
  ```

Prolog variables

- A variable is a string of characters as with atoms except starting with A...Z or _
- For example,
  ```
  ShoppingList
  Result
  _x23
  ```
- A single _ is the anonymous variable
- The anonymous variable matches any variable, atom or number

Prolog variables

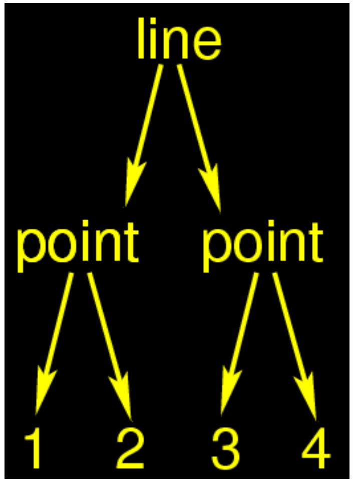
- Prolog satisfies goals by finding bindings to goal variables that will make goal queries true

```
3 ?- mortal(X).
X = Socrates
Yes
4 ?- man(X),not(mortal(X)).
```

`No`

Prolog predicates

- A predicate assigns `true` or `false` to an object
- Are objects with multiple components, possibly including other predicates
- For example, `line(point(1,2),point(3,4))`



Prolog predicates

- May contain variables to be bound later
- For example,
  `date(Day,june,2000)`

- In Prolog, a term is a predicate that does not contain variables

- Finding bindings to goal variables that will satisfy goal queries is by matching terms

Matching of constants

- IF `S` and `T` are constants THEN they only match if they are the same object
- For example,
  ```
  bob=bob.
  fred=3.14.
  ```
- The 1st is true, the 2nd is false.

Matching of variables

- IF `S` is a variable and `T` is anything THEN they match with `T` instantiated to `S`
- For example,
  ```
  X=3.14, Y=X.
  ```
- This can be made `true` by binding
  `X` to `3.14` and
  `Y` to the now bound value of `X`

Matching of predicates

- IF `S` and `T` are predicates THEN they only match if
  - `S` and `T` have same principal functor
  - All of the corresponding components match

  The resulting instantiation is determined by matching components

- For example,
  `point(1,1)=X, point(2,3)=point(2,Z).`
- This can be made `true` by binding
  `X` to `point(1,1)` and
  `Z` to `3`

Matching of rules

- Prolog clauses consist of a head and body
- For example,
  `mother(X,Y) :- parent(X,Y), female(X).`
- The body is a list of goals separated by commas (ANDs)
- The head can be matched (satisfied) by matching each clause in the body

Matching of rules

- Facts are clauses with an empty body
- For example,
  `parent(bob,mary).`
- Queries only have a body
  `parent(bob,X).`
- Rules have a head and a non-empty body
- For example,
  `mother(X,Y) :- parent(X,Y), female(X).`

Matching of rules

- A question posed to Prolog is a list of goals
- For example,
  `mother(ann,Y),mother(Y,Z).`
- A goal clause `G` can be satisfied IFF
  - There is a clause `C` with all variables bound such that
  - the head of `C` is identical to `G` and
  - all the goals in the body of `C` are `true`

Example: Family tree

- Let `parent(X,Y)` mean
  `X` is the parent of `Y`
- Let `grandparent(X,Z)` mean
  `X` is the grandparent of `Z`
- Let the program have a rule clause:
  `grandparent(X,Z) :- parent(X,Y), parent(Y,Z).`

Example: Family tree

- Assert a number of facts:

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

Example: Family tree

- And we provide goals to the Prolog engine:

```
1 ?- grandparent(tom,X).
X = ann
Yes
2 ?- findall(X, grandparent(tom,X), G).
G = [ann, pat]
Yes
3 ?- trace(parent),trace(grandparent).
parent/2: call redo exit fail
grandparent/2: call redo exit fail
Yes
```

Example: Family tree

```
4 ?- grandparent(tom,X).
T Call: ( 7) grandparent(tom, _G200)
T Call: ( 8) parent(tom, _L128)
T Exit: ( 8) parent(tom, bob)
T Call: ( 8) parent(bob, _G200)
T Exit: ( 8) parent(bob, ann)
T Exit: ( 7) grandparent(tom, ann)
X = ann
Yes
```

Side Effects

- The `trace` predicate caused Prolog to behave differently before than after
- This is called a side effect
- We will see examples later such as `!` and I/O
- Sometimes side effects are essential
- However they
  - represent a control on reasoning
  - break declarative nature of encoded knowledge
  - should minimise side effects
  - For example, separate reasoning system from user interface

Recursion

- An important concept needed for Prolog is recursion
- Consider this definition of `factorial`
  `factorial` of `0` is `1`
  `factorial` of `n` is `n*(n-1)*(n-2) ... *1` if `n > 0`
- We can calculate this using a technique known as divide and conquer
  - Given a large problem,
  - divide large problem into smaller ones,
  - continue dividing until solutions to small problems are simple

Recursion

- For `factorial` note that
  - `3!` simplifies to `3*2!`
  - `2!` simplifies to `2*1!`
  - `1!` simplifies to `1*0!`

    - `0!` is `1`
- Thus we need two rules

```
factorial(0,1).
factorial(N,Y) :- N > 0,
 N1 is N - 1,
 factorial(N1,Y1),
 Y is N * Y1.
```

Recursion

- 1st rule `factorial(0,1).` is the terminating condition
- 2nd rule is the general condition
- Using the same clause in the head and body is called recursion
- Recursion is common in Prolog because we try to encode declarative knowledge
- Instead of writing
  `for (int i=0; i<N; i++) val=val*i;`
- ... we express knowledge that relates successive iterations and let Prolog apply them

Example: Monkey & Bananas

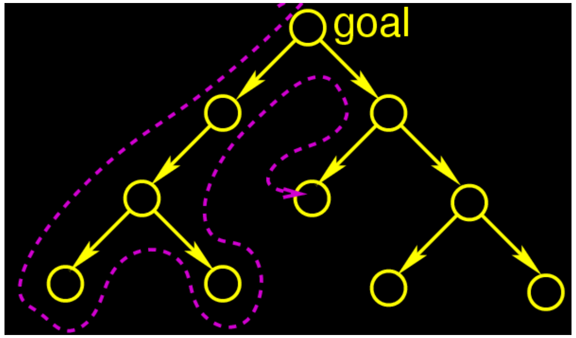
- A monkey is at the door of a room
- A banana hangs from middle of ceiling
- Monkey cannot reach banana without standing on a box
- There is a box at the window
- Define predicate describing state of world: 4 components
    - Horiz position of monkey ∈{`atdoor`,`atwindow`,`middle`}
    - Vertical position of monkey ∈{`onbox`,`onfloor`}
    - Position of box ∈{`atdoor`,`atwindow`,`middle`}
    - Whether monkey has banana ∈{`has`,`hasnot`}

Example: Monkey & Bananas

- Monkey can take 4 actions:
  move ∈ {`grasp`,`climb`,`push`,`walk`}
- Goal is `state(_,_,_,has)`
- The program
- Execution trace

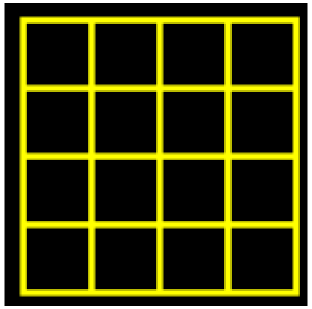
Backtracking

- The Monkey & Bananas problem exhibited backtracking
- Remember: Prolog tries to solve goals by repeatedly matching
- Think of the execution trace as a tree
    - Root node is the goal given to the Prolog engine
    - Leaf nodes contain facts with no variables
    - Each branch uses matching to bind one more variable



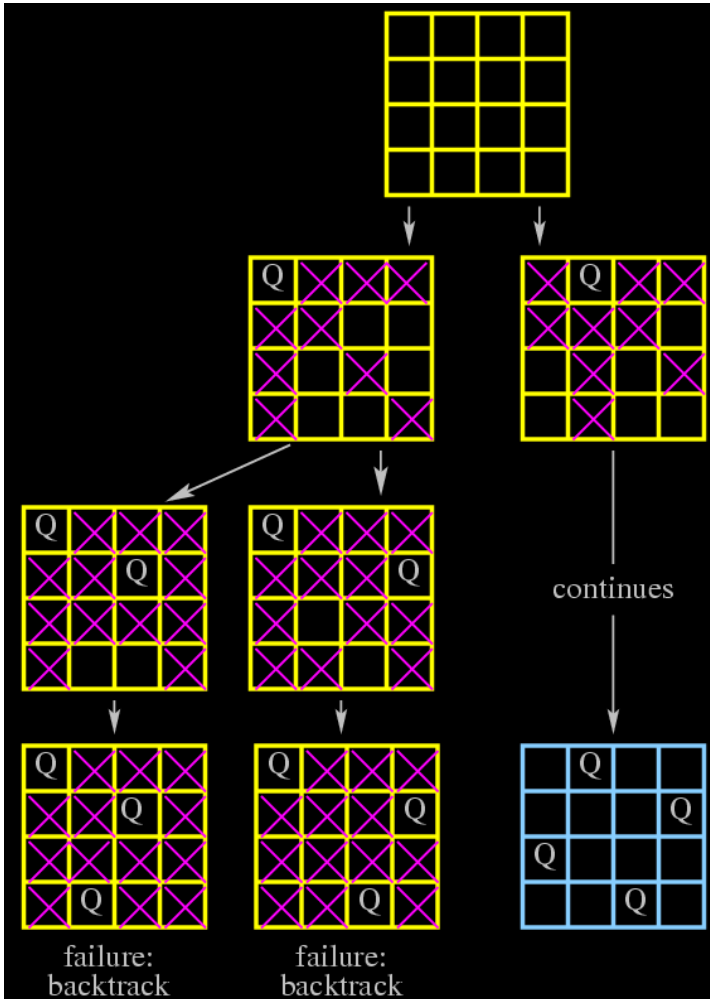
Backtracking

- For example, consider the 4 queens problem:



- Must place 4 queens on 4x4 chessboard
- No queen is to be in same row, column or diagonal as another
- Goal given to prolog is `solution(Q1,Q2,Q3,Q4)` where `Q1,Q2,Q3,Q4` are board positions

Backtracking



Backtracking

- When it finds binding of all variables to values in knowledge base it stops with success

- If reaches a dead end it undoes bindings and tries different ones
- This is called backtracking

- Question: how do we write "Bob likes all animals except snakes"?
- There is a special term `fail` that is always false

Backtracking

- However this will not work:
```
likes(bob,X) :- snake(X),fail.
likes(bob,X) :- animal(X).
```
- The reason is that when Prolog sees the `fail` it backtracks

- The correct way to write it is:
```
likes(bob,X) :- snake(X),!,fail.
likes(bob,X) :- animal(X).
```

- The `!` is called a cut

Backtracking

- A cut it prevents backtracking past it
- Allows for mutually exclusive rules, e.g. if condition then conclusion Q else conclusion R
- Can improve efficiency by reducing search by Prolog
- `fail`, `!`, `true`, and `repeat` all cause side effects
- ⇒use sparingly: destroys declarative nature of knowledge base because you impose imperative knowledge

Lists

- `[bob,rugby,fred,triathlon]` is a list in Prolog
- `bob` is the head of the list
- `[rugby,fred,triathlon]` is the tail of the list
- So an alternative representation is `[bob|Tail]` where `Tail=[rugby,fred,triathlon]`
- That is, `[bob|[rugby,fred,triathlon]]`

- 1st representation useful for printing etc
- 2nd representation useful recursive rules etc

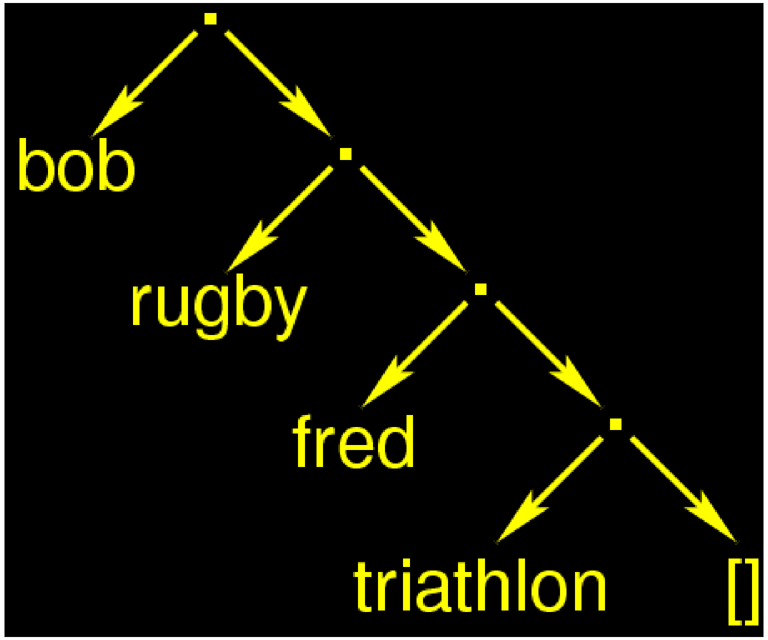
Lists

- E.g. to determine whether `X` is a member of list `L`:
```
member(X,[X|Tail]).
member(X,[Head|Tail]) :- member(X,Tail).
```

- `member(fred,[bob,rugby,fred,triathlon])` is `true`
  - `member` is instantiated recursively until `Head` matches `fred`
  - Then `member` unwinds
- `member(ann,[bob,rugby,fred,triathlon])` is `false`
  - `member` is instantiated recursively until `Tail` matches `[]`
  - Then `member` unwinds

Lists

- Prolog actually stores it like this:



Lists

- Here `.` is an inbuilt predicate
- So we could also represent the list as
  `.(bob,.(rugby, .(fred, .(triathlon, []))))`

- By doing a similar thing we could define more complex data structures E.g. trees

Lists

- Prolog compilers/interpreters come with lots of inbuild list predicates
- For example
  - `append(List1,List2,List3)` is `true` when `List3` matches with the concatenation of `List1` and `List2`
  - `delete(List1,Elem,List2)` is `true` when `List2` matches with `List1` but with elements that match `Elem` removed

Lists

- `nth0(Index,List,Elem)` is `true` when the `Index`-th element of `List` matches `Elem`
- `last(Elem,List)` is `true` when `Elem` matches the last element of `List`
- `reverse(List1,List2)` is `true` when `List2` matches `List1` after the order of the elements in `List1` has been reversed

Lists

- Remember: these are declarative not imperative
- For example,

```
1 ?- nth0(1,[bob,rugby,fred,triathlon],E).
E = rugby
Yes
2 ?- nth0(I,[bob,rugby,fred,triathlon],fred).
I = 2
Yes
3 ?- nth0(1,L,triathlon).
```

```
L = [_G294, triathlon|_G298]
Yes
```

Maths

- Operators specify relations between objects
- Three kinds:
  - Logical relations compare the identity of objects
  - Arithmetic relations compare numerical and algebraic expressions
  - Arithmetic functions manipulate numerical and algebraic quantities
- Operators are predicates but use a different notation
  - Prefix, such as `not p`
  - Infix, such as `X + 1`
  - Postfix, such as `10%`

Maths

- Example logical relations
  - `not` is logical not
  - `==` is strict equality, e.g.
    `male(X) == male(X)` is true
    `male(X) == male(Y)` is false
  - `=` is lax equality, e.g.
    `male(X) == male(X)` is true
    `male(X) == male(Y)` is true

Maths

- Example arithmetic relations
  - `=:=` is arithmetic equality, e.g.
    `1 =:= X` is true iff `X` is already bound to numeric value 1
  - `=\=` is arithmetic inequality
  - `>=` is arithmetic greater-than-or-equal-to, e.g.
    `X >= 6` is true iff `X` is already bound to numeric value no smaller than 6

Maths

- Example arithmetic functions
  - `3 + 2`
  - `X - Y`
  - `Z \ X` (this is modulo, or remainder)
- Note that `X = Y` does not assign a value to `X`
- `X = Y` only matches existing values with lax equality
- To assign the current value of `Y` to `X` use the `is` operator
- For example, `Y is X + 1` computes `X + 1`, and assigns this value to `Y` providing that `X + 1` is true
- You can define your own operators in Prolog!

Input & Output

- Prolog compilers/interpreters come with lots of inbuilt predicates for I/O
- For example:
  - `read(X)` gets one term from the current input stream and matches that term against `X`
  - `write(X)` prints `X` to the current output stream
  - `see(Filename)` succeeds if `Filename` exists, and

side effect is to switch input stream to that file
- `tell(Filename)` succeeds if `Filename` exists, and side effect is to switch output stream to that file

Input & Output

- `consult(Program)` succeeds if `Program` is a file containing Prolog, and side effect is to load those clauses

- These predicates allow for loading/saving knowledge base, getting user responses, and displaying results, etc
- However they all have side effects
- Thus I/O rules from other reasoning

Example App: Expert System

- Simple expert system from book by Bratko
- Three parts
  - Knowledge base is set of Prolog fact clauses, E.g.
    ```
    rule1 ::
    if Animal has hair or Animal gives milk
    then Animal isa mammal.
    ```
  - Inference engine: prolog clauses
  - User interface: prolog clauses
- Prolog can satisfy goals
- But expert system can say why
- When we run it ...

Expert System

```
Question, please
|: lenny isa tiger.
Is it true: lenny has hair? yes.
Is it true: lenny eats meat? no.
Is it true: lenny has pointed teeth? yes.
Is it true: lenny has claws? why.
To investigate, by rule3, lenny isa carnivore
To investigate, by rule5, lenny isa tiger
```

Expert System

```
Is it true: lenny has claws? yes.
Is it true: lenny has forward pointing eyes?
yes.
Is it true: lenny has tawny color? yes.
Is it true: lenny has black stripes? yes.
lenny isa tiger is true
Would you like to see how?
|: yes.
```

Expert System

```
lenny isa tiger
 was derived by rule5 from
  lenny isa carnivore
   was derived by rule3 from
    lenny isa mammal
     was derived by rule1 from
      lenny has hair
       was told
    and
```

```
lenny has pointed teeth
 was told
```

Expert System

```
and
lenny has claws
 was told
and
lenny has forward pointing eyes
 was told
and
lenny has tawny color
 was told
```

Expert System

```
and
lenny has black stripes
 was told
```

- This can do a lot & it can tell you how it did it
- But ...
  - Knowledge is trapped within prolog
  - The user interface is not very good

Inference on a Database

- If company records, design requirements, etc are in RDBMS
- Then wouldn't it be nice to retrive facts directly from RDBMS?

- Consider JIProlog: a Prolog engine written in Java
- Provides Java classes like `JDBCClausesDatabase`
- Provides prolog predicates like
```
extern(myPred/n,
"JIP.xdb.JDBCClausesDatabase",
"driverName,URL,view").
```

Inference on a Database

- Where
  - `driverName` is the JDBC driver class name
  - `URL` is the JDBC URL used to open a connection to the database
  - `view` is a table or a valid SQL SELECT query
- If successful, a set of facts `myPred` each with `n` components will be available for the Prolog engine to match against

User interface: Hanoi's Tower

- Game written in JavaLog, another Prolog engine written in Java
- Graphics and user I/O written in Java
- Reasoning done in Prolog
- Define rules using clause `hanoi(Height, PositionA,`

`PositionB, PositionC, ListOfMoves)`
where `Height` is nof disks to move on `PositionA`
- Solve recursively (again)

Hanoi's Tower

- Creating the general rule:
  - It is possible to move N disks in order from A to B with C as a temporary position, `hanoi(N, A, B, C, Ms2)`
  - If it is possible to move N-1 disks off disk N to temporary C, `hanoi(N1, A, C, B, Ms1)`
  - And it is possible to then move those N-1 disks back onto B, `hanoi(N1, C, B, A, Ms2)`
  - And we can safely move disk N from A to B, `append(Ms1, [[A,B]|Ms2], Moves)`

Hanoi's Tower

- The terminating condition rule says stop recursing at height of 1
- So the program is:

```
hanoi(1, A,B,C,[[A,B]]) :- !.
 hanoi(N, A,B,C,Moves) :-
 N1 is N - 1,
 hanoi(N1, A, C, B, Ms1),
 hanoi(N1, C, B, A, Ms2),
 append(Ms1, [[A,B]|Ms2], Moves),
 !.
```

Demo of Hanoi's Tower

Here in the lecture a Hanoi's Tower applet runs, showing a Java GUI and Prolog reasoning.

Demo of Hanoi's Tower

- The Hanoi's Tower demo shows that
  - Can use Java for user interface with Prolog reasoning
  - Can include Prolog reasoning in your web pages

References

Bratko, I., 1989: Prolog Programming for Artificial Intelligence, Addison-Wesley, Avon UK.

SWI-Prolog, http://www.swi-prolog.org

JIProlog, http://ugosweb.com/jiprolog

JavaLog, http://www.exa.unicen.edu.ar