Prolog Introduction

Running Prolog code

- We are going to use gprolog to check your homework code:
  http://www.gprolog.org/#download
  - Installation instructions (optional)

- There exists a different implementation of the Prolog language -- SWI Prolog
  http://www.swi-prolog.org/Download.html
  - But we do not support it for the homework

Running Prolog code

gprolog install guides
http://www.cslab.pepperdine.edu/warford/cosc450/cosc-450-Setup-for-Prolog.pdf

To load file:
consult('filename').
For example, to load sec5.pl type:
consult('sec5').
Or alternatively **['sec5'].**

To Exit: :- **halt.**
Or alternatively: Ctrl-D
(more tips here)

To compile in interpreter mode:

```
| ?- [user].

{compiling user for byte code...}

even(0).

even(s(s(X))):-

        even(X).
```

(here the user presses Ctl-D to end the input)

```
{user compiled, 3 lines read - 350 bytes written, 1180
ms}
```

What is a Prolog program?

- Technically, you don’t create a Prolog program
- You create a Prolog database
- In the database are two types of clauses
  - Facts
  - Rules

I don’t understand. What is Prolog?

- General-purpose "what" language

- Basic idea:
    - Declare a set of ***facts***
    - Declare a set of ***rules***
        - Ways to learn new facts from old facts
    - A program is a ***query*** that asks what things are true

Prolog Code

```
person(alice).
person(bob).

person(X).
```

- lowercase variable names are ***atoms***
  - An uninterpreted constant
  - `alice` is a constant
- `person` is a ***predicate***
  - An uninterpreted function that returns a boolean
- `X (capital letter x)` is a ***variable***

Prolog Program

- Program describes relations, defined by clauses.
- 2 types of clauses:
  - Predicates

```
person(alice).
person(bob).
person(X).
```

  - Rules
    - Describe a relationship between facts:
      - `left_hand_side :- right_hand_side`

Rules

- Allows new facts to be inferred from existing facts.

- *conclusion* :- *hypothesis*.
  head :- body
  - The head is if the body is true.

Facts and inference rules can use *variables*. A variable is a name, in upper-case letters, which stands for some possible member of a tuple in a relation.

Example

Example

```
father(homer, bart).
father(homer, lisa).
father(homer, maggie).
father(grandpa, homer).
father(grandpa, herb).

mother(marge, bart).
mother(marge, lisa).
mother(marge, maggie).
mother(grandma, homer).
```

- `paternal_grandfather(X, Y) :- father(X, Z), father(Z, Y).`
  - X is Y's paternal grandfather if there exists Z such that X is Z's father, and Z is Y's father.
  - X, Y, and Z are all variables, and the comma means "and"
- `paternal_grandmother(X, Y) :- mother(X, Z), father(Z, Y).`

More Examples

```
likes(john, susie).                    /* John likes Susie */
likes(X, susie).                       /* Everyone likes Susie */
likes(john, Y).                        /* John likes everybody */
likes(john, Y), likes(Y, john).        /* John likes everybody and everybody likes John */
likes(john, susie); likes(john,mary). /* John likes Susie or John likes Mary */
not(likes(john,pizza)).                /* John does not like pizza */
likes(john,susie) :- likes(john,mary)./* John likes Susie if John likes Mary.
```

Examples of valid rules:

```
friends(X,Y) :- likes(X,Y),likes(Y,X).            /* X and Y are friends if they like each other */
hates(X,Y) :- not(likes(X,Y)).                    /* X hates Y if X does not like Y. */
enemies(X,Y) :- not(likes(X,Y)),not(likes(Y,X)).  /* X and Y are enemies if they don't like each other */
```

Examples of invalid rules:

```
left_of(X,Y) :- right_of(Y,X)
likes(X,Y),likes(Y,X) :- friends(X,Y).
not(likes(X,Y)) :- hates(X,Y).
```

More examples

```
                             James I
                                |
                                |
               +----------------+-----------------+
               |                                  |
            Charles I                          Elizabeth
               |                                  |
               |                                  |
    +----------+------------+                     |
    |          |            |                     |
Catherine   Charles II   James II               Sophia
                                                  |
                                                  |
                                                  |
                                               George I
```

Prolog Visualizar

http://www.cdglabs.org/prolog/

links to the tutorial: Slideshare.

More Prolog Resources

- Prolog Tutorial Part I
- Prolog Tutorial Part II
- Some simple Prolog examples
- On-line Prolog Tutorial
- Some Prolog Books:
    - Leon Sterling, Ehud Shapiro: *The Art of Prolog*, MIT Press, 1986 (2nd ed. 1994);
    - W.F. Clocksin and C.S. Mellish: *Programming in Prolog*, Springer-Verlag, New York, 1981. (4th ed. 1994)
    - Several other decent books exist.

Overview of arithmetic

- In first glance, everything is pretty much the same
- However, there are some important points to consider, more ahead

Arithmetic examples	Prolog Notation
$x < y$	`X < Y.`
$x \leq y$	`X =< Y.`
$x = y$	`X =:= Y.`
$x \neq y$	`X =\= Y.`
$x \geq y$	`X >= Y`
$x > y$	`X > Y`

Arithmetic - ‘is’

Posing the following queries yields:

```
?- 8 is 6+2.
yes
?- 12 is 6*2.
yes
?- -2 is 6-8.
yes
?- 3 is 6/2.
yes
?- 1 is mod(7,2).
yes
```

And we can also work out answers

```
?- X is 6+2.

X = 8

?- X is 6*2.

X = 12

?- R is mod(7,2).

R = 1
```

Arithmetic - Define a Predicate

- Define add_3_and_double

  add_3_and_double(X,Y) :- Y is (X+3)*2.

- So:
  - ?-add_3_and_double(1,X).
    X=8
  - ?-add_3_and_double(5,X).
    X=16
  - ?-add_3_and_double(X,12).
    Error… does not work. Why?

Arithmetic - ‘is’ usage

- Arithmetic expressions must be on the right hand side of is
  ?- X is 6+2.
  X = 8

- But instead we asked
  6+2 is X.

- We got an instantiation error or something similar.
- We are free to use variables on the right hand side of is, when we actually carry out evaluation, the variable must already have been assigned to an integer
- If it is unassigned, we get the error message
- Why? We are asking Prolog to evaluate X in the reverse direction
    - The expression should be on the right side of ‘is’, otherwise we get an error

Arithmetic operators

- Relational Operators
- The infix operators:
    - =:=
    - =\=
    - >
    - >=
    - <
    - =<
- are special type known as relational operators

Arithmetic equality

- Arithmetic Expression Equality

E1=:=E2 succeeds if the arithmetic expression E1 and E2 evaluate to the same value

```
?- 6+4=:=6*3-8.
yes
?- sqrt(36)+4=:=9.
no
```

Arithmetic inequality

- Arithmetic Expression Inequality

E1=\=E2 succeeds if the arithmetic expression E1 and E2 evaluate to the same value

```
?- 10=\=8+3.
yes
?- 8+3=\=11.
no
```

Equality Operators

- Terms identical ==
  The goal term1==term2 succeeds only if term1 is identical to term2. Any variables used in the terms may or may not be already bound, but no variables are bound as result of evaluating the goal
  - likes(X,prolog)==likes(X,prolog) → yes
  - likes(X,prolog)==likes(Y,prolog) → no

- Terms not identical \==
  term1\==term2 succeeds if term1==term2 fails

Equality- unification

- Terms identical width unification =
  = is similar to == with one difference. The goal term1=term2 succeeds only if term1 and term2 unify, i.e. there is some way of binding variables to values which would make the terms identical. If the goal succeeds, such binding takes place
  - pred1(X)=pred1(10)
    X=10 → variable X is bound to 10, which makes the two terms identical
  - likes (X,prolog)=likes(john,Y).
    X=john,Y=prolog → binding X to the atom john and Y to the atom prolog makes the two terms identical.

Equality- unification

- likes(X,prolog)=likes(Y,prolog).
  X=Y=_ → binding X and Y makes the terms identical

- likes(X,prolog)=likes(Y,ada).
  No → no unification can make the atoms prolog and ada identical

Equality- non-unification

- term1 \= term2
  The goal term1 \= term2 succeeds if term1=term2 fails, i.e. the two terms cannot be unified. Otherwise it fails.
  - 6+4\=3+7
    Yes

Equality- summary

- = The unification predicate. Succeeds if it can unify its arguments, fails otherwise.
- \= The negation of the unification predicate. Succeeds if = fails, and vice-versa.
- == The identity predicate. Succeeds if its arguments are identical, fails otherwise.
- \== The negation of the identity predicate. Succeeds if == fails, and vice-versa.
- =:= The arithmetic equality predicate. Succeeds if its arguments evaluate to the same integer.
- =\= The arithmetic inequality predicate. Succeeds if its arguments evaluate to different integers.

Hands on

Guiding the Search Using Cut !

- The goal "!", pronounced cut, always succeeds immediately but just once (cannot backtrack over it).

- It has an important side-effect: once it is satisfied, it **disallows** (just for the current call to predicate containing the cut):
  - backtracking before the cut in that clause
  - Using next rules of this predicate

- So, below, before reaching cut, there might be backtracking on b1 and b2 and even trying other rules for p if b1&b2 cannot be satisfied.

```
p:- b1,b2,!,a1,a2,a3. %however, after reaching !, no backtracking on b1&b2
p:- r1,…,rn.          %also this rule won't be searched
p:- morerules.        %this one too!
```

- See the following link for more details and examples:

http://cs.union.edu/~striegnk/learn-prolog-now/html/node88.html#sec.l10.cut

List Count

- Find the number of elements of a list.
- ? my_length([1,2,3], X).
- X = 3.

List Count

- my_length([],0).
- my_length([_|L],N) :- my_length(L,N1), N is N1 + 1.]

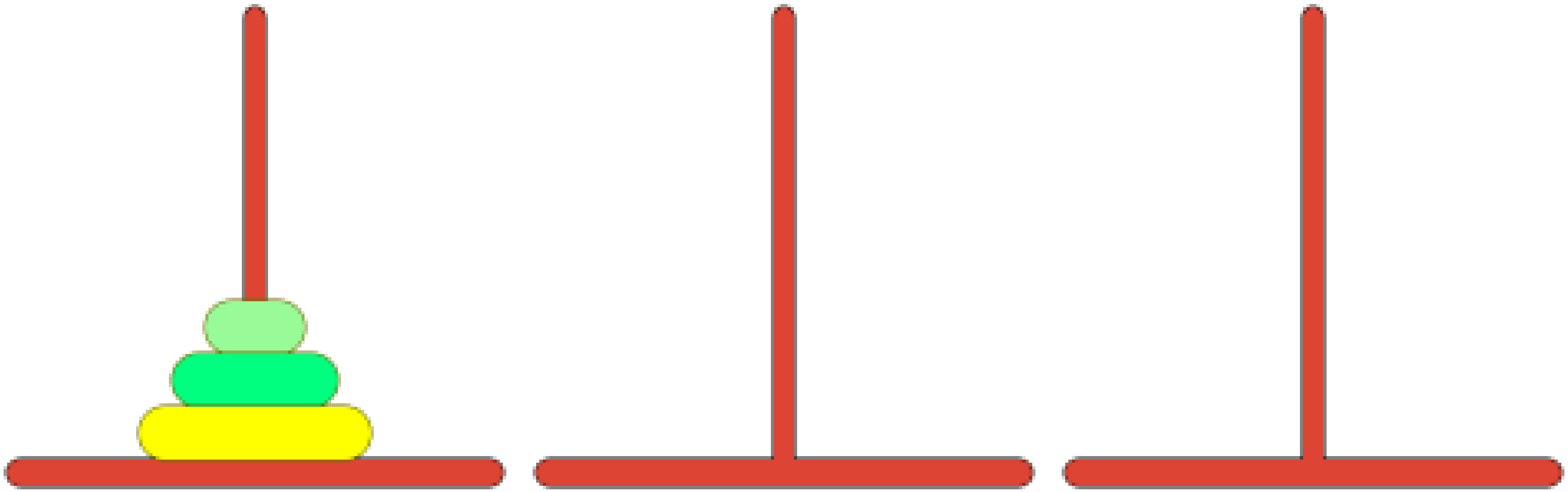
List Compression

- Eliminate consecutive duplicates of list elements.
- If a list contains repeated elements they should be replaced with a single copy of the element. The order of the elements should not be changed.
-
- Example:
- ?- compress([a,a,a,a,b,c,c,a,a,d,e,e,e,e],X).
- X = [a,b,c,a,d,e]

List Compression

- compress([],[]).
- compress([X],[X]).
- compress([X,X|Xs],Zs) :- compress([X|Xs],Zs).
- compress([X,Y|Ys],[X|Zs]) :- X \= Y, compress([Y|Ys],Zs).

Tower of Hanoi



Tower of Hanoi

- move(1,X,Y,_) :-
- write('Move top disk from '),
- write(X),
- write(' to '),
- write(Y),
- nl.
- move(N,X,Y,Z) :-
- N>1,
- M is N-1,
- move(M,X,Z,Y),
- move(1,X,Y,_),
- move(M,Z,Y,X).

Tower of Hanoi

- ?- move(3,left,right,center).
- Move top disk from left to right
- Move top disk from left to center
- Move top disk from right to center
- Move top disk from left to right
- Move top disk from center to left
- Move top disk from center to right
- Move top disk from left to right

- yes