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

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\(\text{N}\):
call(Goal,...)

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

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 uniop operator =.. to construct the goal
Higher-order programming using call/N: implementing map and friends

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

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).

filter(_P, [], []).  
filter(P, [A0|As0], As) :-  
    (call(P, A0) ->  
     As = [A0|As1]  
     ;As = As1),  
    filter(P, As0, As1)
Higher-order programming using call/\(N\): using map and friends (1)

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

?- 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]

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

?- 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

called goal: >\(5,X)\)

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

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

relies on execution order in which \(X\) is bound first

[Higher-order logic programming in Prolog, Lee Naish, 1996]
Higher-order programming using call/N: using map and friends (2)

?- 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]

flatten defined in terms of foldr using empty list and append

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: plus/3
Inspecting terms: var/1 and its use in practice

\( \text{var}(\text{Term}) \)

succeeds when Term is an uninstantiated variable
\( \text{nonvar}(\text{Term}) \) has opposite behavior

\[
\text{plus}(X,Y,Z) :- \\
\quad \text{nonvar}(X),\text{nonvar}(Y),Z \text{ is } X+Y. \\
\text{plus}(X,Y,Z) :- \\
\quad \text{nonvar}(X),\text{nonvar}(Z),Y \text{ is } Z-X. \\
\text{plus}(X,Y,Z) :- \\
\quad \text{nonvar}(Y),\text{nonvar}(Z),X \text{ is } Z-Y.
\]

\[
\text{grandparent}(X,Z) :- \\
\quad \text{nonvar}(X),\text{parent}(X,Y),\text{parent}(Y,Z). \\
\text{grandparent}(X,Z) :- \\
\quad \text{nonvar}(Z),\text{parent}(Y,Z),\text{parent}(X,Y).
\]

?- var(X).
  true.
?- X=3,var(X).
  false.
Inspecting terms: arg/3 and functor/3

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

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).

ground(complement(..)):=
operator
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
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
Extending Prolog:

term_expansion(+In,-Out)

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

trick to merge recursive and base clause

conversion from atom to list of character codes

N-th argument recursed upon

http://www2.cs.mu.oz.au/255/last_semester/last_semester/lec/subject-prolog_meta.pdf

?- X=0'_.
X = 95.
?- char_code(X,95).
X = '_'.
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

\[ a + b/c \equiv \frac{a + b}{c} \equiv \frac{a}{b + \frac{c}{a}} \] if \( / \) smaller than +

:- op(Precedence, Type, Name)

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

associative not right left

+ xfx xfy yfx
\[ X \text{ op } Y \text{ op } Z \quad / \quad \text{op}(X, \text{op}(Y, Z)) \quad \text{op}(\text{op}(X, Y), Z) \]
Extending Prolog: operators in towers of Hanoi

:- 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 disc 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

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

?- 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

+’(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!
Extending Prolog: vanilla and canonical naf meta-interpreter

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

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

prove(true).
prove(true):- !.

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

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

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

% not (A=true; A=(X,Y); A=not(G))

Are these meta-circular interpreters?

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

clause(:Head, ?Body)

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

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

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

**Complete bodies of clauses**

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

**Fill in “output” arguments**

1. `partition([], N, [], [])`.
2. `partition([Head|Tail], N, [Head|Littles], Bigs):-
   Head < N,
   partition(Tail, N, Littles, Bigs).`
3. `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

```prolog
% 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

```prolog
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).

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

with difference lists: