Tools

- Side-Effects

  `assert(C), asserta(C), assertz(C)`

  `retract(C), abolish(C,N)`

- Program as Data

  `clause(RHS,LHS), =..(X,L), call(C)`

- Utilities

  `setof(X,P,S), bagof(X,P,B)`

  `is, =, =:=`

  `op, ...`

Side-Effects

- `assert(C)`

  adds `C` to database

- `asserta(C)`

  adds `C` to *start* of database

- `assertz(C)`

  adds `C` to *end* of database

- `retract(C)`

  deletes first clause of database that matches `C`.

- `abolish(C,N)`

  abolish *all* clauses using predicate `C` of arity `N`.

Examples of assertz

```
| ?- [user].
| man(john)
| man(fred)
| ^D user con...
yes
```

% Here $KB_0 = \left\{ \begin{array}{ll} (1) & \texttt{man(john)} \\ (2) & \texttt{man(fred)} \end{array} \right\}$

```
| ?- man( X ).
X = john ;
X = fred ;
no
| ?- assertz( man(tom) )
yes
```

% Form $KB_1 = \left\{ \begin{array}{ll} (1) & \texttt{man(john)} \\ (2) & \texttt{man(fred).} \\ (3) & \texttt{man(tom).} \end{array} \right\}$

```
| ?- man( X ).
X = john ;
X = fred ;
X = tom ;
no
```

assertz (con't)

$$\% \text{ Using } \quad KB_1 = \left\{ \begin{array}{ll} (1) & \texttt{man(john)} \\ (2) & \texttt{man(fred)} \\ (3) & \texttt{man(tom).} \end{array} \right\}$$

```
| ?- male( X ).

no

| ?- man(X), assertz( male(X) ).

X = john ;     % Performed ``assertz( male(john) )''
               % User asks for another answer:

X = fred ;     % Now added ``male(fred)''

X = tom ;      % ...and here ``male(tom)''

no             % Gone through all X: man(X).
               %   Each is now a male.
```

$$\% \; KB_2 = \left\{ \begin{array}{llll} (1) & \texttt{man(john).} & (4) & \texttt{male(john).} \\ (2) & \texttt{man(fred).} & (5) & \texttt{male(fred).} \\ (3) & \texttt{man(tom).} & (6) & \texttt{male(tom).} \end{array} \right\}$$

```
| ?- male( X ).

X = john ;

X = fred ;

X = tom ;

no

| ?-
```

assertzing Rules

% Using $KB_0 = \left\{ \begin{array}{ll} \texttt{(1)} & \texttt{man(john)} \\ \texttt{(2)} & \texttt{man(fred)} \end{array} \right\}$

```
| ?- human( X ).

no                  %     Of course...

| ?- assertz( :-( (human(X)), (man(X)) ) ).

X = _0 _            % Rule ``human(X) :- man(X).'' is now
                    %    added to KB.
yes

| ?- human(X).
                    % Now can use new rule,
                    %    as if user had explicitly entered it...
                    % This rule leads to man(·) facts in KB0
X = john ;

X = fred ;

no

| ?-
```

Comments on `assert`χ

% Using $KB_0 = \left\{ \begin{array}{ll} (1) & \texttt{man(john).} \\ (2) & \texttt{man(fred).} \end{array} \right\}$

```
| ?- asserta( man( tom ) ).
yes
```

% Now $KB_1 = \left\{ \begin{array}{ll} (0) & \texttt{man(tom).} \\ (1) & \texttt{man(john).} \\ (2) & \texttt{man(fred).} \end{array} \right\}$

```
| ?- man( tom ).
yes

| ?- man(X).
X = tom ;      % Notice finds ``tom'' first.
X = john ;     % Then the others, in order.
X = fred ;
no             % Gone through all X: man(X).

| ?- assert( man( george ) ).
yes            % Added man(george) somewhere.
               % Where?   ...depends on implementation...

| ?-
```

Dynamic Side Effects

```
| ?- [user].
| a( X ) :- asserta( a(3) ), fail.
| ^D user con...
yes

| ?- a(X).

no              % First time through, no ‘‘a’’s
                % Try it again, though:
| ?- a(X).

X = 3 ;         % This time, finds the ‘‘a’’ value just asserted

no

| ?- [user].
| e(X) :- asserta( :-( (e(X)), (f(X)))), fail.
| f(10).
| ^D user con...
yes

| ?- e(Z).
no              % Although it failed, had side-effect:
                % added new rule:  ‘‘e(X) :- f(X).’’
| ?- e(Z).
                % Can now use new rule,
                %   to reduce goal to f(Z), then ...
Z = 10 _

yes

| ?-
```

Dynamic Side Effects – II

```
| ?- [user].
| b(X) :- asserta( :-( (c(X)), (d(X)))), fail.
| b(X) :- c(X).
| d(4).
| ^D user con...
yes

| ?- b(Y).
Y = 4 ;       % Using first rule, ``proves'' asserta( ...)
              %   adding new rule:  ``c(X) :- d(X).''.
              % ``b(X) :- ...''  rule fails (2nd clause),
              %   so tries 2nd rule:  ``b(X) :- c(X).''.
              % Reduces to c(Y) subgoal, then uses
              %   NEW RULE to reduce to d(Y) subgoal,
              %  leading to ``Y=4'' answer!

no
              % No other answer now.  But if we try again:

| ?- b(Y).
Y = 4 ;       % Following path described above...but
Y = 4 ;       % Now have the rule ``c(X) :- d(X).'' twice!
              % Leading to answer twice!
no

| ?-
```

Examples of `retract`

$$\% \text{ Using } \quad KB_3 \; = \; \left\{ \begin{array}{ll} (1) & \texttt{married(john).} \\ (2) & \texttt{married(fred).} \\ (3) & \texttt{married(fred,sue).} \\ (4) & \texttt{married(mark,jane).} \end{array} \right\}$$

```
| ?- retract( married(fred,sue) )

yes              % Removed ``married(fred,sue)'', leaving:
```

$$\% \; KB_4 \; = \; \left\{ \begin{array}{ll} (1) & \texttt{married(john).} \\ (2) & \texttt{married(fred).} \\ (4) & \texttt{married(mark,jane).} \end{array} \right\}$$

```
| ?- retract( married(X) )

X = john _       % Prolog removes ``married(john)''.
                 % If user REJECTs (typing ``;''),
                 %   would find and remove other ``married(Q)''s.

yes
```

$$\% \; KB_5 \; = \; \left\{ \begin{array}{ll} (2) & \texttt{married(fred).} \\ (4) & \texttt{married(mark,jane).} \end{array} \right\}$$

```
| ?- retract( married(_) )
                 % Does not ask user about binding;
                 %   just removes first one.
yes
```

$$\% \; KB_6 \; = \; \{ \; (4) \quad \texttt{married(mark,jane).} \; \}$$

```
| ?- retract( foo(_) )

no               % Fails if cannot do removal.
```

Examples of `abolish`

% Using $KB_3 = \left\{ \begin{array}{ll} (1) & \texttt{married(john).} \\ (2) & \texttt{married(fred).} \\ (3) & \texttt{married(fred,sue).} \\ (4) & \texttt{married(mark,jane).} \end{array} \right\}$

```
| ?- married(X).
X = john ;
X = fred ;
no

| ?- abolish(married,1).
                  % Wipes out (1), (2)
yes

| ?- married(X).
no                % No more unary married relations.

| ?- abolish(married,2).
                  % Wipes out (3), (4)
yes

| ?- married(X,Y).
no                % No more binary married relations.

| ?- abolish(married,3).
                  % No effect --- but does succeed.
yes

| ?- abolish(married,X).
                  % ERROR --- both args must be constants.
```

Recall $\boxed{\texttt{retract(married(\_))}}$ only remove 1 clause

First time: wipes out only *(1)*, leaving *(2)*.

Examples of Side Effects

- $$\left\{\begin{array}{lll} \text{mar(fred,judy).} & \text{mar(fred,sue).} & \text{mar(fred,flo).} \\ \text{mar(mark,fran).} & \text{mar(mark,judy).} & \\ \text{mar(derek,june).} & & \\ \multicolumn{3}{l}{\text{big(X) :- mar(X,Y), mar(X,Z), not(=(Y,Z)).}} \\ \multicolumn{3}{l}{\text{big(X) :- mar(Y,X), mar(Z,X), not(=(Y,Z)).}} \end{array}\right\}$$

- $\boxed{\texttt{big(X)}}$ leads to

  `X/fred`, 6 times

  `X/mark`, 2 times

  `X/judy`, 2 times

- Need side effect, to avoid double-counting:

```
bigamist(X) :-  big(X),
                not( injail(X) ),
                assert( injail(X) ).
```

Side Effects (con't)

```
                % Using KB shown earlier...
| ?- bigamist(X)
X = fred ;
X = mark ;
X = judy ;
no              % Notice each answer only ONCE!
                % KB now includes three injail statments:

| ?- injail(Z)
Z = fred ;
Z = mark ;
Z = judy ;
no
                % Problem:  no jailbird can be bigamist!
| ?- bigamist(X)
no
                % ...as everyone is in jail!  ...
                % Solution?  Set everyone free:
| ?- abolish(injail,1)
yes

| ?- bigamist(X)
X = fred ;
X = mark ;
X = judy ;
no

| ?-
```

Retract Set of Clauses

- To remove *all* marriages to a bigamist, use

```
divorse(X) :-  big(X),
               retractall( mar(X,_) ),
               retractall( mar(_,X) ).
```

- What is `retractall`?

  ¬ `retract`, which only removes 1^{st} illegal marriage
  ¬ `abolish`, which removes *all* marriage (legal or not)

- Def'n

```
retractall(X) :-  retract(X), fail.
retractall(_).
```

- Now, $\boxed{\texttt{retractall(mar(fred,\_))}}$
  removes all of `fred`'s marriages.

Program as Data: `clause`

$\boxed{\texttt{clause(LHS, RHS)}}$

finds KB clauses of form "`LHS :- RHS`".

$$KB_A = \left\{ \begin{array}{l} \text{loves(john, mary).} \\ \text{loves(ellen, ted).} \\ \text{loves(ted, Y) :- loves(Y, mary).} \end{array} \right\}$$

```
| ?- clause( loves(A,B), RHS ).

A = john
B = mary
RHS = true ;

A = ellen
B = ted
RHS = true ;

A = ted
B = _1
RHS = loves(_1,mary) ;

no
```

Views "`loves(john, mary).`" as
"`loves(john, mary) :- true.`" .

(Relation-Symbol of `LHS` must be *constant*.)

Executing Data

"`=..`" operator translates list into ⟨`functor`⟩

"`call`" executes constructed goal

$$KB_A = \left\{ \begin{array}{l} \text{loves(john, mary).} \\ \text{loves(ellen, ted).} \\ \text{loves(ted, Y) :- loves(Y, mary).} \end{array} \right\}$$

```
| ?- =..( X, [loves, john, mary] ).

X = loves(john, mary) ;
                  % Notice X's binding is executable clause!
no
                  % What if clause has a variable?

| ?- =..( X, [loves, john, Y] ).

X = loves(john, _1)
Y = _1 ;          % Y associated with (unique) variable.

no
                  % Now to execute this clause:

| ?- =..( X, [loves, john, Y] ),  call(X).

X = loves(john,mary)
Y = mary ;

no

| ?-
```

Utilities

- `setof(X, P, S)`

  `S` is bound to the set of values of `X`

  for which proposition `P` holds.

- `bagof(X, P, S)`

  like **`setof`**, but allows duplicates.

- `=(X,Y)` vs `=:=(X,Y)` vs `is(X,Y)`

  Structural vs Arithmetic Equality, and Arithmetic assignment.

- `op(...)`

  Used to simplify user input

setof Operator

$$KB_B = \left\{ \begin{array}{l} \text{likes(tom, beer).} \\ \text{likes(dick, beer).} \\ \text{likes(harry, beer).} \\ \text{likes(bill, cider).} \\ \text{likes(jan, cider).} \\ \text{likes(tom, cider).} \end{array} \right\}$$

```
| ?- setof( X, likes(X,beer), S ).

X = _0
S = [dick,harry,tom] ;
          % Notice all such X's are bundled together.
          % Binding for X: just unique variable.
no

| ?- setof( lush(X), likes(X,beer), S ).

X = _0
S = [lush(dick),lush(harry),lush(tom)] ;
          % S is list, each element is instance of first form
          % with appropriate value of each variable.
no

| ?-
```

`setof` Operator — (con't)

$$KB_B = \left\{ \begin{array}{l} \text{likes(tom, beer).} \\ \text{likes(dick, beer).} \\ \text{likes(harry, beer).} \\ \text{likes(bill, cider).} \\ \text{likes(jan, cider).} \\ \text{likes(tom, cider).} \end{array} \right\}$$

```
| ?- setof( X, likes(X,Y), S ).
          % Can have > 1 variables in 2nd arg
X = _0
Y = beer
S = [dick,harry,tom] ;
          % First, values for Y = beer

X = _0
Y = cider
S = [bill,jan,tom] ;
          % Then values for Y = cider

no

| ?- setof( (X,Y), likes(X,Y), S ).
          % First arg can be any form
          %   with any number of variables
X = _0
Y = _1
S = [(bill,cider),(dick,beer),(harry,beer),(jan,cider),
          (tom,beer),(tom,cider)] ;

no
```

setof Variants

```
| ?- setof((Y,S), setof(X, likes(X,Y), S), SS)

Y = _0
S = _1
X = _6
SS = [(beer,[dick,harry,tom]),(cider,[bill,jan,tom])] ;
no
                  % Use of Existentials:  ^

| ?- setof(X, Y^likes(X,Y), S)

X = _0
Y = _1
S = [bill,dick,harry,jan,tom] ;

no

                  % bagof(X, P, S) is like setof,
                  %   but does NOT eliminate duplicates.

| ?- bagof(X, Y^likes(X,Y), S)

X = _0
Y = _1
S = [tom,dick,harry,bill,jan,tom] ;

no

| ?-
```

`=` vs `=:=` vs `is`

- `=(17, +(8,9) )` fails

  `=(N, +(8,9) )` returns {`N/+(8,9)`}

  `=( +(X,9), +(8,9) )` returns {`X/8)`}

  `=( +(16,1), +(8,9) )` fails.

- `=:=(17, +(8,9) )` succeeds.

  `=:=(N, +(8,9) )` gives syntax error

  `=:=( +(X,9), +(8,9) )` gives syntax error

  `=:=( +(16,1), +(8,9) )` succeeds.

- `is(17, +(8,9) )` succeeds.

  `is(N, +(8,9) )` returns {`X/17`}

  `is(+(8,9), N)` gives syntax error

  `is( +(X,9), +(8,9) )` fails

  `is( +(16,1), +(8,9) )` succeeds.

Simpler Expressions: `op`

- Can write

  `N is 8+9` rather than `is(N,+(8,9))`

- Implemented using "`op`"

  (see manual)

- In fact:

  "`:-`", "`;`", ...