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 \( KB_0 = \{ (1) \text{ man(john) }, (2) \text{ man(fred) } \} \)

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

?- assertz( man(tom) )
yes

% Form \( KB_1 = \{ (1) \text{ man(john) }, (2) \text{ man(fred) }, (3) \text{ man(tom) } \} \)

?- man(X).
X = john ;
X = fred ;
X = tom ;
no
assertz (con’t)

% Using $KB_1 = \{ (1) \text{ man(john)} , (2) \text{ man(fred)} , (3) \text{ man(tom)}. \}$

| ?- 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.

% $KB_2 = \{ (1) \text{ man(john)}. \quad (4) \text{ male(john)}. \}$
  \{ (2) \text{ man(fred)}. \quad (5) \text{ male(fred)}. \}
  \{ (3) \text{ man(tom)}. \quad (6) \text{ male(tom)}. \}

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

| ?-
asserting Rules

% Using \( KB_0 = \{ (1) \text{man(john)} \)  \\
\( (2) \text{man(fred)} \) \}

| ?- \text{human(X)}.  \\
no % Of course...

| ?- \text{assertz}((\text{human(X)}), (\text{man(X)})).  \\
X = _0 _ % Rule ‘‘human(X) :- man(X).’’ is now  \\
% added to \( KB \).

yes

| ?- \text{human(X)}.  \\
% Now can use new rule,
% as if user had explicitly entered it...
% This rule leads to man(\cdot) facts in \( KB_0 \)

X = john ;  \\
X = fred ;  \\
no

| ?-
Comments on assert \( \chi \)

\[
\% \text{Using } K B_0 = \left\{ \begin{array}{l}
    (1) \text{ man(john)}. \\
    (2) \text{ man(fred)}. 
\end{array} \right. \\
\%
\]

| :- asserta( man( tom ) ).
| yes

\[
\% \text{Now } K B_1 = \left\{ \begin{array}{l}
    (0) \text{ man(tom)}. \\
    (1) \text{ man(john)}. \\
    (2) \text{ man(fred)}. 
\end{array} \right. \\
\%
\]

| :- 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’
| % 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

% Using $KB_3 = \begin{cases} 
(1) \text{ married(john).} \\
(2) \text{ married(fred).} \\
(3) \text{ married(fred,sue).} \\
(4) \text{ married(mark,jane).} 
\end{cases}$

| ?- retract( married(fred,sue) ) |
yes | % Removed ‘‘married(fred,sue)’’, leaving:

% $KB_4 = \begin{cases} 
(1) \text{ married(john).} \\
(2) \text{ married(fred).} \\
(4) \text{ married(mark,jane).} 
\end{cases}$

| ?- retract( married(X) ) |
X = john _ | % Prolog removes ‘‘married(john)’’.  
% If user REJECTs (typing ‘‘_’’), 
% would find and remove other ‘‘married(Q)’’s.

yes

% $KB_5 = \begin{cases} 
(2) \text{ married(fred).} \\
(4) \text{ married(mark,jane).} 
\end{cases}$

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

yes

% $KB_6 = \{ (4) \text{ married(mark,jane).} \}$

| ?- retract( foo(_) ) |
no | % Fails if cannot do removal.
Examples of abolish

% Using $KB_3 = \{ (1) \text{ married(john).} \\
(2) \text{ married(fred).} \\
(3) \text{ married(fred,sue).} \\
(4) \text{ married(mark,jane).} \}

| ?- \text{married}(X). |
X = john ; |
X = fred ; |
no

| ?- \text{abolish(married,1)}. |
% Wipes out (1), (2)
yes

| ?- \text{married}(X). |
no % No more unary married relations.

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

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

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

| ?- \text{abolish(married,X)}. |
% ERROR --- both args must be constants.
Recall \( \text{retract(married(_))} \) only remove 1 clause
First time: wipes out only (1), leaving (2).
Examples of Side Effects

\[
\begin{align*}
\text{mar}(\text{fred}, \text{judy}). & \quad \text{mar}(\text{fred}, \text{sue}). & \quad \text{mar}(\text{fred}, \text{flo}). \\
\text{mar}(\text{mark}, \text{fran}). & \quad \text{mar}(\text{mark}, \text{judy}). \\
\text{mar}(\text{derek}, \text{june}). & \\
\text{big}(X) & : \text{not(}\text{=} (Y, Z)\text{)}. \\
\text{big}(X) & : \text{not(}\text{=} (Y, Z)\text{)}. \\
\end{align*}
\]

- \begin{boxed}{big(X)}\end{boxed} leads to
  - X/fred, 6 times
  - X/mark, 2 times
  - X/judy, 2 times

- Need side effect, to avoid double-counting:
  \[
  \text{bigamist}(X) : - \quad \text{big}(X), \\
  \text{not(}\text{injail}(X)\text{)}, \\
  \text{assert(}\text{injail}(X)\text{)}.
  \]
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 in jail statements:

| :- 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
  \[
  \text{divorce}(X) :- \quad \text{big}(X), \quad \\
  \quad \text{retractall}( \text{mar}(X,\_)), \quad \\
  \quad \text{retractall}( \text{mar}(\_,X)).
  \]

- What is retractall?
  \[
  \neg \text{retract}, \quad \text{which only removes 1}^{st} \text{ illegal marriage} \\
  \neg \text{abolish}, \quad \text{which removes all marriage (legal or not)}
  \]

- Def’n
  \[
  \text{retractall}(X) :- \quad \text{retract}(X), \text{fail.} \quad \\
  \quad \text{retractall}(\_).
  \]

- Now, \boxed{\text{retractall}( \text{mar}(\text{fred},\_))}\n  removes all of \text{fred’s marriages.}
Program as Data: clause

\[ \text{clause( LHS, RHS )} \]
finds KB clauses of form \( \text{"LHS :- RHS"} \).

\[ KB_A = \begin{cases} 
\text{loves(john, mary).} \\
\text{loves(ellen, ted).} \\
\text{loves(ted, Y) :- loves(Y, mary).} 
\end{cases} \]

\text{\textbackslash ?- 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 \text{“loves(john, mary).”} as
\text{“loves(john, mary) :- true.”} .

( Relation-Symbol of LHS must be constant. )
Executing Data

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

"call" executes constructed goal

KB_A = \{ 
  loves(john, mary).
  loves(ellen, ted).
  loves(ted, Y) :- loves(Y, mary).
\}


\begin{align*}
X &= \text{loves(john, mary);} \\
   &\quad \% \text{Notice X's binding is executable clause!}
\end{align*}

no

\begin{align*}
\% \text{What if clause has a variable?}
\text{X} &= \text{loves(john, } _1) \\
\text{Y} &= _1; \\
   &\quad \% \text{Y associated with (unique) variable.}
\end{align*}

no

\begin{align*}
\% \text{Now to execute this clause:}
|?- &=..( X, [loves, john, Y] ), \text{ call}(X).
\text{X} &= \text{loves(john,mary)} \\
\text{Y} &= \text{mary ;}
\end{align*}

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
The setof operator is used to generate a set of all elements that satisfy a given predicate. For example, consider the predicate `likes` where `likes(tom, beer).` and other similar facts are defined. Given a set `KB_B` defined as:

\[ KB_B = \{ \text{likes(tom, beer).}, \text{likes(dick, beer).}, \text{likes(harry, beer).}, \text{likes(bill, cider).}, \text{likes(jan, cider).}, \text{likes(tom, cider).} \} \]

The setof operator can be used to generate a list of variables that satisfy the predicate. Here are a few examples:

1. `?- setof( X, likes(X,beer), S ).`
   - Variables `X` are bound to elements in the set based on the predicate.
   - Result: `X = _0` and `S = [dick, harry, tom]`.
   - Comment: Notice all such `X`'s are bundled together. Binding for `X` is just a unique variable.

2. `?- setof( lush(X), likes(X,beer), S ).`
   - Variables `X` are bound to elements in the set based on the predicate.
   - Result: `X = _0` and `S = [lush(dick), lush(harry), lush(tom)]`.
   - Comment: `S` is list, each element is instance of first form with appropriate value of each variable.

3. `?-`
**setof Operator — (con’t)**

\[ KB_B = \{ \begin{align*} 
& \text{likes(tom, beer).} \\
& \text{likes(dick, beer).} \\
& \text{likes(harry, beer).} \\
& \text{likes(bill, cider).} \\
& \text{likes(jan, cider).} \\
& \text{likes(tom, cider).} 
\end{align*} \] 

<table>
<thead>
<tr>
<th>?- setof( X, likes(X,Y), S ).</th>
</tr>
</thead>
<tbody>
<tr>
<td>% Can have &gt; 1 variables in 2nd arg</td>
</tr>
<tr>
<td>X = _0</td>
</tr>
<tr>
<td>Y = beer</td>
</tr>
<tr>
<td>S = [dick,harry,tom] ;</td>
</tr>
<tr>
<td>% First, values for Y = beer</td>
</tr>
<tr>
<td>X = _0</td>
</tr>
<tr>
<td>Y = cider</td>
</tr>
<tr>
<td>S = [bill,jan,tom] ;</td>
</tr>
<tr>
<td>% Then values for Y = cider</td>
</tr>
<tr>
<td>no</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>?- setof( (X,Y), likes(X,Y), S ).</th>
</tr>
</thead>
<tbody>
<tr>
<td>% First arg can be any form</td>
</tr>
<tr>
<td>% with any number of variables</td>
</tr>
<tr>
<td>X = _0</td>
</tr>
<tr>
<td>Y = _1</td>
</tr>
<tr>
<td>S = [(bill,cider),(dick,beer),(harry,beer),(jan,cider), (tom,beer),(tom,cider)] ;</td>
</tr>
<tr>
<td>no</td>
</tr>
</tbody>
</table>
set of 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

?-
\[ = \text{ VS } =:= \text{ VS } \text{ is} \]

- \[ = (17, +(8,9)) \] 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.

- \[ \text{is}(17, +(8,9)) \] succeeds.
- \[ \text{is}(N, +(8,9)) \] returns \{X/17\}
- \[ \text{is}(+(8,9), N) \] gives syntax error
- \[ \text{is}(+(X,9), +(8,9)) \] fails
- \[ \text{is}(+(16,1), +(8,9)) \] succeeds.
Simpler Expressions: \( \text{op} \)

- Can write
  
  \[
  N \text{ is } 8+9 \text{ rather than } \text{is}(N,+(8,9))
  \]

- Implemented using “\( \text{op} \)”
  (see manual)

- In fact:
  
  “\(-\)” , “;” , . . .