Programming in Prolog

Aggregates

Romain Barnoud

Thanks to:
Dr Fariba Sadri
Claudia Schulz

From lists to individuals

Program

```
city(asia, [tokyo, seoul, beijing]).
city(europe, [berlin, amsterdam, paris, london]).
city(america, [new_york, vancouver]).
```

From lists to individuals

Program

```
city(asia, [tokyo, seoul, beijing]).
city(europe, [berlin, amsterdam, paris, london]).
city(america, [new_york, vancouver]).
```


Queries


```
?- city(america, _List), member(City, _List).
City = new_york ;
City = vancouver ;
no
```

All cities in America


```
?- city(_Continent, _List), member(City, _List).
City = tokyo ;
City = seoul ;
...
City = vancouver ;
no
```

All cities in the world

From individuals to lists

Program

```
city(asia, tokyo).
city(asia, seoul).
city(asia, beijing).

city(europe, berlin).
city(europe, amsterdam).
city(europe, paris).
city(europe, london).

city(america, new_york).
city(america, vancouver).
```

How to get a list of the cities in Europe? Or in the entire world?

From individuals to lists

Program

```
city(asia, tokyo).
city(asia, seoul).
city(asia, beijing).

city(europe, berlin).
city(europe, amsterdam).
city(europe, paris).
city(europe, london).

city(america, new_york).
city(america, vancouver).
```


How to get a list of the cities in Europe? Or in the entire world?

Prolog has three built-in predicates for this kind of purpose:

- `findall/3`
- `bagof/3`
- `setof/3`

findall/3

Definition

`findall(+T, +Goal, -List)`:

`List` is the list of all instances of `T` for which `Goal` succeeds.

findall/3

Definition

`findall(+T, +Goal, -List)`:
`List` is the list of all instances of `T` for which `Goal` succeeds.

Examples

```
?- findall(City, city(europe, City), L).
L = [berlin, amsterdam, paris, london] ;
no

?- findall(City, city(Cont, City), L).
L = [tokyo, seoul, beijing, berlin, amsterdam, paris, london,
     new_york, vancouver] ;
no
```

findall/3

Definition

`findall(+T, +Goal, -List)`:

`List` is the list of all instances of `T` for which `Goal` succeeds.

Examples:

```
?- findall(Cont, city(Cont, City), L).
L = [asia, asia, asia, europe, europe, europe, europe, america,
     america] ;
no

?- findall(City, city(antarctica, City), L).
L = [] ;
no
```

findall/3

Definition

`findall(+T, +Goal, -List)`:
`List` is the list of all instances of `T` for which `Goal` succeeds.

Examples

```
?- findall(Cont-City, city(Cont, City), L).
L = [asia-tokyo, asia-seoul, asia-beijing, europe-berlin,
     europe-amsterdam, europe-paris, europe-london,
     america-new_york, america-vancouver] ;
no

?- findall(hello, city(asia, City), L).
L = [hello, hello, hello] ;
no
```

findall/3

Definition

`findall(+T, +Goal, -List)`:
`List` is the list of all instances of `T` for which `Goal` succeeds.

Things to be aware of:

- If `Goal` cannot be proven, `List` will be unified with the empty list.
- An instance $\texttt{T}\theta$ may appear several several times in `List` if there are different successful paths to prove $\texttt{Goal}\theta$.
- Free variables in `Goal` are existentially quantified (hence, there are not part of the answer) — more on that later.

bagof/3

Definition

`bagof(+T, +Goal, -List)`:

For a given substitution σ of free variables in `Goal` (all possible substitutions are generated through backtracking), `List` is the list of all instances of `T` such that `Goal`σ succeeds.

bagof/3

Definition

`bagof(+T, +Goal, -List)`:

For a given substitution σ of free variables in `Goal` (all possible substitutions are generated through backtracking), `List` is the list of all instances of `T` such that `Goal`σ succeeds.

Examples


```
?- bagof(City, city(europe, City), L).
L = [berlin, amsterdam, paris, london] ;
no

?- bagof(City, city(Cont, City), L).
Cont = america, L = [new_york, vancouver] ;
Cont = asia, L = [tokyo, seoul, beijing] ;
Cont = europe, L = [berlin, amsterdam, paris, london] ;
no
```

bagof/3

Definition

`bagof(+T, +Goal, -List)`:

For a given substitution σ of free variables in `Goal` (all possible substitutions are generated through backtracking), `List` is the list of all instances of `T` such that `Goal`σ succeeds.

Examples


```
?- bagof(Cont, city(Cont, City), L).
City = amsterdam, L = [europe] ;
City = beijing, L = [asia] ;
City = berlin, L = [europe] ;
...
City = tokyo, L = [asia] ;
City = vancouver, L = [america] ;
no

?- bagof(City, city(antarctica, City), L).
no
```

bagof/3

Definition

`bagof(+T, +Goal, -List)`:

For a given substitution σ of free variables in `Goal` (all possible substitutions are generated through backtracking), `List` is the list of all instances of `T` such that `Goal`σ succeeds.

Examples


```
?- bagof(Cont-City, city(Cont, City), L).
L = [asia-tokyo, asia-seoul, asia-beijing, europe-berlin,
     europe-amsterdam, europe-paris, europe-london,
     america-new_york, america-vancouver] ;
no

?- bagof(hello, city(asia, City), L).
City = beijing, L = [hello] ;
City = seoul, L = [hello] ;
City = tokyo, L = [hello] ;
no
```

bagof/3

Definition

`bagof(+T, +Goal, -List)`:

For a given substitution σ of free variables in `Goal` (all possible substitutions are generated through backtracking), `List` is the list of all instances of `T` such that `Goal`σ succeeds.

Things to be aware of:

- If `Goal` cannot be proven, `bagof/3` will fail.
- For a given instantiation σ of the free variables in `Goal`, an instance `T`θ may appear several several times in `List` if there are different successful paths to prove `Goal`$\sigma\theta$.
- Free variables in `Goal` are part of the answer.

setof/3

Definition

`setof(+T, +Goal, -List)`:

For a given substitution σ of free variables in `Goal` (all possible substitutions are generated through backtracking), `List` is the ***sorted set*** of all instances of `T` such that `Goal`σ succeeds.

setof/3

Definition

`setof(+T, +Goal, -List)`:

For a given substitution σ of free variables in `Goal` (all possible substitutions are generated through backtracking), `List` is the ***sorted set*** of all instances of `T` such that `Goal`σ succeeds.

Examples


```
?- setof(City, city(europe, City), L).
L = [amsterdam, berlin, london, paris] ;
no
```



```
?- setof(City, city(Cont, City), L).
Cont = america, L = [new_york, vancouver] ;
Cont = asia, L = [beijing, seoul, tokyo] ;
Cont = europe, L = [amsterdam, berlin, london, paris] ;
no
```

setof/3

Definition

`setof(+T, +Goal, -List)`:

For a given substitution σ of free variables in `Goal` (all possible substitutions are generated through backtracking), `List` is the ***sorted set*** of all instances of `T` such that `Goal`σ succeeds.

Examples


```
?- setof(Cont, city(Cont, City), L).
City = amsterdam, L = [europe] ;
City = beijing, L = [asia] ;
City = berlin, L = [europe] ;
...
City = tokyo, L = [asia] ;
City = vancouver, L = [america] ;
no

?- setof(City, city(antarctica, City), L).
no
```

setof/3

Definition

`setof(+T, +Goal, -List)`:

For a given substitution σ of free variables in `Goal` (all possible substitutions are generated through backtracking), `List` is the ***sorted set*** of all instances of `T` such that `Goal`σ succeeds.

Examples


```
?- setof(Cont-City, city(Cont, City), L).
L = [america-new_york, america-vancouver, asia-beijing,
     asia-seoul, asia-tokyo, europe-amsterdam,
     europe-berlin, europe-london, europe-paris].
no

?- setof(hello, city(asia, City), L).
City = beijing, L = [hello] ;
City = seoul, L = [hello] ;
City = tokyo, L = [hello] ;
no
```

setof/3

Definition

`setof(+T, +Goal, -List)`:

For a given substitution σ of free variables in `Goal` (all possible substitutions are generated through backtracking), `List` is the ***sorted set*** of all instances of `T` such that `Goal`σ succeeds.

Things to be aware of:

- If `Goal` cannot be proven, `setof/3` will fail.
- For any given instantiation σ of the free variables in `Goal`, the elements in `List` are sorted in ascending order and duplicates are removed.
- Free variables in `Goal` are part of the answer.

Prolog Ordering

Standard Order of Terms

1. *Variables* < *Floats* < *Integers* < *Atoms* < *Compound Terms*
2. *Variables* are sorted according to their order of appearance.
3. *Numbers* are compared according the natural order (but a float is always smaller than an integer[*]).
4. *Atoms* are compared alphabetically
5. *Compound terms* are compared: by their arity, then by their functor name (alphabetically), then recursively by their arguments from left to right

To compare terms, use the built-in predicates '==', '\==', '@<', '@=<', '@>', '@>=' or 'compare'.

---

[*]SWI Prolog uses a different ordering for numbers.
See http://www.swi-prolog.org/pldoc/man?section=compare

bagof/3 *vs.* setof/3

Program

```
foo(2,a).   foo(2,a).   foo(2,a).
foo(1,b).   foo(1,d).   foo(1,d).
foo(3,c).   foo(1,c).   foo(1,a).
```


bagof/3

```
?- bagof(A, foo(N,A), L).
N = 1, L = [b, d, c, d, a] ;
N = 2, L = [a, a, a] ;
N = 3, L = [c] ;
no
```

setof/3

```
?- setof(A, foo(N,A), L).
N = 1, L = [a, b, c, d] ;
N = 2, L = [a] ;
N = 3, L = [c] ;
no
```

Existential quantifiers

`bagof(T, V^Goal, L)`: variable `V` is existentially quantified, it will not be bound in goal (thus reproducing the behaviour of `findall/3`).

Program

```
bar(a,b,c).
bar(a,b,d).
```

```
bar(b,c,e).
bar(b,c,f).
```

```
bar(c,c,g).
```


Queries

```
?- bagof(Z, bar(X,Y,Z), L).
X = a, Y = b, L = [c, d] ;
X = b, Y = c, L = [e, f] ;
X = c, Y = c, L = [g] ;
no
```



```
?- bagof(Z, X^bar(X,Y,Z), L).
Y = b, L = [c, d] ;
Y = c, L = [e, f, g] ;
no

?- bagof(Z, X^Y^bar(X,Y,Z), L).
L = [c, d, e, f, g] ;
no
```

Cautions

Aggregates are powerful, but computationally very expensive as well.

So do not use them if not necessary!.

`setof/3` less efficient than `bagof/3` less efficient than `findall/3`.

To avoid

- `findall(X, member(X, L), List)`
  (use `L` directly).
- `findall(X, Goal, List), member(Elt, List)`
  (call `Goal` directly and use `X` in place of `Elt`).

What have we learned today?

- How to collect solutions in a list, using `findall/3`, `bagof/3` or `setof/3`
- What are the differences between these three aggregates