Second-order predicates: assert/1 and retract/1

[Slides on Computational Logic from CLIP group]

Powerful: enable run-time program modification

Harmful: code hard to understand and debug, often slow

sometimes used as global variables, “boolean” flags or to memoize:

```
fib(0,0).
fib(1,1).
fib(N,F) :-
  N > 1,
  N1 is N-1,
  N2 is N1-1,
  fib(N1,F1),
  fib(N2,F2),
  F is F1+F2.
```

```
mfib(N, F):- memo_fib(N, F), !.
mfib(N, F):-
  N > 1,
  N1 is N-1,
  N2 is N1-1,
  mfib(N1,F1),
  mfib(N2,F2),
  F is F1+F2,
  assert(memo_fib(N, F)).

:- dynamic memo_fib/2.
memo_fib(0,0).
memo_fib(1,1).
```

if you’ve remembered an answer for this goal before, return it

most Prologs require such a declaration for clauses that are added or removed from the program at run-time

Higher-order programming using call/N: call(Goal,...)

a more flexible form of call/1, which takes additional arguments that will be added to the Goal that is called

[Higher-order logic programming in Prolog, Lee Naish, 1996]

```
call(p(X1,X2,X3))
call(p(X1,X2), X3)
call(p(X1), X2, X3)
call(p, X1, X2, X3)
```

all result in `p(X1, X2, X3)` being called

Supported by most Prolog systems in addition to call/1

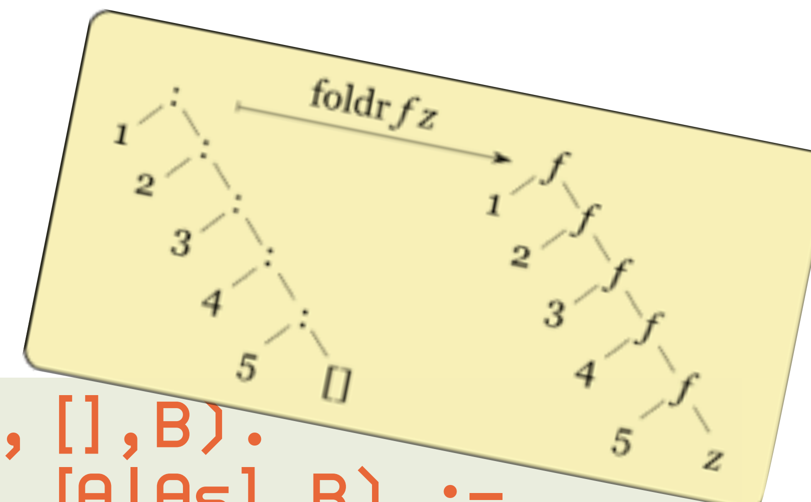
can often be used in places where you would use univ operator =.. to construct the goal

Higher-order programming using call/N: implementing map and friends

[Higher-order logic programming in Prolog, Lee Naish, 1996]

```
map(_F, [], []).
map(F, [A0|As0], [A|As]) :-
   call(F,A0,A),
   map(F,As0,As).
```

```
filter(_P, [], []).
filter(P, [A0|As0],As) :-
   (call(P, A0) ->
     As = [A0|As1]
    ;As = As1),
   filter(P, As0, As1)
```




```
foldr(F,B, [],B).
foldr(F,B, [A|As],R) :-
   foldr(F,B,As,R1),
   call(F,A,R1,R).
```

```
compose(F,G,X,FGX):-
   call(G,X,GX),
   call(F,GX,FGX).
```

Higher-order programming using call/N: using map and friends (1)

[Higher-order logic programming in Prolog, Lee Naish, 1996]

```
?- filter(>(5),[3,4,5,6,7],As).
As=[3,4]
```

called goal: >(5,X)

```
?- map(plus(1),[2,3,4],As).
As=[3,4,5]
```

```
?- map(between(1),[2,3],As).
As=[1,1]; As=[1,2]; As=[1,3];
As=[2,1]; As=[2,2]; As=[2,3]
```

between(I,J,X) binds X to an integer between I and J inclusive.

```
?- map(plus(1),As,[3,4,5]).
As=[2,3,4]
```

assuming that plus/3 is reversible (e.g., Peano arithmetic)

```
?- map(plus(X),[2,3,4],[3,4,5]).
X=1
```

```
?- map(plus(X),[2,A,4],[3,4,B]).
X=1,A=3,B=5
```

relies on execution order in which X is bound first

Higher-order programming using call/N: using map and friends (2)

flatten defined in terms of foldr using empty list and append

```
?- foldr(append, [], [[2], [3,4], [5]],As).
As= [2,3,4,5]

?- compose(map(plus(1)),foldr(append, [])), [[2], [3,4], [5]],As).
As= [3,4,5,6]
```

flattens first, then adds 1

plain Prolog lacks "currying" for higher-order programming: functional programming languages would return a list of functions that take the missing argument

conceptual difficulty: ok to curry a call(sum(2,3)) to a sum(2,3,Z) if there is also a definition for sum(X,Y)?

```
?- map(plus, [2, 3, 4], As).
ERROR: map/3: Undefined procedure: plus/2
ERROR:     However, there are definitions for:
ERROR:         plus/3
```

[Higher-order logic programming in Prolog, Lee Naish, 1996]

Inspecting terms:
var/1 and its use in practice

var(Term)

succeeds when Term is an uninstantiated variable

nonvar(Term) has opposite behavior

```
?- var(X).
true.
?- X=3,var(X).
false.
```

ensuring relational nature of predicates

```
plus(X,Y,Z) :-
  nonvar(X),nonvar(Y),Z is X+Y.
plus(X,Y,Z) :-
  nonvar(X),nonvar(Z),Y is Z-X.
plus(X,Y,Z) :-
  nonvar(Y),nonvar(Z),X is Z-Y.
```

directing search for efficiency

```
grandparent(X,Z) :-
   nonvar(X),parent(X,Y),parent(Y,Z).
grandparent(X,Z) :-
   nonvar(Z),parent(Y,Z),parent(X,Y).
```

Inspecting terms: arg/3 and functor/3

complement =.. operator

arg(N,Term,Arg)
succeeds when Arg is the Nth argument of Term

functor(Term,F,N)
succeeds when the Term starts with the functor F of arity N

tests whether a term is ground (i.e., contains no uninstantiated variables)

```
ground(Term) :-
  nonvar(Term),constant(Term).
ground(Term) :-
  nonvar(Term),
  compound(Term),
  functor(Term,F,N),
  ground(N,Term).
ground(N,Term) :-
   N > 0,
   arg(N,Term,Arg),
   ground(Arg),
   N1 is N-1,
   ground(N1,Term).
ground(0,Term).
```

common Prolog practice: arity of auxiliary and main predicates differ

Extending Prolog: term_expansion(+In,-Out)

called by Prolog for each file it compiles

clause or list of clauses that will be added to the program instead of the In clause

useful for generation code, e.g. :

given compound term representation of data

```
student(Name,Id)
```

want to use accessor predicates

```
student_name(student(Name, _), Name).
student_id(student(_, Id), Id).
```

instead of explicit unifications throughout the code

```
Student = student(Name,_)
```

to ensure independence of one particular representation of the data

[http://ww2.cs.mu.oz.au/255/last_semester/lec/subject-prolog_meta.pdf]

Extending Prolog: term_expansion(+In,-Out)

```
:- struct student(name,id).
```

```
student_name(student(Name, _), Name).
student_id(student(_, Id), Id).
```

declares struct as a prefix operator

```
:- op(1150, fx, (struct)).

term_expansion((:- struct Term), Clauses) :-
   functor(Term, Name, Arity),
   functor(Template, Name, Arity),
   gen_clauses(Arity, Name, Term, Template, Clauses).
```

create Template with same functor and arity, but with variable arguments rather than constants

[http://ww2.cs.mu ... t_semester/lec/subject-prolog_meta.pdf]

Extending Prolog: term_expansion(+In,-Out)

N-th argument recursed upon

trick to merge recursive and base clause

conversion from atom to list of character codes

```
?- X=0'_.
X = 95.

?- char_code(X,95).
X = '_'.
```

```
gen_clauses(N, Name, Term, Template, Clauses) :-
  (N =:= 0 ->
     Clauses = []
  ;arg(N, Term, Argname),
   arg(N, Template, Arg),
   atom_codes(Argname, Argcodes),
   atom_codes(Name, Namecodes),
   append(Namecodes, [0'_|Argcodes],Codes),
   atom_codes(Pred, Codes),
   Clause =.. [Pred, Template, Arg],
   Clauses = [Clause|Clauses1],
   N1 is N - 1,
   gen_clauses(N1, Name, Term, Template, Clauses1)
  ).
```

creates fact

When trying out, put gen_clauses/5 before term_expansion/2

[http://ww2.cs.mu.oz.au/255/last_semester/last_semester/lec/subject-prolog_meta...

Extending Prolog: operators

Certain functors and predicate symbols that be used in infix, prefix, or postfix rather than term notation.

```
:- op(500,xfx,'has_color').
a has_color red.
b has_color blue.
```

```
?- b has_color C.
C = blue.
?- What has_color red.
What = a
```

integer between 1 and 1200;
smaller integer binds stronger
a+b/c ≡ a+(b/c) ≡ +(a,/(b,c)) if / smaller than +

```
:- op(Precedence, Type, Name)
```

prefix: fx, fy
infix: xfx, xfy,yfx
postfix: xf,yf

associative	not	right	left
	xfx	xfy	yfx
X op Y op Z	/	op(X,op(Y,Z))	op(op(X,Y),Z)

Extending Prolog: operators in towers of Hanoi

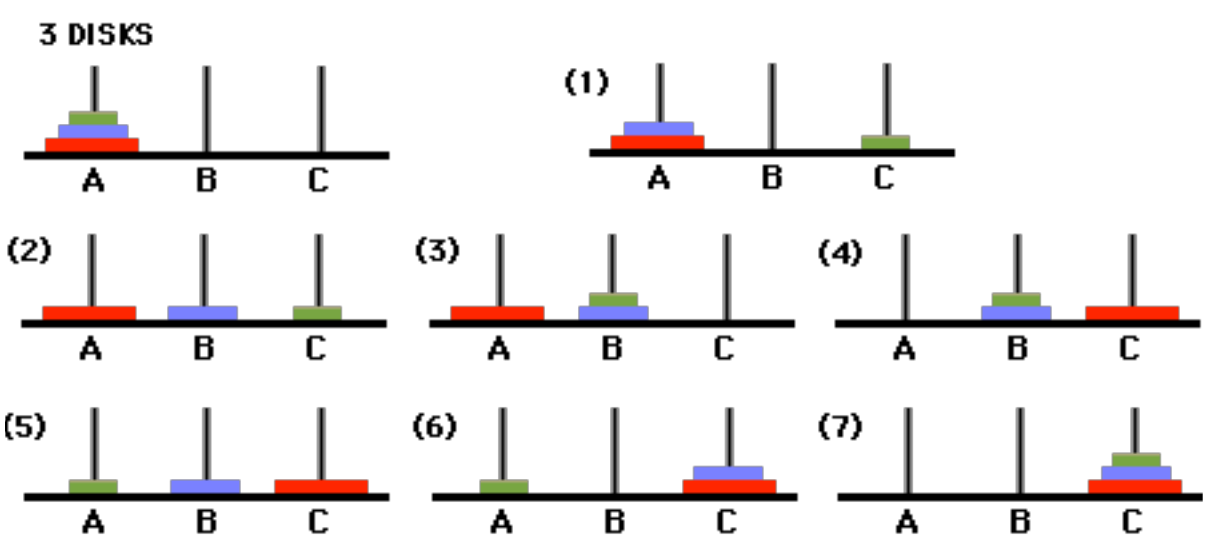
Moves is the list of moves to move N discs from peg A to peg C, using peg B as an intermediary.

```
:- op(900,xfx,to).
hanoi(0,A,B,C,[]).
hanoi(N,A,B,C,Moves):-
  N1 is N-1,
  hanoi(N1,A,C,B,Moves1),
  hanoi(N1,B,A,C,Moves2),
  append(Moves1,[A to C|Moves2],Moves).
```

move n-1 c from A to B. disc #n is left on A

move n-1 discs from B to C. they will rest on disc #n

move disc #n from A to C




```
?- hanoi(3,left,middle,right,M)
M = [left to right,
     left to middle,
     right to middle,
     left to right,
     middle to left,
     middle to right,
     left to right ]
```

Extending Prolog: built-in operators

```
1200 xfx -->, :-
1200 fx  :-, ?-
1150 fx  dynamic, discontiguous, initialization, meta_predicate, module_
1100 xfy ;, |
1050 xfy ->, op*->
1000 xfy ,
900 fy   \+
900 fx   ~
700 xfx <, =, =.., =@=, =:=, =<, ==, =\=, >, >=, @<, @=<, @>, @>=, \=, \==, is
600 xfy :
500 yfx +, -, /\, \/, xor
500 fx   ?
400 yfx *, /, //, rdiv, <<, >>, mod, rem
200 xfx **
200 xfy ^
200 fy   +, -, \
```

```
+'(a,'/'(b,c))                 a+b/c
is(X, mod(34, 7))              X is 34 mod 7
<'('+'(3,4),8)                 3+4<8
'='(X,f(Y))                    X=f(Y)
'-'(3)                         -3
':-'(p(X),q(Y))                p(X) :- q(Y)
':-'(p(X),','(q(Y),r(Z)))      p(X) :- q(Y),r(Z)
```

clauses are also Prolog terms!

[Slides on Computational Logic from CLIP group]

Extending Prolog: vanilla and canonical naf meta-interpreter

```
prove(Goal):-
  clause(Goal,Body),
  prove(Body).

prove((Goal1,Goal2)):-
  prove(Goal1),
  prove(Goal2).

prove(true).
```

Are these meta-circular interpreters?

```
prove(true):- !.

prove((A,B)):- !,
  prove(A),
  prove(B).

prove(not(Goal)):- !,
  not(prove(Goal)).

prove(A):-
  % not (A=true; A=(X,Y); A=not(G))
  clause(A,B),
  prove(B).
```

Avoids problems where clause/2 is called with a conjunction or true.

Availability: *built-in*

clause(*:Head, ?Body*) *[ISO]*

True if *Head* can be unified with a clause head and *Body* with the corresponding clause body. Gives alternative clauses on backtracking. For facts *Body* is unified with the atom *true*.

Extending Prolog: meta-level vs object-level in meta-interpreter

	KNOWLEDGE	REASONING
META-LEVEL	`clause(p(X),q(X)).` `clause(q(a),true).`	`?-prove(p(X)).` `X=a`
OBJECT-LEVEL	`p(X):-q(X).` `q(a).`	`?-p(X).` `X=a`

Canonical meta-interpreter still **absorbs** backtracking, unification and variable environments implicitly from the object-level.

Reified unification explicit at meta-level :

```
prove(A):-
   clause(Head,Body),
   unify(A,Head,MGU,Result),
   apply(Body,MGU,NewBody),
   prove_var(NewBody).
```

Prolog programming: a methodology illustrated on partition/4

(might not work equally well for everyone)

1 Write down declarative specification

```
% partition(L,N,Littles,Bigs) <- Littles contains numbers
%                                 in L smaller than N,
%                                 Bigs contains the rest
```

2 Identify recursion and "output" arguments

what is the recursion argument?
what is the base case?

3 Write down implementation skeleton

```
partition([],N,[],[]).
partition([Head|Tail],N,?Littles,?Bigs):-
  /* do something with Head */
  partition(Tail,N,Littles,Bigs).
```

Empty list is partitioned into two empty lists.

We recurse on the "input" argument list.

Prolog programming: a methodology illustrated on partition/4

4 Complete bodies of clauses

```
partition([],N,[],[]).
partition([Head|Tail],N,?Littles,?Bigs):-
   Head < N,
   partition(Tail,N,Littles,Bigs),
   ?Littles = [Head|Littles],?Bigs = Bigs.
partition([Head|Tail],N,?Littles,?Bigs):-
   Head >= N,
   partition(Tail,N,Littles,Bigs),
   ?Littles = Littles,?Bigs = [Head|Bigs].
```

Head is smaller, has to be added to Littles

has to be added to Bigs otherwise

5 Fill in "output" arguments

```
partition([],N,[],[]).
partition([Head|Tail],N,[Head|Littles],Bigs):-
   Head < N,
   partition(Tail,N,Littles,Bigs).
partition([Head|Tail],N,Littles,[Head|Bigs]):-
   Head >= N,
   partition(Tail,N,Littles,Bigs).
```

Prolog programming: a methodology illustrated on sort/2

1 Write down declarative specification

```
% sort(L,S) <- S is a sorted permutation of list L
```

2 Identify recursion and "output" arguments

3 Write down implementation skeleton

```
sort([],[]).
sort([Head|Tail],?Sorted):-
  /* do something with Head */
  sort(Tail,Sorted).
```

4 Complete bodies of clauses

```
sort([],[]).
sort([Head|Tail],WholeSorted):-
  sort(Tail,Sorted),
  insert(Head,Sorted,WholeSorted).
```

Auxiliary predicate

Prolog programming:
a methodology illustrated on insert/3

1 Write down declarative specification

```
% insert(X,In,Out) <- In is a sorted list, Out is In
%                      with X inserted in the proper place
```

2 Identify recursion and "output" arguments

3 Write down implementation skeleton

```
insert(X,[],?Inserted).
insert(X,[Head|Tail],?Inserted):-
  /* do something with Head */
  insert(X,Tail,Inserted).
```

Prolog programming: a methodology illustrated on insert/3

4 Complete bodies of clauses

```
insert(X, [],?Inserted):-
  ?Inserted=[X].
insert(X, [Head|Tail],?Inserted):-
  X > Head,
  insert(X,Tail,Inserted),
  ?Inserted = [Head|Inserted].
insert(X, [Head|Tail],?Inserted):-
  X =< Head,
  ?Inserted = [X,Head|Tail].
```

5 Fill in “output” arguments

```
insert(X, [], [X]).
insert(X, [Head|Tail], [X,Head|Tail]):-
  X =< Head.
insert(X, [Head|Tail], [Head|Inserted]):-
  X > Head,
  insert(X,Tail,Inserted).
```

More Prolog programming: quicksort

```
quicksort([], []).
quicksort([X|Xs],Sorted):-
   partition(Xs,X,Littles,Bigs),
   quicksort(Littles,SortedLittles),
   quicksort(Bigs,SortedBigs),
   append(SortedLittles, [X|SortedBigs],Sorted).
```

with difference lists:

```
quicksort(Xs,Ys) :- qsort(Xs,Ys-[]).

qsort([],Ys-Ys).
qsort([X0|Xs],Ys-Zs) :-
   partition(Xs,X0,Ls,Bs),
   qsort(Bs,Ys2-Zs),
   qsort(Ls,Ys-[X0|Ys2]).
```