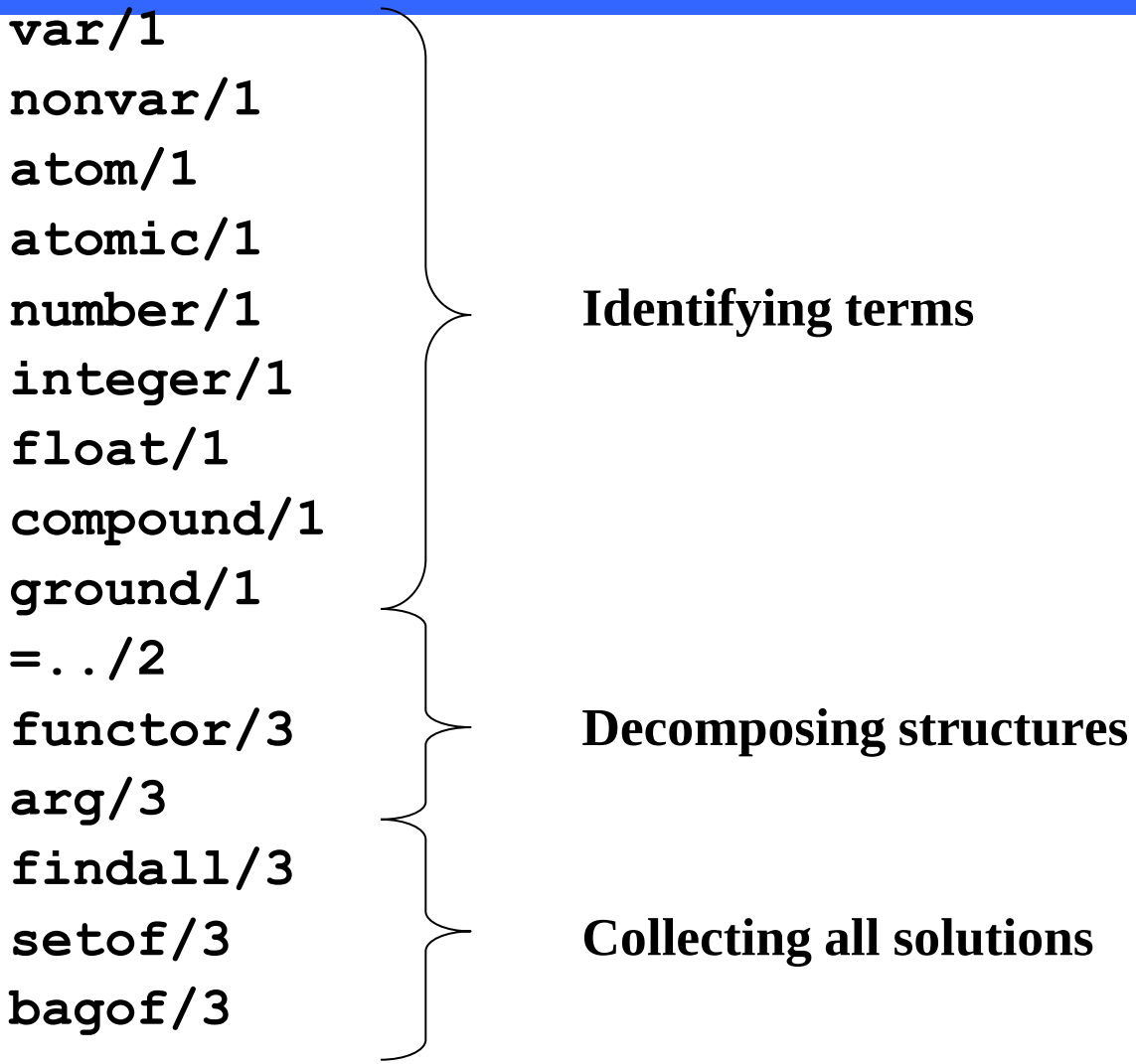
ITSE301 Logic Programming

Built-in Predicates

14-5-2024

Built-in Predicates



Identifying Terms

- **These built-in predicates allow the type of terms to be tested.**

`var(X)`	**succeeds if X is currently an uninstantiated variable.**
`nonvar(X)`	**succeeds if X is not a variable, or already instantiated**
`atom(X)`	**is true if X currently stands for an atom**
`number(X)`	**is true if X currently stands for a number**
`integer(X)`	**is true if X currently stands for an integer**
`float(X)`	**is true if X currently stands for a real number.**
`atomic(X)`	**is true if X currently stands for a number or an atom.**
`compound(X)`	**is true if X currently stands for a structure.**
`ground(X)`	**succeeds if X does not contain any uninstantiated variables.**

var/1, nonvar/1, atom/1

`var(X)` is true if X is currently an uninstantiated variable.

```
?- var(X).
true

?- X = 5, var(X).
false

?- var([X]).

true
```

var/1, nonvar/1, atom/1

`nonvar(X)` is true if X is not a variable, or already instantiated.

```
| ?- nonvar(X).
no

?- X = 5,nonvar(X).
X = 5 ?


?-nonvar([X]).
true
```

var/1, nonvar/1, atom/1

`atom(X)` **is true if X currently stands for an atom: a non-variable term with 0 arguments, and not a number**

```
?- atom(paul).
true

?- X = paul,atom(X).
X = paul ?

?- atom([]).

?-atom([a,b]).
false
```

number/1, integer/1, float/1

`number(X)` **is true if X currently stands for any number**

```
| ?- number(X).
no

| ?- X=5,number(X).
X = 5 ?
yes

| ?- number(5.46).
yes
```

To identify what type of number it is use:

`integer(X)` **is true if X currently stands for an integer (a whole positive or negative number or zero).**

`float(X)` **is true if X currently stands for a real number.**

```
| ?- integer(5).
yes

| ?- integer(5.46).
no

| ?- float(5).
no

| ?- float(5.46).
yes
```

atomic/1, compound/1, ground/1

- If `atom/1` is too specific then you can use **`atomic/1`** which accepts numbers and atoms.

```
| ?- atom(5).      | ?- atomic(5).
no                 yes
```

- If `atomic/1` fails then the term is either an uninstantiated variable (which you can test with `var/1`) or a **`compound`** term:

```
|?-compound([]). |?-compound([a]). |?-compound(b(a)).
no                yes                yes
```

- **`ground(X)`** succeeds if X does not contain any uninstantiated variables. Also checks inside compound terms.

```
|?-ground(X).              |?-ground(a(b,X)).
no                         no
|?-ground(a).              |?-ground([a,b,c]).

yes                        yes
```

Decomposing Structures

- When using compound structures you can't use a variable to check or make a functor.

```
|?- X=tree, Y = X(maple).
Syntax error Y=X<<here>>(maple)
```

functor(T,F,N) is true if F is the principal functor of T and N is the arity of F.

arg(N,Term,A) is true if A is the Nth argument in Term.

```
|?-functor(t(f(X),a,T),Func,N).
N = 3, Func = t ?
yes
```

```
|?-arg(2,t(t(X),[]),A).
A = [] ?
yes
```

```
| ?- functor(D,date,3), arg(1,D,11), arg(2,D,oct),
   arg(3,D,2004).
D = date(11,oct,2004) ? yes
```

Decomposing Structures (2)

- **We can also decompose a structure into a list of its components using `=../2`.**

  `Term =.. L` **is true if L is a list that contains the principal functor of Term, followed by its arguments.**

```
| ?- f(a,b) =.. L.
L = [f,a,b] ?
yes
```

```
|?- T =.. [is_blue,sam,today].
T = is_blue(sam,today) ?
yes
```

- **By representing the components of a structure as a list they can be recursively processed without knowing the functor name.**

```
| ?- f(2,3)=..[F,N|Y], N1 is N*3, L=..[F,N1|Y].
L = f(6,3)?
yes
```

Collecting all solutions

- You've seen how to generate all of the solutions to a given goal, at the prompt (;):

```
| ?- member(X, [1,2,3,4]).
    X = 1 ? ;
    X = 2 ? ;
    X = 3 ? ;
    X = 4 ? ;
    no
```

- It would be nice if we could generate all of the solutions to some goal within a program.
- There are three similar built-in predicates for doing this:

findall/3

setof/3

bagof/3

Meta-predicates

- `findall/3`, `setof/3`, and `bagof/3` are all ***meta-predicates***
  - ❖they manipulate Prolog's proof strategy.

`findall(X,P,L)`
`setof(X,P,L)`
`bagof(X,P,L)`

All produce a list L of all the objects X such that goal P is satisfied (e.g. `age(X,Age)`).

- They all repeatedly call the goal P, instantiating the variable X within P and adding it to the list L.
- They succeed when there are no more solutions.
- Exactly simulate the repeated use of **';'** at the SICStus prompt to find all of the solutions.

findall/3

- **`findall/3`** is the most straightforward of the three, and the most commonly used:

```
| ?- findall(X, member(X, [1,2,3,4]), Results).
    Results = [1,2,3,4]
    yes
```

- This reads: `find all of the Xs, such that X is a member of the list `[1,2,3,4]` and put the list of results in `Results`'.
- Solutions are listed in the result in the same order in which Prolog finds them.
- If there are duplicated solutions, all are included. If there are infinitely-many solutions, it will never terminate!

findall/3 (2)

- **We can use `findall/3` in more sophisticated ways.**
- **The second argument, which is the goal, might be a compound goal:**

```
| ?- findall(X, (member(X, [1,2,3,4]), X > 2), Results).
     Results = [3,4]?
     yes
```

- **The first argument can be a term of any complexity:**

```
|?- findall(X/Y, (member(X,[1,2,3,4]), Y is X * X),
    Results).
    Results = [1/1, 2/4, 3/9, 4/16]?
    yes
```