PROLOG

Books

- Text Book: PROLOG programming for AI By Ivan Bratko, P.Ed. Publication, 3rd Edition

Optional books for further reading:

- The Art of Prolog, Leon Sterling and Ehud Shapiro, 2nd Edition, MIT Press, 1994.
- W. F. Clocksin, C. S. Mellish, Programming in Prolog:Using the Iso Standard, 5th edition, Springer-Verlag,2003.
- Prolog : A Relational language and its Applications, John Malpas, Prentice-Hall, Inc. 1987.
- “Programming in Prolog”, by Clocksin & Mellish, Narosa Publishing House.

Prolog Interpreter/Compiler

- Most common free Prolog implementation is
  - You can download freely EasyProlog compiler by Strawberry also.
  - SWI Prolog is Very nice,
  - though faster ones are for sale (e.g., SICSTUS Prolog).
  - **SICStus Prolog** is a commercial s/w used in a wide range of domains from medicine research to data mining of financial data.
- Prolog has grown from a simple stand-alone interpreter into a full compiler-based technology with
  - links to external databases, GUIs, and other languages such as C,C# and Java, and in so doing has effectively made the transition from **academic-theorem prover** to a **complete general-purpose programming language**.
  - E.g. **SICStus Prolog (commercial sw)**

SWI Prolog Program

- Write the program in simple text editor & save it as .pl
- On the terminal, type gprolog.
- To load a file, consult '[filename]'

Using Prolog

1. First, write your program (away from computer!) or using SWI Prolog editor.
2. Then, type it into a file, with a **.pl** extension.
   - Any text editor will do, but **Emacs** is recommended.
3. Then, type:

   sicstus
4. You will be presented with the Prolog prompt

   |?-
5. Then, `consult' your file (omitting the .pl):

   |?- consult(yourfilename). or
   |?- [yourfilename]. or **['folder/filename'].**
6. The entire content of your file is then stored in the memory of the Prolog interpreter.
   - You can see what is consulted by typing **| ?- listing.**
7. Then you can ask questions of your database.

Using Prolog (2)

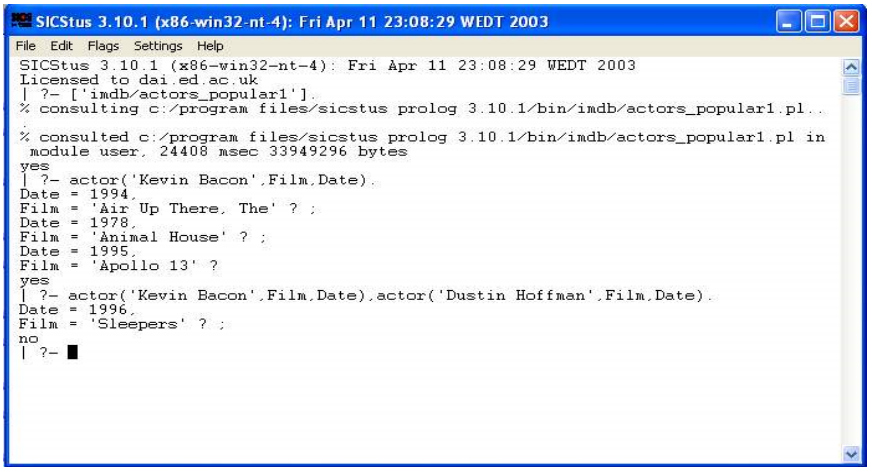
- If you edit your program file (e.g. to correct something), be sure to consult it again afterwards!
- To exit from Prolog, type

  |?- halt.
  or press
  Control/D
- The Prolog comment characters:
  - Single line comments: **%**

    % This is a comment
    This not a comment, but an error
  - Multiple line comments: **/***

    /* This is a multi-line comment
    which must be closed with a */

Prolog Demo



The original declarative programming language

- It was developed in the 1970's by
  - **Robert Kowalski** and **Maarten Van Emden** (Edinburgh)
  - **Alain Colmerauer (Marseille)**
- Prolog is declarative language.
- Types of programming languages
  - (imperative, functional, object-oriented, declarative)
- Alain Colmeraeur & Philippe Roussel, 1971-1973
  - With help from theorem proving folks such as Robert Kowalski
  - Original project: Type in French statements & questions
    - Computer needed NLP and deductive reasoning
  - Efficiency by David Warren, 1977 (compiler, virtual machine)
  - Colmerauer & Roussel wrote 20 years later: "Prolog is so simple that one has the sense that sooner or later someone had to discover it … that period of our lives remains one of the happiest in our memories.
  - "We have had the pleasure of recalling it for this paper over almonds accompanied by a dry martini."

An example Prolog Program

Shows path with mode of conveyance from city C1 to city C2

```
byCar(auckland, hamilton).
byCar(hamilton,raglan).
byCar(valmont, saarbruecken).
byCar(valmont, metz).
byTrain(metz,frankfurt).
byTrain(saarbruecken, frankfurt).
byTrain(metz, paris).
byTrain(saarbruecken, paris).
byPlane(frankfurt, bangkok).
byPlane(frankfurt, singapore).
byPlane(paris, losAngeles).
byPlane(bangkok, auckland).
byPlane(losAngeles, auckland).
go(C1,C2) :- travel(C1,C2,L), show_path(L).
travel(C1,C2,L) :- direct_path(C1,C2,L).
travel(C1,C2,L) :-direct_path(C1,C3,L1), travel(C3,C2,L2),append(L1,L2,L).
direct_path(C1,C2,[C1,C2,' by car']):- byCar(C1,C2).
direct_path(C1,C2,[C1,C2,' by train']):- byTrain(C1,C2).
direct_path(C1,C2,[C1,C2,' by plane']):- byPlane(write(C1), write(' to_'),
    write(C2), write(Me(C1,C2).
show_path([C1,C2,M|T]) :- nl, show_path(T).
```

```
/* output
4 ?- consult('G:/AI2009/lecture programs/best_fs.pl').
% G:/AI2009/lecture programs/best_fs.pl compiled 0.02 sec, 24 bytes true.

5 ?- go(saarbruecken, auckland).
saarbruecken to_frankfurt by train
frankfurt to_bangkok by plane
bangkok to_auckland by plane
saarbruecken to_paris by train
paris to_losAngeles by plane
losAngeles to_auckland by plane
false.

6 ?- go(auckland,craglan).
false.

7 ?- go(auckland,raglan).
auckland to_hamilton by car
hamilton to_raglan by car
false.                                  */
```

Imperative vs Declarative

- Programming in imperative programming (e.g. Java, Pascal, C++), you tell the computer HOW to solve the problem, i.e. do this, then do that.
- In declarative programming (e.g. Prolog),you declare WHAT the problem is, and leave it to the compiler/interpreter to use built-in reasoning to solve the problem.

Programming Language Comparison

Imperative programming languages	Logic programming languages
• 'procedural' -> they describe *how* a sequence of instructions compute the result to a certain problem.	• specify the problem in a declarative style (facts about objects, relations between objects), describe *what* is the objective and let the system prove it
• we concentrate on how to formulate a solution in terms of the primitive operations available	• we concentrate on problem
• what you see is what is being done	• there are *underlying mechanisms* that help the program reach its goal

Logic Programming

- Systems should be constructed by expressing knowledge in a formal language and that problems should be solved by running inference processes on that knowledge.
- Robert Kowalski's equation
  - **Problem Solving =Problem Description + Logical Deductions**
  - How do we build in the ability to do the deductions?
  - Ideally we would like to be able to tell the computer **'what'** we want it to do and not **'how'** we want it to do it.

Prolog

- Prolog is based on first order logic.
- **When you write a Prolog program, you are writing down your knowledge of the problem – you are modelling the problem.**
- Prolog is **declarative** (as opposed to **imperative**):
  - You specify *what* the problem is rather than *how* to solve it.
  - A Prolog program consists of a set of clauses
  - Each clause is either a fact or a rule

Applications of Prolog

- Prolog is very useful in some areas (**AI, natural language processing, Solving puzzles and Games, theorem proving**, **expert systems**), but less useful in others (**numerical algorithms/computations**).
- Prolog is no longer confined to research laboratories, but is now considered to be a powerful tool for the development of commercial applications
  - e.g. some banks and insurance companies use it to evaluate loan applications, make profitability calculations etc.
  - Similar applications in finance, defence, telecommunications, law, medicine, agriculture, engineering, manufacturing, and education

Prolog

- Prolog was the first attempt to design a purely declarative programming language.
- It is short for **'Programmation en Logique'.**
- It was efficiently implemented by DavidWarren.
- It is used primarily as a rapid – prototyping language and for symbol manipulation tasks such as writing compilers and parsing natural language.
- **Many expert systems have been written in prolog for legal, medical, financial, and other domains.**

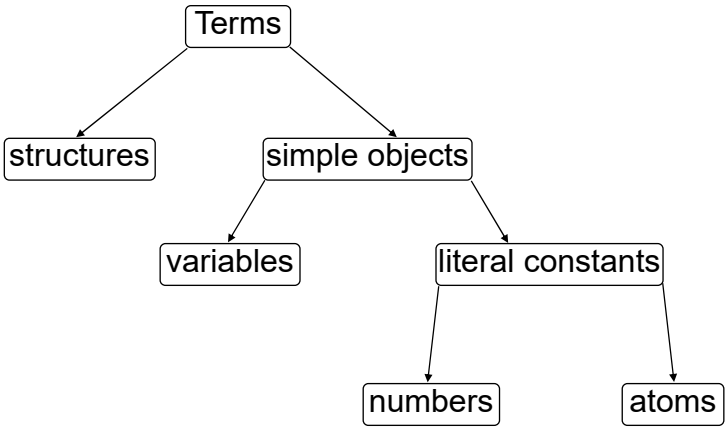
What is Prolog used for?

- Good at
  - Grammars and Language processing,
  - Knowledge representation and reasoning,
  - Unification,
  - Pattern matching,
  - Planning and Search.
    - i.e. Prolog is good at Symbolic AI.
- Poor at:
  - Repetitive number crunching,
  - Representing complex data structures,
  - Input/Output (interfaces).

Logic Programming

- A program in logic is a definition (declaration) of the world - the entities and the relations between them.
- Logic programs establishing a theorem (goal) and asks the system to prove it.
- Satisfying the goal:
  - **yes**, prove the goal using the information from the knowledge base
  - **no**:
    - cannot prove the truth of the goal using the information from the knowledge base
    - the goal is false according to available information

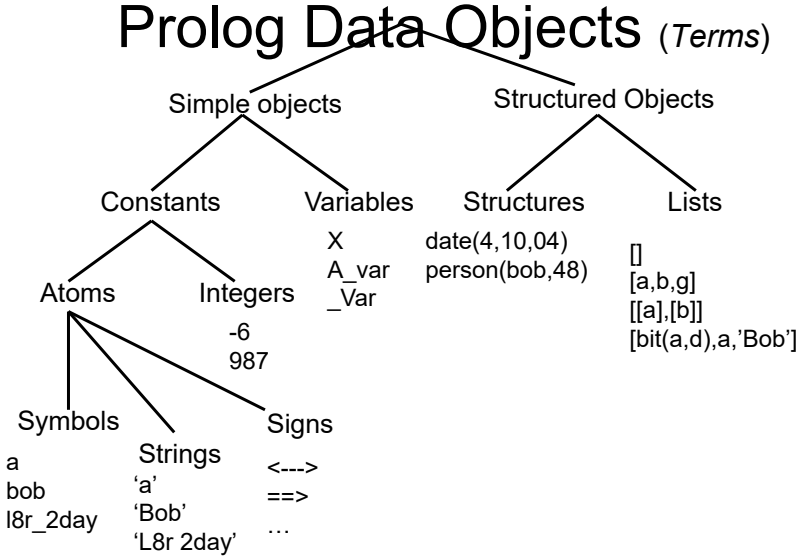
- Prolog programs are knowledge bases.
- Programs process data.
- PROLOG data objects :



Data Objects

- PROLOG data objects are called terms.
- PROLOG distinguishes between terms according to their syntactic structure.
  - Structures look different from simple objects.
- PROLOG is a typeless Language.
  - All it checks is that arithmetic operations are done with numbers and I/O is done with ASCII characters.
  - Checks are done at run time.
- PROLOG fans claim that typelessness gives great flexibility.

Prolog Data Objects (*Terms*)



THREE SYNTACIC FORMS FOR ATOMS

(1) Strings of letters, digits, and "_", starting with lowercase letter:

x x15 x_15 aBC_CBa7

alpha_beta_algorithm taxi_35

peter missJones miss_Jones2

ATOMS

(2) Strings of special characters

---> <==> <<

. < > + ++ ! .. .::.
::= []

ATOMS

(3) Strings in single quotes

‘X_35’ ‘Peter’ ‘Britney Spears’

SYNTAX FOR NUMBERS

- Integers

  1 1313 0 -55

- Real numbers (floating point)

  3.14 -0.0045 1.34E-21 1.34e-21

Numbers

- PROLOG allows both integer and floating point types.
- Integer :
  - e.g. `23`, `10243`, `-23`.
- Floats :
  - e.g. `0.23`, `1.0234`, `-12.23`.
- Floats using exponential notation :
  - e.g. `12.3e34`, `-11.2e-45`, `1.0e45`.
- PROLOG suffers from the usual problems with floating point rounding errors.
- PROLOG is terrible at numeric computation.

Features of PROLOG

- Logical variables
- Pattern matching facility
- Backtracking strategy to search for proofs

Structure of PROLOG Programs

- Data Objects
  - Facts
  - Rules
  - Query
- Variables: must begin with an upper case letter
- Constants: must begin with lowercase letter, or enclosed in single quotes

Facts

- Fact is a statement that affirms or denies something.
- Often a proposition like “It is sunny”
- represented as ‘It is sunny’ or sunny but not Sunny.

Variables

- Strings of letters, digits, and underscores, starting with uppercase letter
  - Valid variables: X Results Object2B Participant_list _x35 _335
- Lexical scope of variable names is one clause
- Underscore stands for an anonymous variable.
- Each appearance of underscore: another anon. var.
- Always begin with a capital letter
  - *?- likes (john,X).*
  - *?- likes (john, Something).*
- But *not*
  - *?- likes (john,something)*

Example of usage of variable

Facts:

likes(john,flowers).
likes(john,mary).
likes(paul,mary).

Question:

?- likes(john,X)

Answer:

X=flowers and wait
;
mary
;
no

Variables

- Scope rule:

– Two uses of an identical name for a logical variable only refer to the same object if the uses are within a single clause.

```
happy(Person) :- healthy(Person). % same person

wise(Person) :- old(Person).
/* may refer to other person than in above clause. */
```

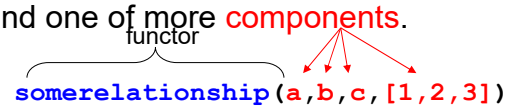
- Two commenting styles!
- a Prolog variable does not represent a location that contains a modifiable value; it behaves more like a mathematical variable (and has the same scope)

FUNCTORS

- Functor name chosen by user
- Syntax for functors: atoms
- Functor defined by name and *arity*

Structures

- To create a single data element from a collection of related terms we use a *structure*.
- A structure is constructed from a *functor* (a constant symbol) and one of more components.



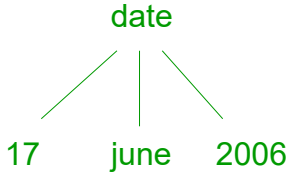
- The components can be of any type: atoms, integers, variables, or structures.
- As functors are treated as data objects just like constants they can be unified with variables

```
|?- X = date(04,10,04).
X = date(04,10,04)?
yes
```

TREE REPRESENTATION OF STRUCTURES

Often, structures are pictured as trees

date(17, june, 2006)



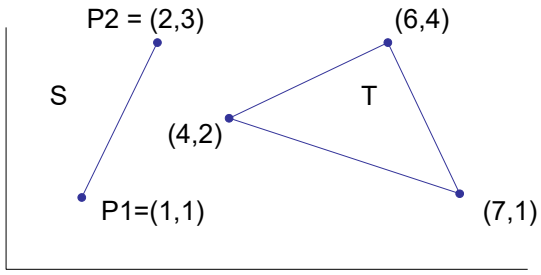
TREE REPRESENTATION OF STRUCTURES

- Therefore all structured objects in Prolog can be viewed as trees
- This is the *only* way of building structured objects in Prolog

Structures - Exercise

- Description:
  - point in the 2D space
  - triangle
  - a country
    - has a name
    - is located in a continent at a certain position
    - has population
    - has capital city which has a certain population
    - has area

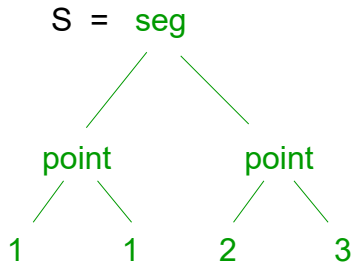
SOME GEOMETRIC OBJECTS



P1 = point(1, 1) P2 = point(2, 3)
S = seg(P1, P2) = seg(point(1,1), point(2,3))
T = triangle(point(4,2), point(5,4), point(7,1))

LINE SEGMENT

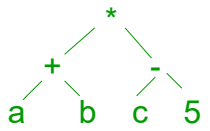
S = seg(point(1,1), point(2,3))



ARITHMETIC EXPRESSIONS ARE ALSO STRUCTURES

- For example: (a + b) * (c - 5)
- Written as term with functors:

*(+(a, b), -(c, 5))



A Particular Structure

- How can we represent the courses a student takes?
  courses(csi2111, csi2114, csi2165)
  courses(csi2114, csi2115, csi2165, mat2343)
  courses(adm2302, csi2111, csi2114, csi2115, csi2165)
- Three different structures.
- In general, how do we represent a **variable** number of arguments with a **single** structure?
- LISTS

Naming tips

- Use real English when naming predicates, constants, and variables.

  e.g. "John wants to help Somebody."

  Could be: `wants(john,to_help,Somebody).`

  Not: `x87g(j,_789).`

- Use a Verb Subject Object structure:

  `wants(john,to_help).`

- **BUT** do not assume Prolog Understands the meaning of your chosen names!
  - You create meaning by specifying the body (i.e. preconditions) of a clause.

Clauses/Rules

- In Prolog, rules (and facts) are called **clauses.**
- **A clause always ends with '.'**
  - Clause: **<head> :- <body>.**
    - you can conclude that <head> is true, if you canprove that <body> is true
  - Facts - clauses with an empty body: <head>.
    - you can conclude that <head> is true
  - Rules - normal clauses (or more clauses)
  - Queries - clauses with an empty head: ?- <body>.
    - Try to prove that <body> is true

Queries

- The goal represented as a question.
- Once we have a Prolog program we can use it to answer a query.

  ?- parent(philip, anne).

  ?- border(wales, scotland).
- The Prolog interpreter responds ‘yes’ or ‘no’.
- Note that all queries must end with a dot.
- A query may contain variables:
- ?- parent(philip, **Who**).
- The Prolog interpreter will respond with the values for the variables which make the query true (if any).
- Otherwise it will respond with a ‘no’.

Questions/Queries

- *Questions* based on facts
- Answered by *matching*

Two facts *match* if their predicates are same (spelt the same way) and the arguments each are same.

- If matched, prolog answers *yes*, else *no*.
- *No* does not mean falsity.

Prolog does *theorem proving*

- When a question is asked, prolog tries to match *transitively.*
- When no match is found, answer is *no.*
- This means *not provable* from the given facts.

DECLARATIVE vs PROCEDURAL MEANING OF PROLOG PROGRAMS

- Consider:

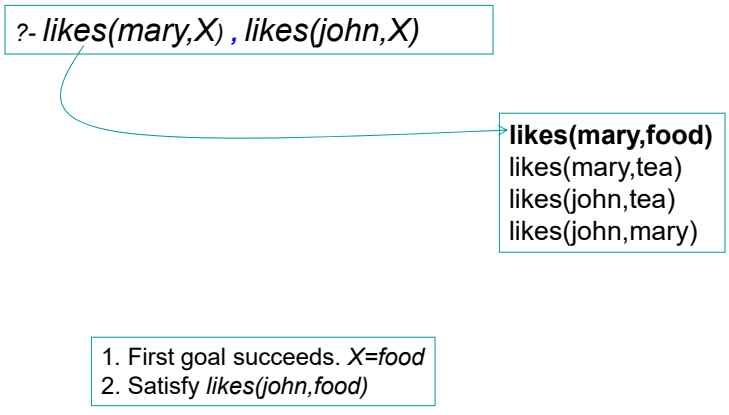
  P :- Q, R.

- Declarative readings of this:
  - P is true if Q *and* R are true.
  - From Q *and* R follows P.
- Procedural readings:
  - To solve problem P, *first* solve subproblem Q and *then* R.
  - To satisfy P, *first* satisfy Q and *then* R.

Conjunctions

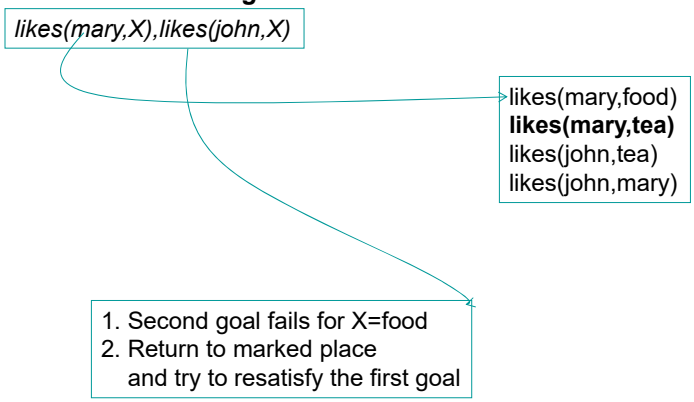
- Use ‘,’ and pronounce it as *and*.
- Example
  - Facts:
    - likes(mary,food).
    - likes(mary,tea).
    - likes(john,tea).
    - likes(john,mary)
- ?- likes(mary,X),likes(john,X).
  - Meaning *is anything liked by Mary also liked by John?*
  - *X = tea*

Backtracking *(an inherent property of prolog programming)*

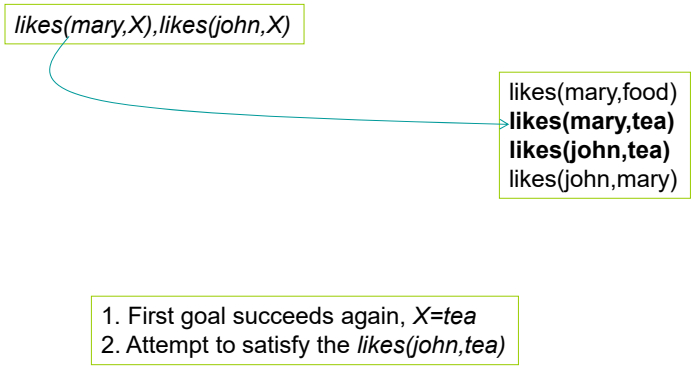


Backtracking *(continued)*

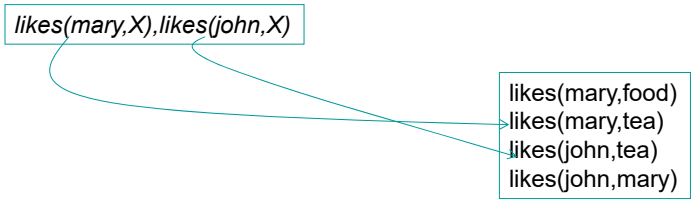
Returning to a marked place and trying to resatisfy is called *Backtracking*



Backtracking *(continued)*



Backtracking *(continued)*



1. Second goal also suceeds
2. Prolog notifies success and waits for a reply

Disjunctions

- The symbol ; is read as OR operator.

  ?- likes(mary,X) **;** likes(john,X).

  - X = food ;
  - X = tea ;
  - X = tea ;
  - X = mary.

Rules

- Statements about *objects* and their *relationships*
- Expess
  - *If-then conditions*
    - *I use an umbrella if there is a rain*
    - *use(i, umbrella) :- occur(rain).*
  - *Generalizations*
    - *All men are mortal*
    - *mortal(X) :- man(X).*
  - *Definitions*
    - *An animal is a bird if it has feathers*
    - *bird(X) :- animal(X), has_feather(X).*

Rules

- Rules state information that is *conditionally* true of the domain of interest.
- The general form of these properties
  - p is true if (p1 is true, and p2 is true, … and pn is true)
- Horn clause
  - p :- p1, p2, …, pn.
- Interpretation (Prolog) :
  - in order to prove that p is true, the interpreter will provethat each of p1, p2, …, pn is true
  - p - the **head** of the rule
  - p1, p2, …, pn - the **body** of the rule (**subgoals**)

Rules and Disjunctions

- Someone is happy if he/she is healthy, wealthy or wise.
- In Prolog:
- More exactly:
  - Someone is happy if they are healthy OR
  - Someone is happy if they are wealthy OR
  - Someone is happy if they are wise.

```
happy(Person) :- healthy(Person).
happy(Person) :- wealthy(Person).
happy(Person) :- wise(Person).
```

Both Disjunctions and Conjunctions

- A woman is happy if she is healthy or wealthy or wise.
- In Prolog:

```
happy(P) :- healthy(P), woman(P).
happy(P) :- wealthy(P), woman(P).
happy(P) :- wise(P), woman(P).
```

Anonymous variables

- a variable that stands in for some unknown object
- stands for some objects about which we don't care
- several anonymous variables in the same clause
- need not be given consistent interpretation
- written as _ in Prolog

  ?- birth_date(_ , X , _). //for birth_date(day , month , year).

  X = October;

  X = April;

  …
- We are interested in the names of months but not their day and year of birth date.

MATCHING

- Matching is operation on terms (structures)
- Given two terms, they match if:

  (1) They are identical, or

  (2) They can be made identical by properly instantiating the variables in both terms

Matching / Unification of 2 goals

- Matching two dates:
  date(D1, M1, 2006) = date(D2, june, Y2)
- This causes the variables to be instantiated as:
  D1 = D2
  M1 = june
  Y2 = 2006
- This is the *most general instantiation*
- A less general instantiation would be: D1=D2=17, ...

Unification

- When two terms match we say that they unify.
  - Their structures and arguments are compatible.
- This can be checked using =/2

```
|?- loves(john,X) = loves(Y,mary).
X = mary,   ← unification leads to instantiation
Y = john?   ←
yes
```

Terms that don't unify
fred = jim.
'Hey you' = 'Hey me'.
frou(frou) = f(frou).
foo(bar) = foo(bar,bar).
foo(N,N) = foo(bar,rab).

Terms that unify	Outcome
fred = fred.	yes.
'Hey you' = 'Hey you'.	yes
fred=X.	X=fred.
X=Y.	Y = X.
foo(X) = foo(bar).	X=bar.
foo(N,N) = foo(bar,X).	N=bar, X=bar.
foo(foo(bar)) = foo(X)	X = foo(bar)

MOST GENERAL INSTANTIATION

- In Prolog, matching always results in most general instantiation
- This commits the variables to the least possible extent, leaving flexibility for further instantiation if required
- For example:

```
?- date( D1, M1, 2006)  =  date( D2, june, Y2),
   date( D1, M1, 2006)  =  date( 17, M3, Y3).
```

D1 = 17, D2 = 17, M1 = june, M3 = june,
Y2 = 2006, Y3 = 2006

MATCHING

- Matching succeeds or fails; if succeeds then it results in the most general instantiation
- To decide whether terms S and T match:

  (1) If S and T are constants then they match only if they are identical

  (2) If S is a variable then matching succeeds, S is instantiated to T; analogously if T is a variable

  (3) If S and T are structures then they match only if

  (a) they both have the same principal functor, and

  (b) all their corresponding arguments match

MATCHING ≈ UNIFICATION

- Unification known in predicate logic
- Unification = Matching + Occurs check
- What happens when we ask Prolog:

  ?- X = f(X).

  Matching succeeds, unification fails

COMPUTATION WITH MATCHING

```
% Definition of vertical and horizontal segments
vertical( seg( point( X1,Y1), point( X1, Y2))).
horizontal( seg( point( X1,Y1), point( X2, Y1))).
?- vertical( seg( point( 1,1), point( 1, 3))).
yes
?- vertical( seg( point( 1,1), point( 2, Y))).
no
?- vertical( seg( point( 2,3), P)).
P = point( 2, _173).
```

AN INTERESTING SEGMENT

- Is there a segment that is both vertical and horizontal?

?- vertical(S), horizontal(S).

S = seg(point(X,Y), point(X,Y))

- Note, Prolog may display this with new variables names as for example:

S = seg(point(_13,_14), point(_13, _14))

Using predicate definitions

- Command line programming is tedious

  e.g.
```
| ?- write('What is your name?'), nl, read(X),
     write('Hello '), write(X).
```

- We can define predicates to automate commands:

```
greetings:-
    write('What is your name?'),
    nl,
    read(X),
    write('Hello '),
    write(X).
```

Prolog Code

```
?- greetings.
What is your name?
|: tim.
Hello tim
X = tim ?
yes
```

Terminal

Arity

- `greetings` is a predicate with no arguments.
- The number of arguments a predicate has is called its arity.
  - The arity of `greetings` is zero = greetings/0
- The behaviour of predicates can be made more specific by including more arguments.
  - `greetings(hamish)` = greetings/1
- The predicate can then behave differently depending on the arguments passed to it.

Using multiple clauses

- Different clauses can be used to deal with different arguments.

```
greet(hamish):-
    write('How are you doin, pal?').
greet(amelia):-
    write('Awfully nice to see you!').
```

= "Greet Hamish or Amelia" = a disjunction of goals.

```
| ?- greet(hamish).
How are you doin, pal?
yes
```

```
| ?- greet(amelia).
Awfully nice to see you!
yes
```

- Clauses are tried in order from the top of the file.
- The first clause to match succeeds (= yes).

Variables in Questions

- We can call `greet/1` with a variable in the question.
- A variable will match any head of `greet/1`.

```
| ?- greet(Anybody).
How are you doin, pal?
Anybody = hamish?
yes
```

- The question first matches the clause closest to the top of the file.
- The variable is instantiated (i.e. bound) to the value '`hamish`'.
- As the variable was in the question it is passed back to the terminal (`Anybody = hamish?`).

Re-trying Goals

- When a question is asked with a variable as an argument (e.g. `greet(Anybody).`) we can ask the Prolog interpreter for multiple answers using: **`;`**

```
| ?- greet(Anybody).
How are you doin, pal?
Anybody = hamish? ;    ← "Redo that match."
Anybody = amelia? ;    ← "Redo that match."
no                     ← "Fail as no more matches."
```

Variable clause head.

- If `greet/1` is called with a constant other than `hamish` or `amelia` it will fail (return `no`).
- We can create a default case that always succeeds by writing a clause with a variable as the head argument.

```
greet(Anybody):-
    write('Hullo '),
    write(Anybody).
```

```
|?- greet(bob).
Hullo bob.
yes
```

- Any call to `greet/1` will unify (i.e. match) `greet(Anybody).`
- Once the terms unify the variable is instantiated to the value of the argument (e.g. `bob`).

Ordering of clauses

- The order of multiple clauses is important.

```
greet(Anybody):-
        write('Hullo '), write(Anybody).

greet(hamish):-
   write('How are you doin, pal?').

greet(amelia):-
   write('Awfully nice to see you!').
```

```
| ?- greet(hamish).
Hullo hamish?
yes
```

- The most specific clause should always be at the top.
- General clauses (containing variables) at the bottom.

Ordering of clauses

- The order of multiple clauses is important.

```
| ?- greet(hamish).
How are you doin,
pal?.
yes
```

```
greet(hamish):-
   write('How are you doin, pal?').

greet(amelia):-
   write('Awfully nice to see you!').

          greet(Anybody):-
                 write('Hullo '), write(Anybody).
```

- The most specific clause should always be at the top.
- General clauses (containing variables) at the bottom.

Arithmetic Operators

- + addition
- - subtraction
- * multiplication
- / real division
- // integer division
- mod modulus
- ** power

- ?- X is 3*4.
  X = 12
  yes

Some Built-in Predicates(Operators)

for constants:

- number/1
- integer/1
- float/1
- atom/1
- atomic/1

Examples:

number(15)	atom(my_atom)
number(0.001)	atom(*)
number(4.2E+01)	atom('This?')
integer(16)	atom(15)
integer(1.0)	atomic(a)
float(1)	atomic(4)
float(1.5E-1)	atomic(4.2E+01)
float(1.0)	

for variables:

- var/1
- nonvar/1
- is/2
- =/2
- …

Examples:

var(X)	X = abc, var(X)
var(x)	X = abc,nonvar(X)
var(5)	_ = abc, var(_)
nonvar(X)	_ = abc, nonvar(_)
nonvar(abc)	X is 5
Y = abc	X is 5+1
Z = 4.2E+01	X = 5+1
var(X), X = abc	X = 5

Try them out on your computer!

Logical Operators

- a :- b. % a if b
- a :- b,c. % a if b and c.
- a :- b;c. % a if b or c.
- a :- \++ b. % a if b is not provable
- a :- not b. % a if b fails
- not(S) :- S, ! , fail ; true.
- a :- b -> c ; d. % a if (if b then c else d)

```
%SWI program
not(S) :- S, ! , fail ; true.
b.
c.
d.
e.
a :- b.                % a if b
p :- b,c.              % p if b and c.
q :- b;d.              % q if b or d.
%k :- \++ b.     % k if b is not
provable
r :- not(b).     % a if b fails
r :- \+ b.         % r if b is not true
m :- b -> c ; d. % m if (if b then c
else d)
n:-b; c, d; e.
j:-b; c; d; e.
```

```
4 ?- consult('G:/OR_demo.pl').
% G:/OR_demo.pl compiled 0.00 sec, 0 bytes
true.
5 ?- a.
true.
6 ?- p.
true.
7 ?- q.
true ;
;
true.
8 ?- |    r.
false.
9 ?- m.
true.

10 ?- n.
true ;
;
true ;
;
true.
11 ?- j.
true ;
;
true ;
;
true ;
;
true.
```

Input/output

- *built-in* predicates for input and output work as follows:

  - *input predicate*: proof is interrupted and user is asked for input,
    input is unified with the argument of the input, proof is continued

  - *output predicate*: argument shown on display, proof is continued

- No *backtracking*!
  => take into account procedural behavior

- *built-in* predicates for reading resp. writing Prolog terms:
  read/1 write/1

- *built-in* predicates for reading resp. writing characters:
  get0/1 put/1 nl/0 tab/1

Success and failure

- *true*: always successful

  equivalent:

```
fact(a,b,c).
fact(a,b,c) :- true.
```

- *fail*: always fails

  Use of *fail*: compute all solutions for a given goal:

Arithmetic expressions

- For computing numeric values without unification, Prolog provides arithmetic operators:

  `+`, `-`, `*`, `/`, `//`, `mod`, `/\`, `\/`, `...`

  and the evaluation operator `is`.

- `X is Y` is true if `Y` is an arithmetic expression, where all variables are instantiated to numbers and `X` is unified with the result of evaluating `Y`.

- `X < Y` holds if the evaluation of `X` gives a value smaller than the evaluation of `Y`.

- Analogously, `X =< Y`, `X > Y`, `X >= Y`, `X =:= Y`, `X =\= Y`

- The arguments have to be fully instantiated with arithmetic expressions (so that they can be evaluated).

```
?-X is 5+7-3.
  X = 9

?-9 is 5+7-3.
  Yes

?-9 is X+7-3.
  Error in arithmetic expression

?-X is 5*3+7/2.
  X = 18.5
```

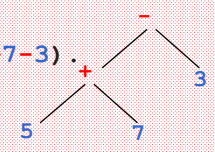
```
?-X = 5+7-3.
  X = 5+7-3

?-9 = 5+7-3.
  No

?-9 = X+7-3.
  No

?-X = Y+7-3.
  X = _947+7-3
  Y = _947
```

```
?-display(5+7-3).
-(+(5,7),3)
```



Prolog arithmetic vs.unification

Summary

- A Prolog program consists of predicate definitions.
- A predicate denotes a property or relationship between objects.
- Definitions consist of clauses.
- A clause has a head and a body (Rule) or just a head (Fact).
- A head consists of a predicate name and arguments.
- A clause body consists of a conjunction of terms.
- Terms can be constants, variables, or compound terms.
- We can set our program goals by typing a command that unifies with a clause head.
- A goal unifies with clause heads in order (top down).
- Unification leads to the instantiation of variables to values.
- If any variables in the initial goal become instantiated this is reported back to the user.