Principles of Programming Languages
Lecture 6

Wael Aboulsaadat

wael@cs.toronto.edu

http://portal.utoronto.ca/

Acknowledgment: parts of these slides are based on material by Diane Horton & Eric Joanis @ UoT

References: Scheme by Dybvig
Concepts of PL by Sebesta
Prog. in Prolog by Clocksin and Mellish
PL Concepts and Constructs by Sethi
ML for the Working Prog. By Paulson
PL Pragmatics by Scott

Prolog: recursion – examples

- **Appending lists:**
  - Declarative Semantics:

    Appending an empty list to a non-empty list is the non-empty list else work on one list by removing its elements and adding it to the other list.
  - Prolog

Prolog: recursion – examples

- **Appending lists:**
  - Declarative Semantics:

    Appending an empty list to a non-empty list is the non-empty list else work on one list by removing its elements and adding it to the other list.
  - Prolog

```
append([],L,L).
```

Prolog: recursion – examples

- **Appending lists:**
  - Declarative Semantics:

    Appending an empty list to a non-empty list is the non-empty list else work on one list by removing its elements and adding it to the other list.
  - Prolog

```
append([],L,L).
append([H | L1], L2, [H | L3]) :- append(L1,L2,L3).
```

```
|?-  append([a,b,c],[1,2,3], R).
 R= [a,b,c,1,2,3]
```

Prolog: recursion – examples

- **Appending lists:**

```
append([],L,L).
append([H | L1], L2, [H | L3]) :- append(L1,L2,L3).
```

```
?- append([a,b,c],[1,2,3], R).
  /                        \
†                   R = [a|L0]
                   ?- append([b,c],[1,2,3],L0)
```

Prolog: recursion – examples

- **Appending lists:**

```
append([],L,L).
append([H | L1], L2, [H | L3]) :- append(L1,L2,L3).
```

```
?- append([a,b,c],[1,2,3], R).
  /                          \
†                    R = [a|L0]
                     ?- append([b,c],[1,2,3],L0)
                        /                  \
                     †                     L0=[b|L1]
                                           ?- append([c],[1,2,3],L1)
```

Prolog: recursion – examples

- **Appending lists:**

```
append([],L,L).
append([H | L1], L2, [H | L3]) :- append(L1,L2,L3).
```

```
?- append([a,b,c],[1,2,3], R).
   /                          \
 †                    R = [a|L0]
                      ?- append([b,c],[1,2,3],L0)
                         /                      \
                       †                  L0=[b|L1]
                                          ?- append([c],[1,2,3],L1)
                                             /                   \
                                           †               L1=[c|L2]
                                                           ?- append([],[1,2,3],L2)
```

Prolog: recursion – examples

- **Appending lists:**

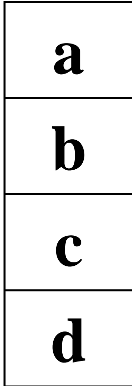
```
append([],L,L).
append([H | L1], L2, [H | L3]) :- append(L1,L2,L3).

?- append([a,b,c],[1,2,3], R).
   /                        \
 †                   R = [a|L0]
                     ?- append([b,c],[1,2,3],L0)
                        /                   \
                      †                L0=[b|L1]
                                       ?- append([c],[1,2,3],L1)
                                          /                  \
                                        †               L1=[c|L2]
                                                        ?- append([],[1,2,3],L2)
                                                           /                  \
                                                     L2=[1,2,3]               †
```

L2=[1,2,3]
L1=[c|L2]=[c,1,2,3]
L0=[b|L1]=[b,c,1,2,3]
R=[a|L0]=[a,b,c,1,2,3]

Prolog: recursion – examples

- **Blocks:**
  - Declarative Semantics:

    Block X is above block Y if X is placed on top of Y, or X is placed on top of some block Z that is above Y.



Prolog: recursion – examples

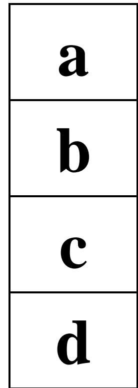
- **Blocks:**
  - Declarative Semantics:

    Block X is above block Y if X is placed on top of Y, or X is placed on top of some block Z that is above Y.
  - Prolog:

```
above(X,Y) :- on(X,Y).                (1)
```

a
b
c
d

Prolog: recursion – examples

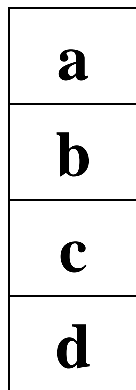


- **Blocks:**
  - Declarative Semantics:

    Block X is above block Y if X is placed on top of Y, or X is placed on top of some block Z that is above Y.
  - Prolog:

```
above(X,Y) :- on(X,Y).                          (1)
above(X,Z) :- above(X,Y), above(Y,Z).  (2)
```

Prolog: recursion – examples

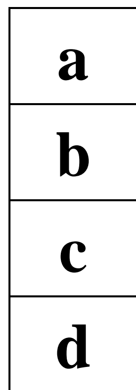


- **Blocks:**
  - Declarative Semantics:

    Block X is above block Y if X is placed on top of Y, or X is placed on top of some block Z that is above Y.
  - Prolog:

```
above(X,Y) :- on(X,Y).                        (1)
above(X,Z) :- above(X,Y), above(Y,Z).  (2)
on(a,b).        (3)
on(b,c).        (4)
on(c,d).        (5)
```

Prolog: recursion – examples



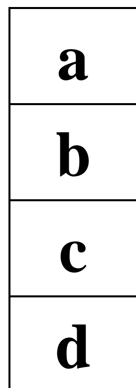
- **Blocks:**
  - Declarative Semantics:

    Block X is above block Y if X is placed on top of Y, or X is placed on top of some block Z that is above Y.
  - Prolog:

```
above(X,Y) :- on(X,Y).                    (1)
above(X,Z) :- above(X,Y), above(Y,Z).     (2)
on(a,b).     (3)
on(b,c).     (4)
on(c,d).     (5)
|?- above(a,b).


|? above(a,d).
```

Prolog: recursion – examples

- **Blocks:**



  – Declarative Semantics:

    Block X is above block Y if X is placed on top of Y, or X is placed on top of some block Z that is above Y.

  – Prolog:

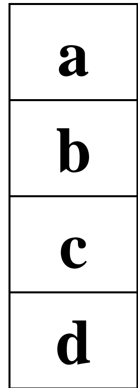
```
above(X,Y) :- on(X,Y).                     (1)
above(X,Z) :- above(X,Y), above(Y,Z).      (2)
on(a,b).     (3)
on(b,c).     (4)
on(c,d).     (5)
|?- above(a,b).
Yes
|? above(a,d).
Yes
|? above(c,a).
```

above(a,b) → on(a,b)

above(a,d) → above (a,Y) →above(a,b) → above(b,d) → above(b,Y) → above(b,c) → above(c,d) → true

Prolog: recursion – examples

- **Blocks:**



  - Declarative Semantics:

    Block X is above block Y if X is placed on top of Y, or X is placed on top of some block Z that is above Y.

  - Prolog:

above(X,Y) :- on(X,Y). **(1)**

above(X,Z) :- above(X,Y), above(Y,Z). **(2)**

on(a,b). **(3)**

on(b,c). **(4)**

on(c,d). **(5)**

|?- above(a,b).

Yes

|? above(a,d).

Yes

|? above(c,a).

Infinite recursion!

Prolog: recursion – examples

- **Blocks:**

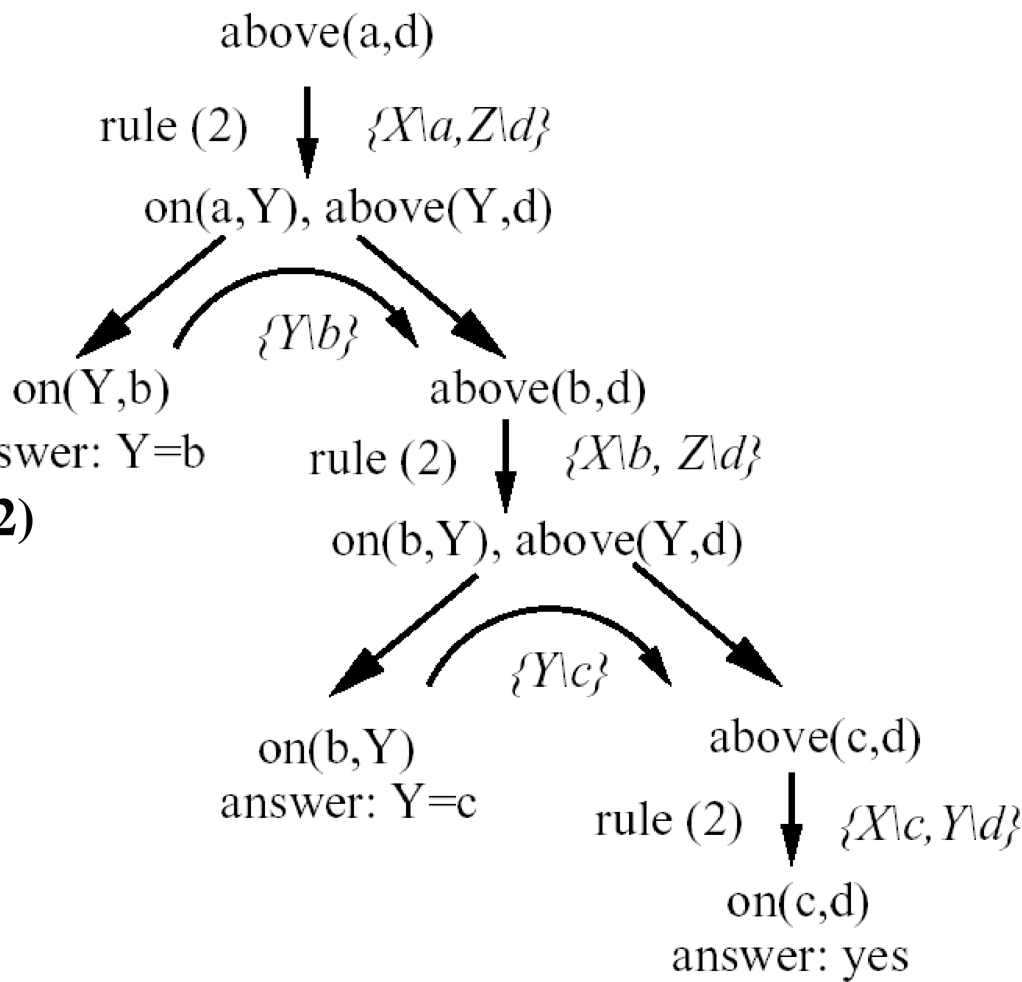
a
b
c
d

  - Declarative Semantics:

    Block X is above block Y if X is placed on top of Y, or X is placed on top of some

    block Z that is above Y.

  - Prolog:




```
% Second attempt
above(X,Y) :- on(X,Y).            (1)
above(X,Z) :- on(X,Y), above(Y,Z). (2)
on(a,b).     (3)
on(b,c).     (4)
on(c,d).     (5)
|?- above(a,d).
Yes
```

Prolog: recursion – examples

- **Blocks:**
  - Declarative Semantics:

    Block X is above block Y if X is placed on top of Y, or X is placed on top of some

    block Z that is above Y.
  - Prolog:

a
b
c
d

```
% Second attempt
above(X,Y) :- on(X,Y).              (1)
above(X,Z) :- on(X,Y), above(Y,Z).  (2)
on(a,b).      (3)
on(b,c).      (4)
on(c,d).      (5)
|?- above(c,a).
false
```

above(c,a) → on(c,Y) → on(c,d) → above(d,a) → on(d,Y)→ false

Prolog: recursion – examples

- **Note that sometimes changing the order of rules and/or rule premises can cause problems for Prolog**

- **Example:** above(X,Z) :- on(X,Y), above(Y,Z). **(1)**

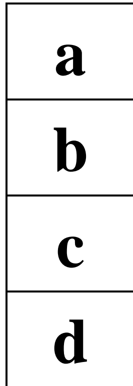
  above(X,Y) :- on(X,Y). **(2)**

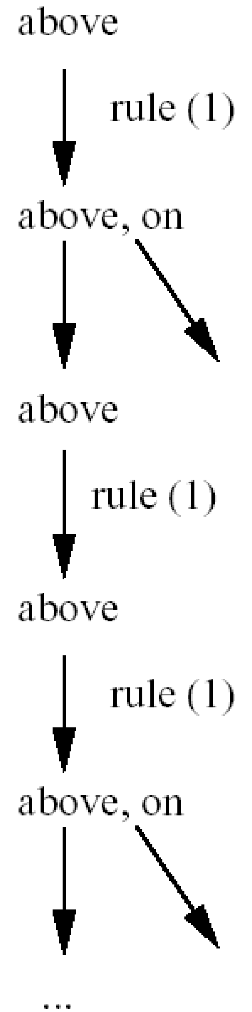
  on(a,b). **(3)**

  on(b,c). **(4)**

  on(c,d). **(5)**

  |?- above(a,d).





Prolog: recursion – examples

- **What to do about infinite recursion?**
  - Rewrite the rules and facts *(most widely used technique)*
    - ***ALWAYS ATTEMPT TO INSERT FACT INTO RHS.***
    - ***PUT SIMPLER RULE FIRST.***
  - Use ! to stop the unification

Prolog: cut !

- **The cut is written as ! and is inserted between goals as a pseudo-goal,** ***for example:***

```
head(X) :- goal1a, goal1b,!,goal1c,goal1d.      %rule 1
head(X) :- goal2a, goal2b,goal2c.               %rule 2
head(X) :- goal3a,goal3b,goal3c,goal3d.         %rule 3
| ?- head(someconstant).                        %query
```

Prolog: cut !

- **The cut is written as ! and is inserted between goals as a pseudo-goal, *for example:***

```
head(X) :- goal1a, goal1b,!,goal1c,goal1d.      %rule 1
head(X) :- goal2a, goal2b,goal2c.               %rule 2
head(X) :- goal3a,goal3b,goal3c,goal3d.         %rule 3
| ?- head(someconstant).                        %query
```

% Without the !, interpreter will try to unify the query with the 3 rules
% because their heads match the query and it will stop only after the
% 3rd. ! is a signal to the interpreter not to try unification on rules 2&3
% after it is done with rule 1 if rule succeeds.

- **Is the location of the !, in the same clause, significant?**
  - Yes, consider above example, placing ! after goal1b means that only satisfying *goal1a* and *goal1b* is sufficient for your solution and you do not need to unify on any of the goals in rules 2&3.
  - Note that this does not mean that rule1 will succeed, because that is dependent on *goal1c* and *goal1d* being true as well.

Prolog: cut ! – when to use?

- **Common uses of the !:**
  - Tell the Prolog system that it has found the *right rule* for a particular goal:

    ***If you get this far, you have picked the correct rule for this goal*.**

  - Tell the Prolog system to fail a particular goal immediately without trying for alternate solutions:

    ***If you get to here, you should stop trying to satisfy the goal*.**

  - Terminate the generation of alternative solutions:

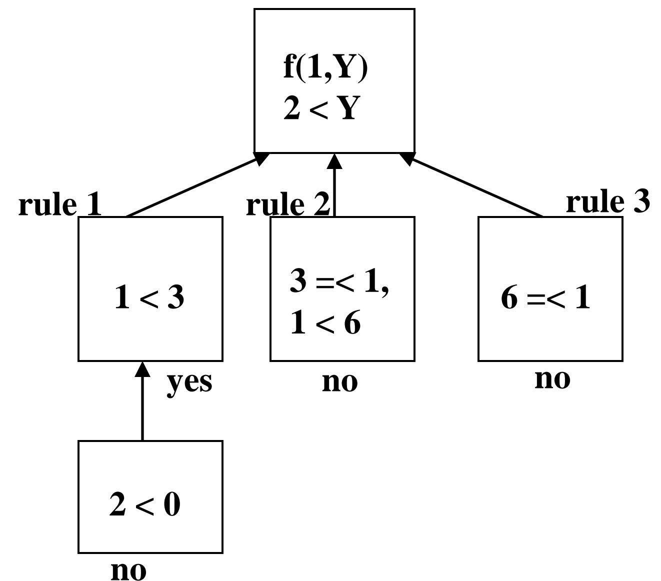
    ***If you get to here, you have found the only solution to this problem, no point in looking for alternatives*.**

Prolog: cut ! - examples

- **Double-step function:**

if X < 3 then Y = 0
if 3 =< X and X < 6 then Y = 2
if 6 =< X then Y = 4

```
% In Prolog
f(X,0) :- X < 3.             %rule 1
f(X,2) :- 3 =< X, X < 6.     %rule 2
f(X,4) :- 6 =< X.            %rule 3

| ?- f(4,Y).                 %query
Y=2

| ?- f(1,Y),2<Y.             %query
no
```



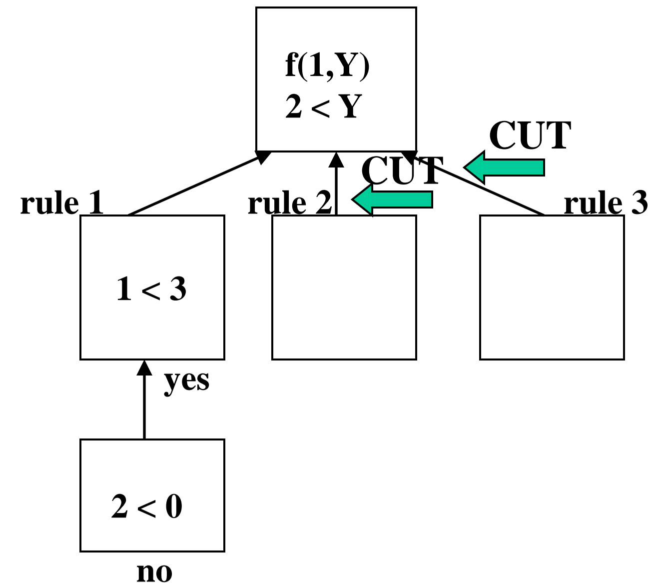
Prolog: cut ! – example

- **Double-step function - cont’d:**

if X < 3	then Y = 0
if 3 =<X and X < 6	then Y = 2
if 6 =< X	then Y = 4

What do we know about this function that Prolog doesn’t?

```
% same relations with !
f(X,0) :- X < 3,!.            %rule 1
f(X,2) :- 3 =< X, X < 6,!.    %rule 2
f(X,4) :- 6 =< X.             %rule 3

| ?- f(1,Y),2<Y.              %query
no
```

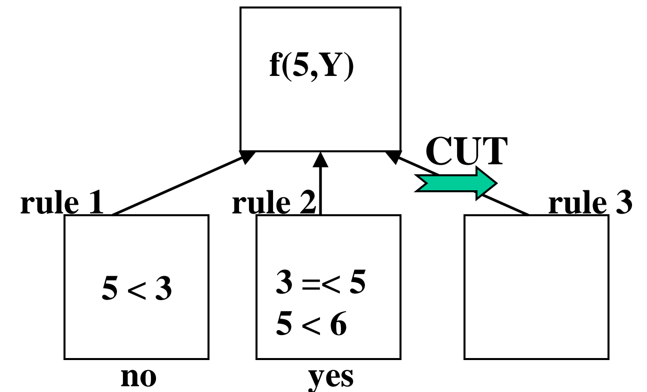


In this example, we changed the procedural meaning of the program, but not the declarative meaning

Prolog: cut ! – example

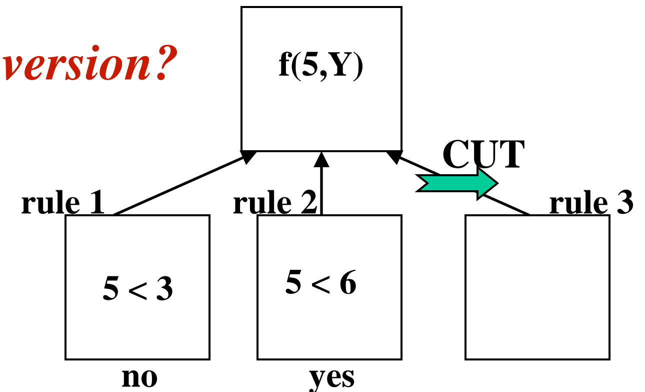
- **Double-step cont'd:**

```
% same relations with !
f(X,0) :- X < 3,!.              %rule1
f(X,2) :- 3 =< X, X < 6,!.      %rule2
f(X,4) :- 6 =< X.               %rule3
| ?- f(5,Y).                    %query
Y = 2
```



Can we come up with a more efficient version?

```
f(X,0) :- X < 3,!.              %rule1
f(X,2) :- X < 6,!.              %rule2
f(X,4).                         %rule3
| ?- f(5,Y).                    %query
Y=2
```




```
% What if we removed the cuts?
f(X,0) :- X < 3.                %rule1
f(X,2) :- X < 6.                %rule2
f(X,4).                         %rule3
```

```
| ?- f(1,Y).% query
Y = 0;          % right answer
Y = 2;          % wrong answer,why?
Y = 4;          % wrong answer,why?
```

Here, we changed the procedural and also the declarative meaning

Prolog: cut ! classification

- **Green cuts:**
  - Affect procedural meaning of program but has no effect on the declarative meaning.
  - Do not affect readability of programs.
  - Used mainly to avoid wasted computations.

- **Red cuts:**
  - Affect declarative meaning of program as well as procedural meaning.
  - Affect readability:
    - Similar to unconditional jump (goto) in imperative PLs ☹
  - Used to avoid wasted computations and also introduce semantics
  - If not used cautiously, can affect result in arbitrary way.

Prolog: cut ! - examples

- **Member function:**

```
member(Element, [Element | _]).
member(Element, [Head| Rest]) :- member(Element, Rest).

% What's the problem in the above function?

member(Element, [Element | _]) :- !.
```

Prolog: cut ! - examples

- **Member function:**

```
member(Element, [Element | _]).
member(Element, [Head| Rest]) :- member(Element, Rest).

% What's the problem in the above function?

member(Element, [Element | _]) :- !.
```

- **Sum function:**
  - Without !:

```
sum_to(1,1).
sum_to(N, Res) :- N1 is N-1, sum_to(N1,TRes), Res is TRes + N.
| ?- sum_to(1,X).
X=1;
%Infinite loop…………………
```

  - With !:

```
sum_to(1,1) :- !.
sum_to(N, Res) :- N1 is N-1, sum_to(N1,TRes), Res is TRes + N.
| ?- sum_to(1,X).
X=1;
```

Prolog: cut ! Pros & Cons

- **Pros:**
  - More efficient: interpreter will not process unnecessary branches in the unification tree
  - Save memory: interpreter will not allocate space for tree nodes that will not be unified.

- **Cons:**
  - Creates side effects that change the way backtracking works *(not really logic programming…)*
  - Makes the place markers of certain goals inaccessible.

Prolog: complex types - structures

- **Recall: what's a function term?**

  functor(some-parameters) e.g. woman(marry)

- **We can construct complex data structures using nested *function terms* and *lists*.**
  - Represents a statement about the world

- **Example 2:**
  - A family consist of 2 persons, and 0 or more children. Each person is either employed for some salary or unemployed.

```
family(person(First-name,Last-name,date(Day,Month,Year),works(Company,Salary)),
       person(First-name,Last-name,date(Day,Month,Year),works(Company,Salary)),
       [person(First-name,Last-name,date(Day,Month,Year),unemployed),
        person(First-name,Last-name,date(Day,Month,Year),unemployed)]).
```

Prolog: structures – example 2

- Family database:

```
family(
person(tom,fox,date(7,may,1950),works(cbc,15200)),
person(ann,fox,date(9,may,1951), works(ctv,25700)),
[person(pat,fox,date(5,may,1973),unemployed),
 person(jim,fox,date(5,may,1973),unemployed)]).

%To find if there is a family of three children,
| ?- family(_,_,[_,_,_]).
no

%To find all the fox families
| ?- family(person(X,fox,Y,Z),T,W).
X = tom
Y = date(7,may,1950)
Z = works(cbc,15200)
T = person(ann,fox,date(9,may,1951),works(ctv,25700))
W = [person(pat,fox,date(5,may,1973),unemployed),
     person(jim,fox,date(5,may,1973),unemployed)];
no
```

```
%To find if there is a married woman that
% have at least three children:
| ?-
family(_,person(Name,Surname,_,_),[_,_,_|_]).
no
```

Prolog: structures – example 2

- Family database – cont'd:

```
family(
person(tom,fox,date(7,may,1950),works(cbc,15200)),
person(ann,fox,date(9,may,1951), works(ctv,25700)),
[person(pat,fox,date(5,may,1973),unemployed),
 person(jim,fox,date(5,may,1973),unemployed)]).

% Let us add more useful rules
husband(X) :- family(X,_,_).
wife(X)      :- family(_,X,_).
child(X)      :- family(_,_,Children),
                 member(X,Children).
member(X,[X|L]).
member(X,[Y|L]) :- member(X,L).
exists(Person) :-husband(Person);
                  wife(Person);child(Person).
salary(person(_,_,_,works(_,S)),S).
salary(person(_,_,_,unemployed),0).
dateofbirth(person(_,_,Date,_),Date).
```

```
% Find the names of all the people in database
| ?- exists(person(Name,Surname,_,_)).
Name = tom
Surname = fox;
Name = ann
Surname = fox;
Name = pat
Surname = fox;
Name = jim
Surname = fox;
no

% Find all children born in 1973
| ?- child(X),dateofbirth(X,date(_,_,1973)).
X = person(pat,fox,date(5,may,1973),
    unemployed);
X = person(jim,fox,date(5,may,1973),
     unemployed);
no
```

Prolog: structures – example 2

- Family database – cont'd:

```
% To find the names of unemployed people who were born before 1975
| ?- exists(person(Name,Surname,date(_,_,Year),unemployed)),Year < 1975.
Name = pat
Surname = fox
Year = 1973;
Name = jim
Surname = fox
Year = 1973;
no

% To find people born before 1951 whose salary is less than 80000
| ?- exists(Person),dateofbirth(Person,date(_,_,Year)),Year<1951,
salary(Person,Salary), Salary<80000.
Person = person(tom,fox,date(7,may,1950),works(cbc,15200))
Year = 1950
Salary = 15200;
no
```

Prolog: structures – example 2

- **Family database – cont'd:**

```
% Let us add a rule to add the salaries
total([],0).
total([Person|List],Sum):- salary(Person,S),total(List,Rest),Sum is S + Rest.

% To find the total income of family
| ?- family(Husband,Wife,Children),total([Husband,Wife|Children],Income).
Husband = person(tom,fox,date(7,may,1950),works(cbc,15200))
Wife = person(ann,fox,date(9,may,1951),works(ctv,25700))
Children = [person(pat,fox,date(5,may,1973),unemployed),
            person(jim,fox,date(5,may,1973),unemployed)]
Income = 40900;

% To retrieve the nth child of a family, we need to define how to get the nth element of a list
nth_member(1,[X|L],X).
nth_member(N,[Y|L],X):-N1 is N –1,nth_member(N1,L,X).

% Now, let us define how to get the nth child. Note: I left children clause for you to define
| ?- nthchild(N,Family,Child) :- children(Family,ChildList),nth_member(N,ChildList,Child).
```

Prolog: negation

- **Prolog includes a "not" …**

- **Can say not(P) for any term P**
  - Can't have a negative head, in the body only (head :- body)
  - Can't declare a negative fact

- **Failure as negation:**
  - If you cannot prove something is true, then assume it is false.
  - Prolog's not means the symbol failed to find/prove P
  - The term not(P) succeeds iff P fails.
  - E.g. **a :- b, not c**

    means infer a if b can be inferred and c cannot be inferred.

Prolog: negation – when to use?

- Use **not** to enforce semantics….

- Example:

```
loves(bill,X) :- pretty(X), female(X),
                  not loves(tom,X).
```

i.e., Bill loves any pretty female, unless Tom loves her.

```
loves(tom,X) :- famous(X), female(X),
                 not dead(X).
```

i.e., Tom loves any famous living female.

```
female(marilyn-monroe).   famous(marilyn-monroe).
female(cindy-crawford).   famous(cindy-crawford).
female(martha-stewart).   famous(martha-stewart).
female(girl-next-door).

pretty(marilyn-monroe).   dead(marilyn-monroe).
pretty(cindy-crawford).
pretty(girl-next-door).
```

```
| ?- loves(tom,X).
X = cindy-crawford;
X = martha-stewart;

| ?- loves(bill,X).
X = marilyn-monroe;
X = girl-next-door;
```

Prolog: negation – when to use?

- Using not is not always safe…

- Example:

```
loves(bill,X) :- pretty(X), female(X),
                 not loves(tom,X).
```

i.e., Bill loves any pretty female, unless Tom loves her.

```
loves(tom,X) :- famous(X), female(X),
                not dead(X).
```

i.e., Tom loves any famous living female.

```
female(marilyn-monroe).   famous(marilyn-monroe).
female(cindy-crawford).   famous(cindy-crawford).
female(martha-stewart).   famous(martha-stewart).
female(girl-next-door).

pretty(marilyn-monroe).   dead(marilyn-monroe).
pretty(cindy-crawford).
pretty(girl-next-door).
```

Consider the following rule:

(*)   `hates(tom,X) :- not loves(tom,X).`

This may NOT be what we want, for several reasons:

- The answer is *infinite*, since for any person `p` not mentioned in the database, we cannot infer `loves(tom,p)`, so we must infer `hates(tom,p)`.

  Rule (*) is therefore said to be unsafe.

- The rule does not require `X` to be a person.
  e.g., since we cannot infer
  `loves(tom,hammer)`
  `loves(tom,verbs)`
  `loves(tom,green)`
  `loves(tom,abc)`

we must infer that tom hates all these things.

Prolog: negation – when to use?

- **To avoid these problems, rules with negation should be <u>guarded</u>**

- **Example:**

```
loves(bill,X) :- pretty(X), female(X),
                 not loves(tom,X).
```

i.e., Bill loves any pretty female, unless Tom loves her.

```
loves(tom,X) :- famous(X), female(X),
                not dead(X).
```

i.e., Tom loves any famous living female.

```
female(marilyn-monroe).   famous(marilyn-monroe).
female(cindy-crawford).   famous(cindy-crawford).
female(martha-stewart).   famous(martha-stewart).
female(girl-next-door).

pretty(marilyn-monroe).   dead(marilyn-monroe).
pretty(cindy-crawford).
pretty(girl-next-door).
```

```
hates(tom,X) :- female(x), pretty(X),
                not loves(tom,X).
```

i.e., Tom hates every pretty female whom he does not love.

Here, `female` and `pretty` are called <u>guard literals</u>. They guard against safety problems by binding `X` to specific values in the database.

Prolog: negation & Closed World Assumption

- **Closed World Assumption means that Prolog's world is closed :**
  - Everything in the universe that is true is provable from the facts and rules in the knowledge base. Everything else is false.

Prolog: negation – when to use?

- **Example:**

```
krispie( snap ).
krispie( crackle ).
krispie( pop ).
breakfast(A,B,C) :- krispie(A), krispie(B),
                    krispie(C).

| ?- breakfast(X,Y,Z).
X = snap
Y = snap
Z = snap ;
X = snap
Y = snap
Z = crackle ;
X = snap
Y = snap
Z = pop ;
```

```
krispie( snap ).
krispie( crackle ).
krispie( pop ).
breakfast(A,B,C) :- krispie(A), krispie(B),
                    krispie(C),
                    not(A=B), not(A=C),
                    not(B=C).

| ?- breakfast(X,Y,Z).
X = snap
Y = crackle
Z = pop;
X = snap
Y = pop
Z = crackle;
X = crackle
Y = snap
Z = pop
```

Prolog: negation – cont'd

- **Why is this unsafe?**

```
mother(mary, tyler).
father(scott, tyler).
blue_eyed(tyler).
not_parent(X, Y) :-not(father(X,Y)), not(mother(X,Y)).
```

```
|?- not_parent(joe, tyler).
yes

|?- not_parent(scott, tyler).
no

|?- not_parent(lloyd, scott).
yes

|?- not_parent(X,Y).
no

|?- not_parent(X,Y), blue_eyed(X).
no
```

```
|?- blue_eyed(X), not_parent(X,Y).
X = tyler
Y = ….?
yes
```

Prolog: fail & true predicates

- **That something is not true can be said in Prolog by using a special goal: fail , which *always and immediately* fails. That something is true can be said in Prolog by using the goal true, which *always and immediately* succeeds.**

- **The fail/true are inserted between goals as a pseudo-goal:**

```
head1(X) :- goal1a, goal1b, fail, goal1c.        %rule 1
head2(X) :- true.                                %rule 2
```

Prolog: fail & true predicates

- **Examples:**
  - How to represent "*Mary likes all animals but snakes*"?

```
% If X is a snake then Mary likes X is not ture
% otherwise if X is an animal then Mary likes X
snake(cobra).
cat(persian).
animal(X):-snake(X);cat(X).
likes(mary,X):-snake(X),fail.   % rule 1
likes(mary,X):-animal(X).       % rule 2
! ?- likes(mary,cobra).         % query
yes
% Why did we get a wrong answer?
%  rule 1 failed because of fail but
% Prolog interpreter went on to unify
% with rule 2 which succeeded so we
% got yes
```

```
snake(cobra).
cat(persian).
animal(X):-snake(X);cat(X).
likes(mary,X):-snake(X),!,fail. % rule 1
likes(mary,X):-animal(X).       % rule 2
! ?- likes(mary,cobra).         % query
no
%Why did we get the right answer?
% rule 1 failed because of fail and the !
% told the interpreter not to try rule 2
% because since we have reached
% there(aka to the !), it must be a snake
```

Prolog: fail & true predicates

- **Examples – cont'd:**
  - How to represent "*X and Y are different if they do not match*"?

```
% If X and Y match then different(X,Y) fails
% otherwise different(X,Y) succeeds
different(X,Y) :- X=Y,fail.    %rule 1
different(X,Y) :- true.        %rule 2
| ?- different(5,5).           %query
yes
% Why did we get a wrong answer?
%  rule 1 failed because of fail but
% Prolog interpreter unified with
% rule 2 which succeeded so we got yes.
```

```
different(X,Y) :- X=Y,!, fail. %rule 1
different(X,Y) :- true.        %rule 2
| ?- different(5,5).           %query
no
% Why did we get the right answer?
%  rule 1 failed because of fail and
% the ! told the interpreter not to try
% rule 2.
```

  - How to represent "*not(Goal) relation*"?

```
% if Goal succeeds then not(Goal) fails,
% otherwise not(Goal) succeeds
not(P) :- P,!,fail.
not(P) :- true.
```

Prolog: dynamic programming

- **What is it?**

- **Adding/removing rules:**
  - assert(C). % adds a clause to the database
  - asserta(C). % adds a clause to the beginning of the database
  - assertz(C). % adds a clause to the end of the database
  - retract(C). % removes a clause from the database

- **Examples:**

```
| ?- crisis.
no
| ?- assert(crisis).
yes
| ?- crisis.
yes
| ?- retract(crisis).
yes
| ?- crisis.
no
```

```
fast(ann).  slow(tom).   slow(pat).
| ?- assert( (faster(X,Y) :- fast(X), slow(Y) ) ).
yes
| ?- faster(A,B).
….
| ?- retract( slow(X) ).
 X = tom;
 X = pat;
no
faster( ann,_).
no
```

Prolog: dynamic programming – cont'd

- **Useful applications of *assert* (*improving efficiency*):**
  - Caching a single solution:

```
solve(Problem,Solution) :- …..
| ?- solve(problem1, Solution), asserta(solve (problem1,Solution) ).
```

  - Caching a table of solutions, *for example*:

```
maketable :- L = [0,1,2,3,4,5,6,7,8,9],member(X,L),member(Y,L),Z is X * Y,
             asserta(product(X,Y,Z)).
| ?- maketable.              % create product table and add it to the top
no
| ?- product(A,B,8).         % query to find all A's and B's whose product is 8
A = 1   B = 8;        A = 2   B = 4;
A = 4   B = 2;        A = 8   B = 1;
no
```

- **Excessive/careless use of assert/retract yields programs that are hard to read:**
  - Relations that hold true at one point will not hold true at some other time
  - At different times, the same question receive different answers

Prolog: input/output

- **Reading from input:**
  - Read a term from the user read(X).

- **Writing to output:**
  - output the term X write(X).
  - New line nl.
  - Output N spaces tab(N).
  - Outputting a list:

```
writelist([]).
writelist([X|L]) :- write(X), nl, writelist(L).
```

```
| ?- writelist([12,14]).
12
14
```

  - Outputting a sequence of characters as *

```
bars([]).
bars([N|L]) :- stars(N), nl, bars(L).
stars(N) :- N > 0, write(*), N1 is N-1, stars(N1).
stars(N):- N =< 0.
```

```
| ?- bars([3,4,6,5]).
***
****
******
*****
```

Prolog: input/output

- **Interactive program example:**

```
cube :- write('Next item, please: '),read(X),process(X).
process(stop):- !.
process(N) :- C is N * N * N, write('Cube of '), write(N),
                        write('is '),write(C),nl,cube.
| ?- cube.
Next item, please: 5.
Cube of 5 is 125
Next item, please: 10.
Cube of 10 is 1000
```

Prolog: using write in debugging

- **Example:**

  Rule base:

```
p(X) :- q(X), write(X), r(X).
q(a).   q(b).   q(c).   q(d).   q(e).
r(a).   r(d).
```

  Query: Find `X` such that `p(X)` is true.

  Then Prolog prints:

```
a
X = a
bcd
X = d
e
no
```

Prolog: pros & cons

- **Cons:**
  - Horn clauses have limited expressive power
  - Closed world assumptions *(anything not mentioned is false)*
  - Ordering of clauses change the result
  - Because horn clause is the basic construct, you must program carefully to avoid *infinite loops* and *incorrect negation*.
    - There is no 1-solution-fits-all for these problems…

- **Pros:**
  - Pattern matching
  - Backtracking
  - Unification
  - Rules and goals are also data (*dynamic programming*).
  - The logical model is powerful

Pure Logic Programming vs. Prolog

- **Prolog: deterministic**
  - Expand first rule first
  - Explore first(leftmost) sub-goal first
  - Results may depend on rule and sub-goal ordering.

- **Pure Logic Programming: non-deterministic**
  - Arbitrarily choose rules to expand first
  - Arbitrarily chose sub-goal to explore first
  - Results don't depend on rule and sub-goal ordering

Logic vs. Functional programming

- **Functional programming model:**
  - Main construct: functions
  - Return one particular answer for a given set of inputs

- **Logic/relational programming model:**
  - Main construct: relations
  - Can return many different answers for a given set of inputs.
  - When we run a query, it not only tell us if it is true, but also lists all the situations (that it has found) which make it true.

Both are based on recursion

Logic vs. Imperative Languages

	Pascal, C, Java etc.	Prolog
Program sections	1. Types 2. Procedures 3. Procedure body 4. statements, proc-calls 5. calculations w/functions 6. boolean expressions, <,=,>, and, or, not .... etc	1. Types (or domains) 2. Predicates 3. Clauses 4. predicate-calls 5. calculations w/functions 6. boolean expressions, <,=,>, and, or, not .... etc
Assignment	Yes (x=x+1)	no
Unification	no	yes
Parameter passing	simple	complex
Pattern matching	no	yes
Top-down reasoning & back-tracking	no	yes

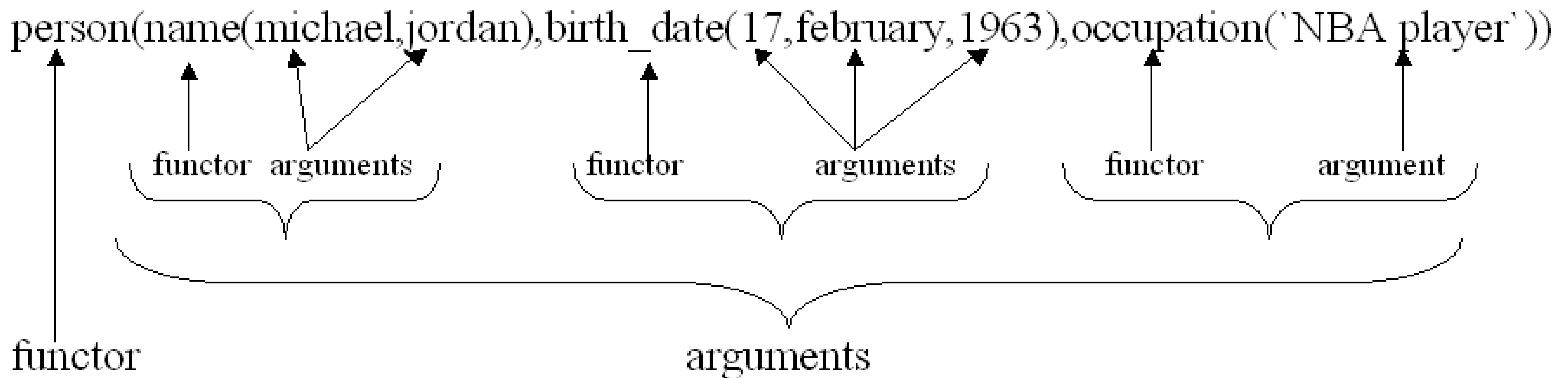
Prolog: complex types - structures

- **Recall: what's a function term?**

  functor(some-parameters) e.g. woman(marry)

- **We can construct complex data structures using nested *function terms*.**
  - Represents a statement about the world

- **Example:**
  - A person has; name: first name, last name - birth date: day, month, year & occupation



Prolog: complex types - structures

Database:

```
owns(john, car(red,corvette))
owns(john, cat(black,siamese,sylvester))
owns(elvis, copyright(song,"jailhouse rock"))
owns(tolstoy, copyright(book,"war and peace"))
owns(elvis, car(red,cadillac))
```

Query:

"Retrieve everything that John owns."

i.e. , Find `X` such that `owns(john,X)` is true.

```
answers:  X = car(red,corvette)
          X = cat(black,siamese,sylvester)
```

Query:

"Retrieve the colour and make of John's car."

i.e., `owns(john,car(Colour,Make))`

```
answer:  Colour = red
         Make = corvette
```

Prolog: complex types - structures

Same Database:

```
owns(john, car(red,corvette))
owns(john, cat(black,siamese,sylvester))
owns(elvis, copyright(song,"jailhouse rock"))
owns(tolstoy, copyright(book,"war and peace"))
owns(elvis, car(red,cadillac))
```

Query: "Who owns a red car?"

i.e., Find values for `Who` so that

∃Make `owns(Who,car(red,Make))` is true.

```
answers:  Who = john
          Who = elvis
```

Prolog: complex types - structures

Same Database:

```
owns(john, car(red,corvette))
owns(john, cat(black,siamese,sylvester))
owns(elvis, copyright(song,"jailhouse rock"))
owns(tolstoy, copyright(book,"war and peace"))
owns(elvis, car(red,cadillac))
```

Query: "Who owns a copyright?"

i.e., Find values for `Who` so that

$\exists$`X,Y owns(Who,copyright(X,Y))` is true.

```
answers:   Who = elvis
           Who = tolstoy
```