University of BRISTOL

Artificial Intelligence: Logic Programming II

Oliver Ray

From Datalog to Prolog

- Last week began our exploration of logic programming from the simple **Datalog** perspective (of relational database facts, rules and queries)
- This week explores some more advanced features of the **Prolog** language that involve **recursive definitions** and **structured terms**
- This lecture aims to build upon your experience of previous units by highlighting some key similarities and differences between **logic programming (Prolog)** and **classical first-order logic (FOL)** and **functional programming (Haskell)**
- In particular, we introduce the **list** datatype (which plays a fundamental a role in Prolog as it does in Haskell) and show how Prolog definitions of corresponding Haskell functions can often be **shorter**, **simpler** and can be used more **flexibly**!
- We conclude by introducing some **higher-order** and **meta-logical** list predicates

Declarative & Procedural Semantics

- One huge benefit of Prolog is that it allows us to use declarative **problem specifications** as a executable **solution generators** (so formalising a problem amounts to solving a problem!)
- This is because Prolog clauses can viewed both **declaratively** (as denoting a logical formula) and also **procedurally** (as a way of unfolding a query) – and these two semantics coincide
- For example (as we started to discuss in the last lab), the clause

  ```
  teenager(X) :- (male(X) ; female(X)), age(X,Y), Y>12, Y<20.
  ```

  can be declaratively interpreted as the following logical formula

  $$\forall X \forall Y(teenager(X) \leftarrow (male(X) \vee female(X)) \wedge age(X,Y) \wedge Y>12 \wedge Y<20)$$

  but it can also be procedurally interpreted as follows

  "to find an X that is a teenager, it suffices to first find an X that is a male or female, then find the age of X, and finally ensure the age is above 12 but below 20"
- In this way, every solution that Prolog returns is actually extracted from proof which shows that the solution is in fact logically correct (so computation amounts to proof construction)

University of BRISTOL

Pure and Impure Prolog

- Another huge benefit of Prolog is that logically specified definitions can often be queried in ways that go beyond their original purpose (facilitating reuse of programming effort)
- For example, if we can correctly specify how to concatenate two lists into one, then that same definition ought to enable us to conversely split one list into two (as we will see later)
- In practice, efficiency considerations necessitate the introduction of "**impure**" features into Prolog which, if not used carefully, can potentially break the correspondence between the declarative and procedural semantics and limit the ways in which definitions can be queried
- Such features include built-in **arithmetic operators** (for efficiently comparing and evaluating arithmetic expressions) and Prolog's **negation-as-failure** operator (which has enormous practical application in AI for knowledge representation and non-monotonic reasoning)
- When used with care, such features can enhance the readability and efficiency of programs, but they can be potentially "**unsafe**" if they are allowed to be called with unbound variables

e.g. `teenager(X) :- Y>12, (male(X) ; female(X)), age(X,Y), Y<20.`

arithmetic comparisons can only be called with grounded arguments!

Prolog vs. Classical Logic

Logic Programming is not Classical Logic (although they are closely related)

- Both are Turing Complete but they embody quite different modelling paradigms
- Prolog's core syntax restricts first-order logic (to definite clausal normal form)
- Prolog's core semantics extends first-order logic (to minimal model constructions)
- Prolog assumes false any atoms with no reason to suggest they may be true!
  - This allows Prolog to compute relations (e.g. transitive closure) that are not even classically definable (see later - as they imply a closed world semantics)
  - And this is the foundation of Prolog's famous "negation-as-failure" operator that is very useful in AI for knowledge representation and reasoning
- Prolog also has a simple procedural interpretation (consistent with its declarative semantics, if some care is taken) by attaching additional operational meaning to the order in which clauses and literals are written (by default: top->bottom & left->right)

Logical Symbols (recap)

Variables	e.g. X Y	denote arbitrary objects (in some implicit domain)
Constants	e.g. oliver peter	denote specific objects (in that same domain)
Functions	e.g. mother/1 father/1	denote mappings between objects
Propositions	e.g p q	represent unstructured assertions
Predicates	e.g. happy/1 loves/2	represent object properties and relations
Connectives	e.g. $\neg$ $\wedge$ $\vee$ $\leftarrow$ $\leftrightarrow$ …	not and or if iff …
Quantifiers	e.g. $\forall$ $\exists$	for_all there_exists
Truth values	e.g. T $\perp$	truth falsity (often called logical constants)
Punctuation	e.g. ( ) ,	brackets commas
Equality	e.g. =	

Functions and predicates have "arities" (number of arguments they take); constants and propositions can be seen as arity 0 functions or predicates (so may be written with an empty tuple of arguments)

Predicate and function symbols (including propositions and constants) comprise the signature (or parameters) of the language (as their meaning is context dependent, unlike the other fixed symbols).

University of BRISTOL

Flash Quiz!!!

Q2) If x(y(z)) is a well-formed ground formula of first-order classical logic where x, y and z are symbols, then what sort of symbols must each of them be (e.g. a variable, constant, function, predicate, connective, punctuation or other)?

predicate function constant

This result is implied by the syntactic definition of formulae in first order logic (which, for convenience is summarized on the next slide). It is important to note that, in classical logic, functions may appear inside predicates and other functions, but predicates may not appear inside other predicates or functions!

Logical Expressions (recap)

A **term** is a **constant** c, **variable** X or **function** f of artity n applied to an n-tuple of terms $f(t_1, \ldots, t_n)$

An **atom** is a **proposition** p or a **predicate** r of artity n applied to an n-tuple of terms $r(t_1, \ldots, t_n)$

A **formula** is an atom a; a **logical constant** T or $\perp$; a **negation** $\neg\phi$ of a formula ϕ; a **conjunction** $\phi\wedge\gamma$, **disjunction** $\phi\vee\gamma$ or **conditional** $\phi\leftarrow\gamma$ of formulae ϕ and γ; a **universal quantification** $\forall X \phi$ or an **existential quantification** $\exists X \phi$ of a formula ϕ with respect to a variable X

Common abbreviations: **implication** $f\rightarrow g \equiv g\leftarrow f$, **equivalence** $f\leftrightarrow g \equiv f\rightarrow g \wedge g\leftarrow f$, **nested** quantification $\forall_{XY}\, \phi \equiv \forall X \forall Y \phi$ or $\exists_{XY}\, \phi \equiv \exists X \exists Y \phi$ and **restricted quantification** $\forall_{X\#@}\, \phi \equiv \forall X (X\#@ \rightarrow \phi)$ or $\exists_{X\#@}\, \phi \equiv \exists X{:}(X\#@ \wedge \phi)$ where # and @ are various operators and expressions (eg "≥1"). **Unique existence** is $\exists!_X\, \phi \equiv \exists X \forall Y (\phi_{[X/Y]} \leftrightarrow X{=}Y)$ where $\phi_{[X/Y]}$ means replace X by Y in ϕ (which should not already contain Y)

A **minimal set** of logical symbols (e.g. $\forall$ $\neg$ $\wedge$) may be used to define all of the others;

A **standard precedence** order is used ($\forall$ $\exists$ $\neg$ $\wedge$ $\vee$ $\leftarrow$ $\rightarrow$ $\leftrightarrow$ in decreasing order) to avoid excessive brackets and improve readability; and some parameters may be written in postfix, infix or mixfix style

A logical language is the "**smallest set**" containing all such inductively defined terms

University of BRISTOL

Flash Quiz!!!

Q1) Which of the following are logically equivalent to this first-order formula:

$$\forall X \forall Y(\mathrm{teenager}(X) \leftarrow (\mathrm{male}(X) \vee \mathrm{female}(X)) \wedge \mathrm{age}(X,Y) \wedge Y{>}12 \wedge Y{<}20)$$

a) $\forall X \forall Y(\mathrm{teenager}(X) \leftarrow (\mathrm{female}(X) \vee \mathrm{male}(X)) \wedge \mathrm{age}(X,Y) \wedge Y{>}12 \wedge Y{<}20)$ √

b) $\forall X \forall Y(\mathrm{teenager}(X) \leftarrow Y{>}12 \wedge Y{<}20 \wedge (\mathrm{female}(X) \vee \mathrm{male}(X)) \wedge \mathrm{age}(X,Y))$ √

c) $\forall Y \forall X(\mathrm{teenager}(X) \leftarrow Y{>}12 \wedge ((\mathrm{female}(X) \wedge Y{<}20) \vee (\mathrm{male}(X) \wedge Y{<}20)) \wedge \mathrm{age}(X,Y))$ √

d) $\forall X(\mathrm{teenager}(X) \leftarrow \exists Y((\mathrm{male}(X) \vee \mathrm{female}(X)) \wedge \mathrm{age}(X,Y) \wedge Y{>}12 \wedge Y{<}20))$ √

e) $\forall X(\mathrm{teenager}(X) \leftarrow (\mathrm{male}(X) \vee \mathrm{female}(X)) \wedge \exists Y(\mathrm{age}(X,Y) \wedge Y{>}12 \wedge Y{<}20))$ √

f) $\forall X \forall Y(\mathrm{teenager}(X) \vee \neg((\mathrm{male}(X) \vee \mathrm{female}(X)) \wedge \mathrm{age}(X,Y) \wedge Y{>}12 \wedge Y{<}20))$ √

g) $\forall X \forall Y(\mathrm{teenager}(X) \vee (\neg\mathrm{male}(X) \wedge \neg\mathrm{female}(X)) \vee \neg\mathrm{age}(X,Y) \vee Y{\le}12 \vee Y{\ge}20)$ √

h) $$\boxed{\underbrace{\forall X \forall Y}_{\text{prefix}} \; + \; \underbrace{\left(\begin{array}{l} (\mathrm{teenager}(X) \vee \neg\mathrm{male}(X) \vee \neg\mathrm{age}(X,Y) \vee Y{\le}12 \vee Y{\ge}20) \wedge \\ (\mathrm{teenager}(X) \vee \neg\mathrm{female}(X) \vee \neg\mathrm{age}(X,Y) \vee Y{\le}12 \vee Y{\ge}20) \end{array} \right)}_{\text{matrix}}} \; = \; \text{clausal form}$$ √

Clausal Form

- To facilitate the storage of logical formulae in computer memory and simplify the inference procedures needed for their manipulation, it is convenient to transform formulae into some restricted subsets of First-Order Logic (FOL) known as **normal forms**
- Automated reasoning methods such as logic programming, theorem proving, and satisfiability solving all make heavy use of so-called **prenex normal forms** (PNFs) that only allow formulae of the form **⟨prefix⟩⟨matrix⟩** where the prefix is a string of quantifiers and the matrix is a quantifier-free formula
- Most common of these is the so-called **conjunctive normal form** (CNF) where the prefix is further restricted to a string of universal quantifiers and the matrix is further restricted to a conjunction of disjunctions (aka **clauses**) of atoms or their negations (aka **literals**)
- **Clausal form** is CNF with the prefix omitted and the matrix written as either a set of sets of literals or a set of clauses of the form **⟨head⟩** $\leftarrow$ **⟨body⟩** where the head is an (often implicit) disjunction of literals (or $\perp$ if empty) and the body is an (often implicit) conjunction of literals (or T if empty)
- **Prolog** is a variation of clausal form where exactly one atom must be used in the head of each clause but nested conjunctions, disjunctions and negations of atoms may be used in the body; and which are usually written using symbols ":-", ",", ";", "\+" in place of connectives "$\leftarrow$", "$\wedge$", "$\vee$" ,"$\neg$" respectively

From Prolog to Classical Logic

```
plays(M,A,R) :- actor(M,A,R) ; actress(M,A,R).
```

$$\equiv \forall_{MAR} \; (\; plays(M,A,R) \leftarrow (\; actor(M,A,R) \vee actress(M,A,R) \;)\;)$$

```
solo(M,A) :- plays(M,A,_), \+ (plays(M,B,_), A\==B).
```

$$\equiv \forall_{MABXY} \; (\; solo(M,A) \leftarrow (\; plays(M,A,X) \wedge \neg (\; plays(M,B,Y) \wedge A \neq B \;)\;)\;)$$

$$\equiv \forall_{MABXY} \; (\; solo(M,A) \leftarrow (\; plays(M,A,X) \wedge (\neg plays(M,B,Y) \vee A{=}B \;)\;)\;)$$

≠ `solo(M,A) :- plays(M,A,_), ( \+plays(M,B,_) ; A==B ).`

operationally false – so does not coincide with classical declarative semantics!!!

To go the other way, we can use App.B1 of Simply Logical (p.201-6) to convert any FOL formula ϕ to an equivalent PNF formula and then to an 'almost' equivalent set of clauses Σ (in the sense of Sec. 2.5 on p.38-41 that is sufficient for computational logic applications) whereby (i) $\Sigma \models \phi$ and (ii) $\Sigma \models \bot$ iff $\phi \models \bot$

Transitive Closure and Friends

- Suppose we give Prolog the definition of some base relation (such as biblical fatherhood):

```
father(adam,cain). father(adam,abel). father(adam,seth).
father(cain,enoch). father(enoch,irad).            ...
```

- Then we can easily compute its transitive closure (to give patrilineal ancestorship):

```
ancestor(X,Y) :- father(X,Y).
ancestor(X,Z) :- father(X,Y), ancestor(Y,Z).
```

- It is interesting to note that, while Prolog does exactly what we expect and correctly computes the transitive closure, it has been shown that this operation is not even definable within classical FOL (Fagin, 1974).
- For example, $\forall_{XY}$ (ancestor(X,Y) $\leftarrow$ father(X,Y)) and $\forall_{XYz}$ (ancestor(X,Z) $\leftarrow$ father(X,Y), ancestor(Y,Z)) can easily be shown to have valid classical models in which every person is an ancestor of every other!
- The issue is that FOL can't express the required closed-world property that the intended solution is in fact the *smallest* possible relation satisfying the transitive closure property!
- To a first approximation, Prolog assumes false any atoms not true in the *minimal* classical model of the program

Prolog vs. Functional Programming

Logic Programming is not Functional Programming (though they are closely related)

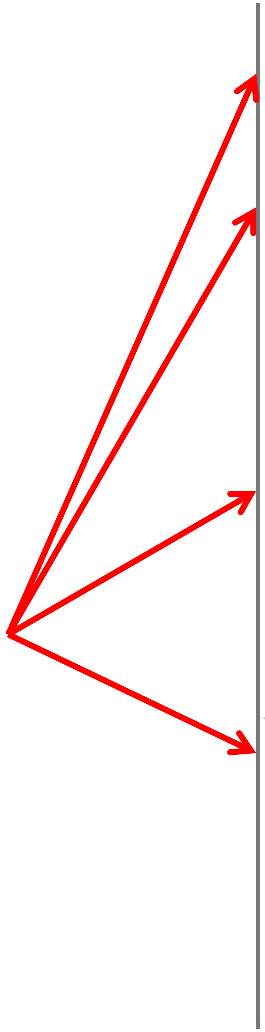
- Both are Turing Complete but they embody quite different modelling paradigms
- Mathematical Functions always return exactly one answer
- BUT Logical Predicates may relate zero, one or many different answers
- AND Prolog functions are NOT evaluated – they're like Haskell data constructors (except for arithmetic operators like "is" which evaluate ground arithmetic terms)
- Compared to Haskell functions, you will need to add an extra argument to the corresponding Prolog predicate in order to explicitly represent any "answers"
- Typically, Prolog definitions will be shorter, simpler and may be used more flexibly than their Haskell counterparts (as there'll be no need to denote types, or to represent the false cases of Boolean functions, or to fix the direction in which a relation will be used) - even for basic list processing definitions…

List Notation

Lists are the prototypical structured term

- the empty list is []/0 and the list constructor: ‘[|]’/2
  - in other Prologs and SWI v≤6 the list constructor is ‘.’/2
  - but the ‘.’/2 is now reserved for dict structures as of SWI v≥7
- in Prolog lists are written and displayed in shorthand notation
  - ‘[|]’(Head,Tail) is written [Head|Tail]
  - [Head|[]] is written [Head]
  - [Head1| [Head2|Tail]] is written [Head1,Head2|Tail]

Primitive List Definitions: Haskell vs. Prolog



verifier

forwards generator

backwards generator

arbitrary generator

University of BRISTOL

Primitive List Definitions: Haskell vs. Prolog

```
% head :: [a] -> a
% head (x:_) = x
head([X|_],X).

% tail :: [a] -> [a]
% tail (_:xs) = xs
tail([_|Xs],Xs).

%null :: [a] -> Bool
%null [] = True
%null (_:_) = False
null([]).

% length :: [a] -> Int
% length [] = 0
% length (_:l) = 1 + length l
length([],0).
length([_|L],N):-length(L,N0), N is 1+N0.

% member :: (Eq a) => a->[a]->Bool
% member x [] = False
% member x (y:ys) = x == y || member x ys
member(X,[X|_]).
member(X,[_|Ys]):-member(X,Ys).
```

```
%append :: [a] -> [a] -> [a]
%append [] ys = ys
%append (x:xs) ys = x : append xs ys
append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]):-append(Xs,Ys,Zs).

% prefix :: (Eq a) => [a] -> [a] -> Bool
% prefix _ [] = True
% prefix [] (_:_) = False
% prefix (x:xs) (y:ys) = x == y && prefix xs ys
prefix(Xs, Ys) :- append(Ys, _, Xs).

% suffix :: (Eq a) => [a] -> [a] -> Bool
% suffix xs xs = True
% suffix [] (_:_) = False
% suffix (x:xs) ys = suffix xs ys
suffix(Xs, Ys) :- append(_, Ys, Xs).

% sublist :: (Eq a) => [a] -> [a] -> Bool
% sublist xs ys = any (prefix ys) (tails xs)
% where
%    tails :: [a] -> [[a]]
%    tails xs = xs : case xs of
%      [] -> []
%      _ : xs' -> tails xs'
sublist(Xs, Ys) :- suffix(Xs, Zs), prefix(Zs, Ys).
```

```
suffix([1,2,3,4],[3,4]).
true
```

verifier

```
suffix([1,2,3,4],X).
X = [1, 2, 3, 4]
X = [2, 3, 4]
X = [3, 4]
X = [4]
X = []
```

forwards generator

```
suffix(X,[3,4]).
X = [3, 4]
X = [_1344, 3, 4]
X = [_1344, _1350, 3, 4]
X = [_1344, _1350, _1356, 3, 4]
Next | 10 | 100 | 1,000 | Stop
```

backwards generator

```
suffix(X,Y).
X = Y
X = [_1348|Y]
X = [_1348, _1354|Y]
X = [_1348, _1354, _1360|Y]
Next | 10 | 100 | 1,000 | Stop
```

arbitrary generator

Note: the Prolog operator “is” forces evaluation of arithmetic expression

From Haskell to Prolog

base case

`% length [] = 0`

`% length([], X) :- X is 0 .`

Note: we try to force any mandatory bindings into the head of the clause

`length([], 0).`

recursive case

`% length (_:l) = 1 + length l`

`% length([_|L], X) :- X is 1 + Y , length(L, Y) .`

Note: we must move the recursive “length” call before the “is” call since the latter requires its right argument to be ground at call time

`length([_|L], X) :- length(L, Y) , X is 1 + Y .`

Example: (Peano) Addition

```
% data Peano = Zero | Succ Peano
% add Zero b  =  b
% add (Succ a) b  =  Succ (add a b)
```

base case

```
% add(zero, B, C) :- C = B.
```

```
add(zero,B,B).
```

recursive case

```
% add(succ(A), B, C) :- C = succ(D), add(A,B,D).
```

```
add(succ(A), B, succ(D)) :- add(A,B,D).
```

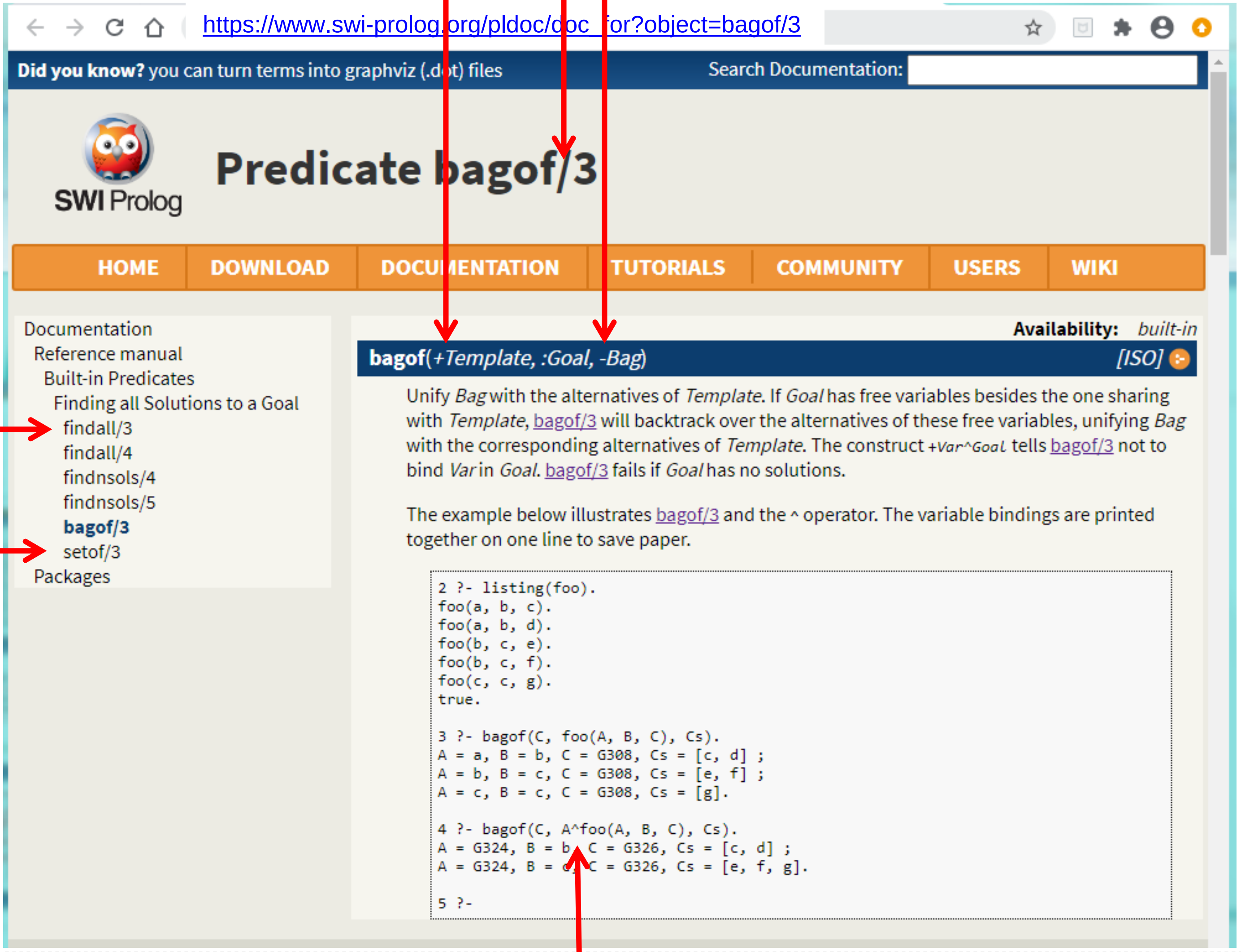
Common Mistakes (from lab)

```
m(_) :- m(n) ; m(s) ; m(e) ; m(w) .              % X  (infinite loop)
m(_) :- n ; s ; e ; w .                          % X  (undefined predicates)
m(X) :- X = (n ; s ; e ; w) .                    % X  (compound term)
m(X) :- X==n ; X==s ; X==e ; X==w .              % X  (input arg only)
m(X) :- X=n ; X=s ; X=e ; X=w .                  % `/ (delayed binding)
m(n). m(s). m(e). m(w).                          % `/ (best solution)

complete(L) :-                                   % X
    N=ailp_grid_size(_),                           % complex term
    N2=N*N,                                        % more complex term
    length(L)=N2.                                  % contradiction
complete(L) :-                                   % `/
    ailp_grid_size(N),                             % integer
    N2 is N*N,                                     % integer
    length(L,N2).                                  % correct
```

University of BRISTOL

List forming (higher-order) predicates


https://www.swi-prolog.org/pldoc/doc_for?object=bagof/3

Did you know? you can turn terms into graphviz (.dot) files   Search Documentation:

SWI Prolog

Predicate bagof/3

HOME | DOWNLOAD | DOCUMENTATION | TUTORIALS | COMMUNITY | USERS | WIKI

Documentation
Reference manual
Built-in Predicates
Finding all Solutions to a Goal
findall/3
findall/4
findnsols/4
findnsols/5
bagof/3
setof/3
Packages

Availability: *built-in*

bagof(*+Template, :Goal, -Bag*) *[ISO]*

Unify *Bag* with the alternatives of *Template*. If *Goal* has free variables besides the one sharing with *Template*, bagof/3 will backtrack over the alternatives of these free variables, unifying *Bag* with the corresponding alternatives of *Template*. The construct *+Var^Goal* tells bagof/3 not to bind *Var* in *Goal*. bagof/3 fails if *Goal* has no solutions.

The example below illustrates bagof/3 and the ^ operator. The variable bindings are printed together on one line to save paper.

```
2 ?- listing(foo).
foo(a, b, c).
foo(a, b, d).
foo(b, c, e).
foo(b, c, f).
foo(c, c, g).
true.

3 ?- bagof(C, foo(A, B, C), Cs).
A = a, B = b, C = G308, Cs = [c, d] ;
A = b, B = c, C = G308, Cs = [e, f] ;
A = c, B = c, C = G308, Cs = [g].

4 ?- bagof(C, A^foo(A, B, C), Cs).
A = G324, B = b, C = G326, Cs = [c, d] ;
A = G324, B = c, C = G326, Cs = [e, f, g].

5 ?-
```

- bagof(C,Q,L), setof(C,Q,L) and findall(C,Q,L) collect all instantiations of term **C** generated by solutions to query **Q** all together in a list **L**

- bagof/3 allows explicit (existential) quantification of (free) query variables (in Q but not in **C**); and *fails* if Q fails

- setof/3 is like bagof/3 but returns a sorted list (without duplicates)

- findall/3 is like bagof/3 but with all implicit quantification over *all* free query variables; and it *succeeds* with L=[] if Q fails

Note: arities (.../n) and mode declarations (+,-,? and :) are only used in predicate documentation !!!

bagof/3

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

foo(b, c, e).
foo(b, c, f).

foo(c, c, g).

/** <examples>

?- bagof(C, foo(A, B, C), Cs).

?- bagof(C, A^foo(A, B, C), Cs).

*/

```

```
bagof(C, foo(A, B, C), Cs).

A = a,
B = b,
Cs = [c, d]
A = b,
B = c,
Cs = [e, f]
A = B, B = c,
Cs = [g]
```

```
bagof(C, A^foo(A, B, C), Cs).

B = b,
Cs = [c, d]
B = c,
Cs = [e, f, g]
```

explicit existential quantification

Flash Quiz – Question 3

```prolog
:- discontiguous movie/2,director/2,actor/3.
:- dynamic actress/3.

movie(covid19, 2019).
director(covid19, oliver_ray).
actor(covid19, nirav_ajmeri, the_hero).
actor(covid19, nirav_ajmeri, the_villain).
actor(covid19, nirav_ajmeri, the_narrator).
actor(covid19, seth_bullock, the_victim).

plays(M,A,R) :- actor(M,A,R) ; actress(M,A,R).

multi_role_a(M,A) :- plays(M,A,R1), plays(M,A,R2), R1\==R2.
multi_role_b(M,A) :- plays(M,A,R1), R1\==R2, plays(M,A,R2).
multi_role_c(M,A) :- plays(M,A,R1), R1\=R2, plays(M,A,R2).
multi_role_d(M,A) :- plays(M,A,R1), plays(M,A,R2), R1@>R2.
multi_role_e(M,A) :- bagof(R,plays(M,A,R),[_,_|_]).
```

incorrect

Explain the purpose of the directives at the top of this program; and, for each definition of multi_role explain how many copies of the correct answer are produced and how many incorrect answers are produced?

include/3 and exclude/3

include(*:Goal, +List1, ?List2*) *[det]*

Filter elements for which *Goal* succeeds. True if *List2* contains those elements Xi of *List1* for which `call(Goal, Xi)` succeeds.

See also

exclude/3, partition/4, convlist/3.

Compatibility

Older versions of SWI-Prolog had sublist/3 with the same arguments and semantics.

exclude(*:Goal, +List1, ?List2*) *[det]*

Filter elements for which *Goal* fails. True if *List2* contains those elements Xi of *List1* for which `call(Goal, Xi)` fails.

See also

include/3, partition/4

Examples

▼ ★ **Use include/3 to filter odd numbers**

```
is_odd(I) :-
    0 =\= I mod 2.
```

```
?- numlist(1, 6, List),
   include(is_odd, List, Odd).
List = [1, 2, 3, 4, 5, 6],
Odd = [1, 3, 5].
```

Folds are also supported: https://www.swi-prolog.org/pldoc/man?section=apply

And there is even a library for lambda expressions: https://www.swi-prolog.org/pldoc/man?section=yall

between/3 and maplist/3

Availability: *built-in*

between(*+Low, +High, ?Value*)

Low and *High* are integers, *High* >=*Low*. If *Value* is an integer, *Low* =<*Value* =<*High*. When *Value* is a variable it is successively bound to all integers between *Low* and *High*. If *High* is `inf` or `infinite`[111] between/3 is true iff *Value* >=*Low*, a feature that is particularly interesting for generating integers from a certain value.

```
?- between(2,5,Result).
Result = 2
Result = 3
Result = 4
Result = 5
```

Availability: `:- use_module(library(apply)).` *(can be autoloaded)*

~~**maplist**(*:Goal, ?List1*)~~

maplist(*:Goal, ?List1, ?List2*)

~~**maplist**(*:Goal, ?List1, ?List2, ?List3*)~~

~~**maplist**(*:Goal, ?List1, ?List2, ?List3, ?List4*)~~

True if *Goal* is successfully applied on all matching elements of the list. The maplist family of predicates is defined as:

```
maplist(P, [X11,...,X1n], ..., [Xm1,...,Xmn]) :-
    P(X11, ..., Xm1),
    ...
    P(X1n, ..., Xmn).
```

This family of predicates is deterministic iff *Goal* is deterministic and *List1* is a proper list, i.e., a list that ends in `[]`.

```
maplist(p(X1,…,Xk), [Y1,…,Yn], [Z1,…,Zn]) :-
        p(X1,…,Xk, Y1, Z1),
        …
        p(X1,…,Xk, Yn, Zn).
```

```
?- maplist(plus(1), [1, 2, 3, 4, 5], Result).
Result = [2, 3, 4, 5, 6]
```

Availability: *built-in*

forall(*:Cond, :Action*) *[semidet]*

For all alternative bindings of *Cond*, *Action* can be proven. The example verifies that all arithmetic statements in the given list are correct. It does not say which is wrong if one proves wrong.

```
forall(member(Result = Formula, [2 = 1 + 1, 4 = 2 * 2]), Result is Formula).
true
```

The predicate forall/2 is implemented as `\+ ( Cond, \+ Action)`, i.e., *There is no instantiation of Cond for which Action is false.*. The use of double negation implies that forall/2 *does not change any variable bindings*. It proves a relation. The forall/2 control structure can be used for its side-effects. E.g., the following asserts relations in a list into the dynamic database:

```
?- forall(member(Child-Parent, ChildPairs),
          assertz(child_of(Child, Parent))).
```

Using forall/2 as `forall(Generator, SideEffect)` is preferred over the classical *failure driven loop* as shown below because it makes it explicit which part of the construct is the generator and which part creates the side effects. Also, unexpected failure of the side effect causes the construct to fail. Failure makes it evident that there is an issue with the code, while a failure driven loop would succeed with an erroneous result.

```
        ...,
        (   Generator,
            SideEffect,
            fail
        ;   true
        )
```

If your intent is to create variable bindings, the forall/2 control structure is inadequate. Possibly you are looking for maplist/2, findall/3 or foreach/2.

forall/2

```
% higher order predicate
count_sheep(N) :-
    forall(
        between(1,N,X),
        format('~w..',[X])
    ),
    writeln("ZZZzzz").
```

```
count_sheep(5)
1..2..3..4..5..ZZZzzz
true
```

```
% failure driven loop
count_sheep(N) :-
    between(1,N,X),
    format('~w..',[X]),
    fail
    ;
    writeln("ZZZzzz").
```

sort/2 and sort/4

I'm sure your are aware that the default sort behaviour when sorting lists is to first sort on the values of the first item (if it exists), and then sort on the values of the second items (if it exists), etc

Thus sort([[g,b], [c,d,e], [f], []], S). returns S = [[], [c, d, e], [f], [g, b]].
Other behaiours can be obtained using sort 4 OR by tuning the lists into compound terms (e.g. by prepending with some score such as the length of the list or some heuristic cost)

Thus sort([2-[g,b], 3-[c,d,e], 1-[f], 0-[]], S). returns S = [0-[], 1-[f], 2-[g, b], 3-[c, d, e]].

The above rewriting ensures shorter lists are placed before longer lists (using the minus sign to act as a uninterpreted infix functor that simply groups a number with a list in this case).

sort/4 (swi-prolog.org)

If-Then-Else

Thank you