FURTHER PROLOG

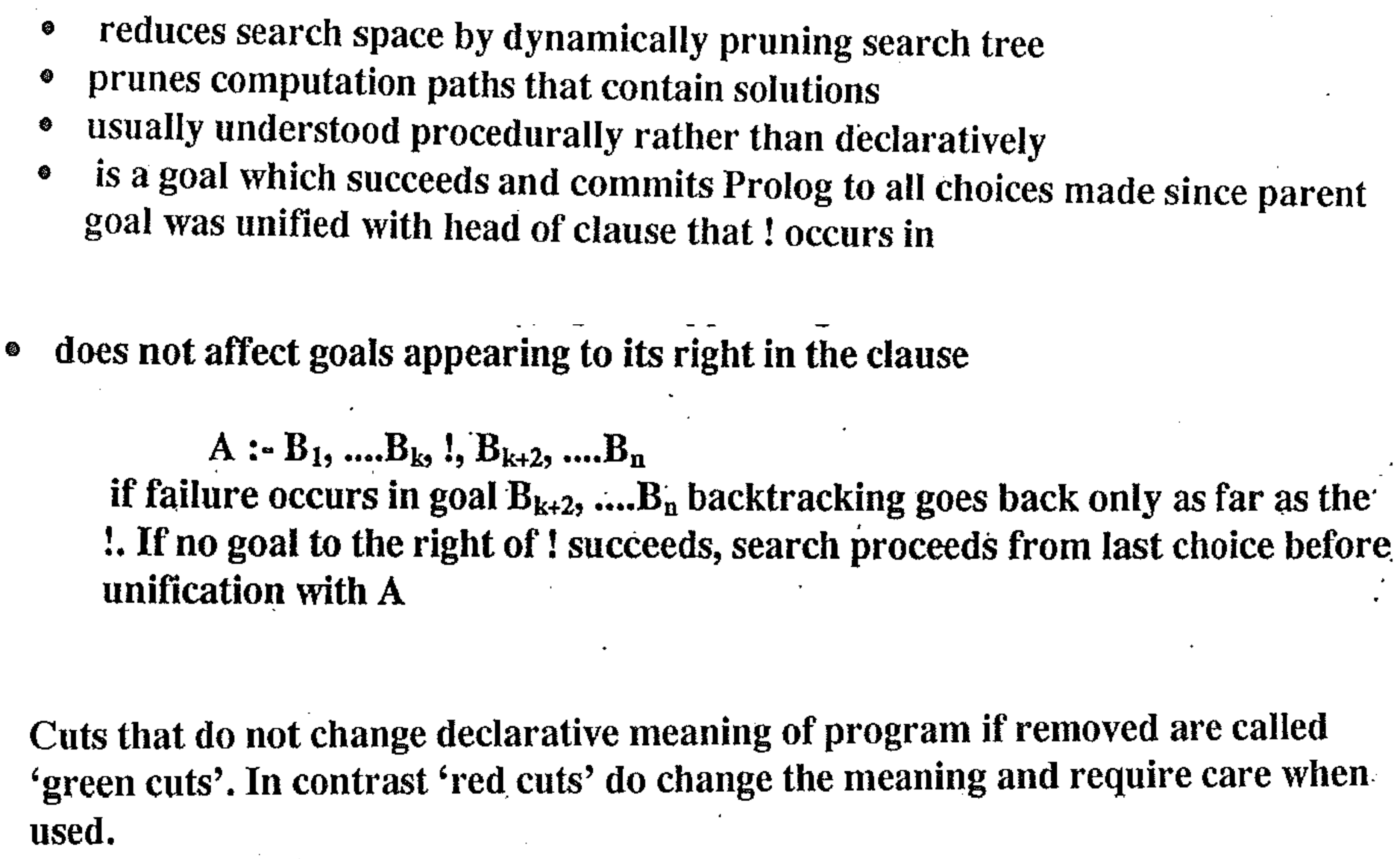
Cuts, Negation

Input/output

Set operations

Style and layout

The cut !



- reduces search space by dynamically pruning search tree
- prunes computation paths that contain solutions
- usually understood procedurally rather than declaratively
- is a goal which succeeds and commits Prolog to all choices made since parent goal was unified with head of clause that ! occurs in

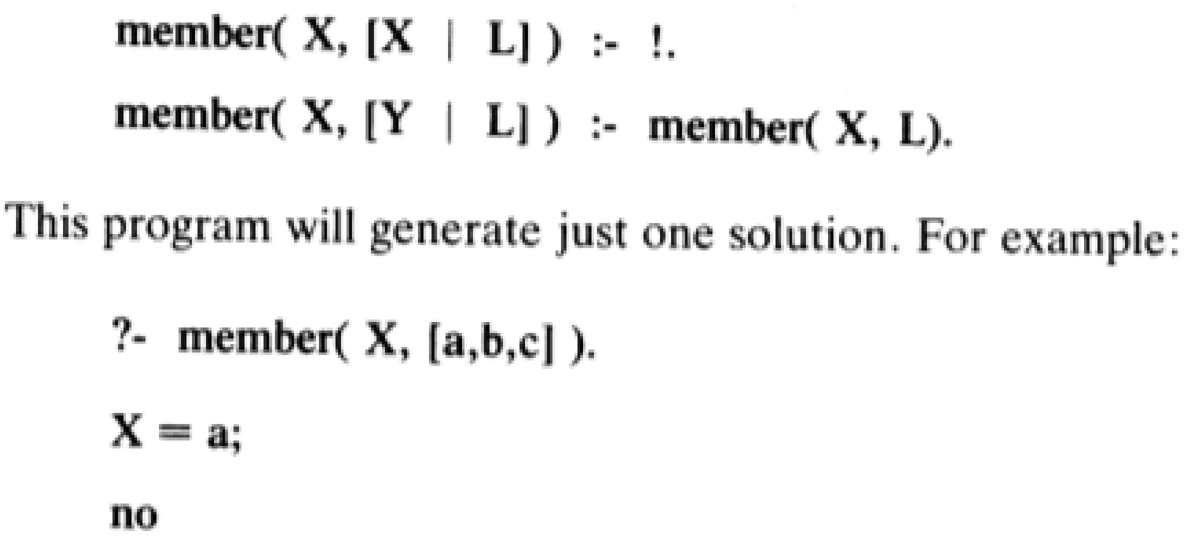
- does not affect goals appearing to its right in the clause

A :- B_1, ...B_k, !, B_{k+2}, ...B_n

if failure occurs in goal B_{k+2}, ...B_n backtracking goes back only as far as the !. If no goal to the right of ! succeeds, search proceeds from last choice before unification with A

Cuts that do not change declarative meaning of program if removed are called 'green cuts'. In contrast 'red cuts' do change the meaning and require care when used.

Example cut: single solution membership of list



```
member( X, [X  |  L] )  :-  !.
member( X, [Y  |  L] )  :-  member( X, L).
```

This program will generate just one solution. For example:

```
?-  member( X, [a,b,c] ).
X = a;
no
```

Negation as Failure

A goal G fails **not(G) succeeds if G cannot be derived**

- Search tree is finitely failed if no success nodes or infinite branches

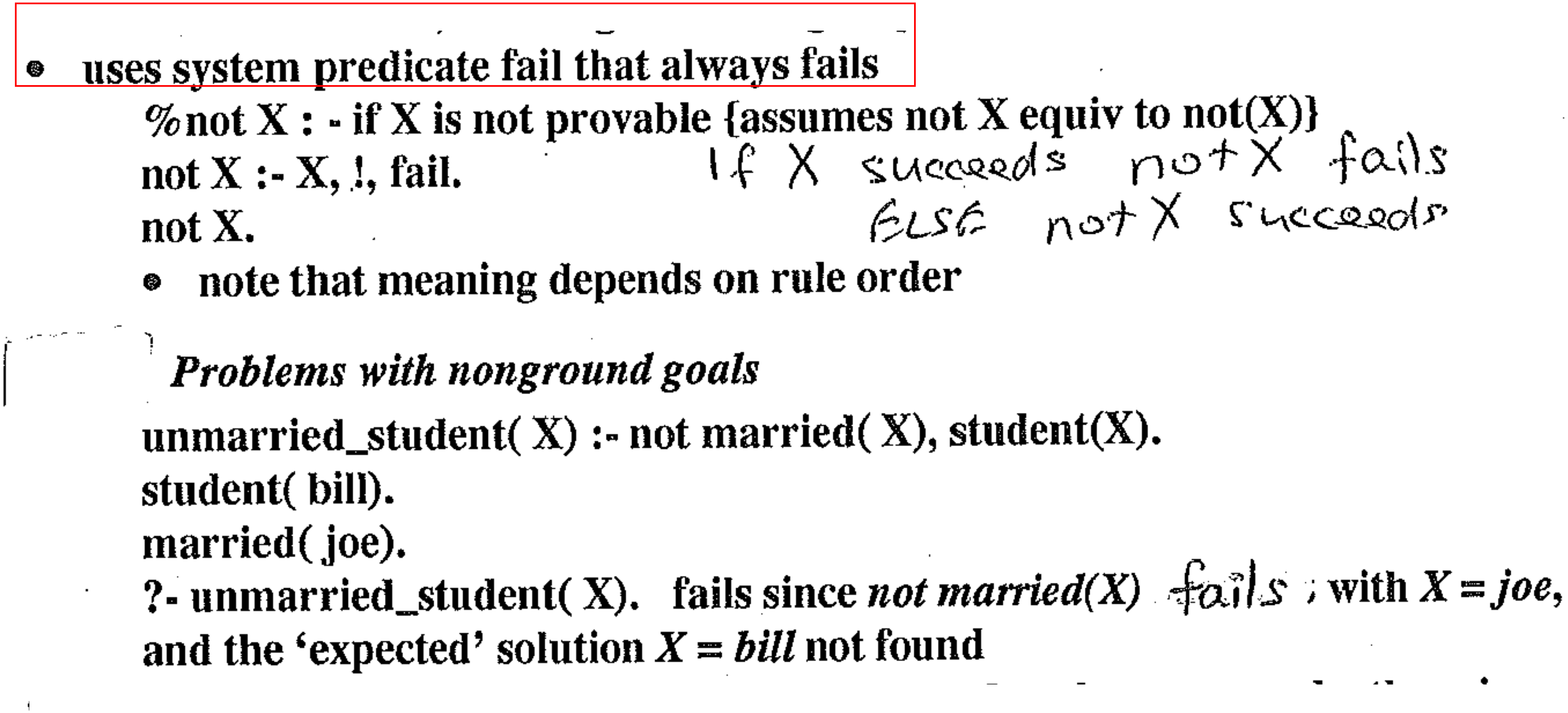
not(G) succeeds if G is in finite failure set

Example:
```
loves (john, mary).
loves (jim, mary).

?- not (loves( ken, mary)).
YES
```

- negation as failure follows from closed world assumption

In effect, if G not in the database - not(G) returns yes

Negation as failure using the red cut



- uses system predicate fail that always fails

```
%not X : - if X is not provable {assumes not X equiv to not(X)}
not X :- X, !, fail.
not X.
```

If X succeeds not X fails
ELSE not X succeeds

- note that meaning depends on rule order

Problems with nonground goals

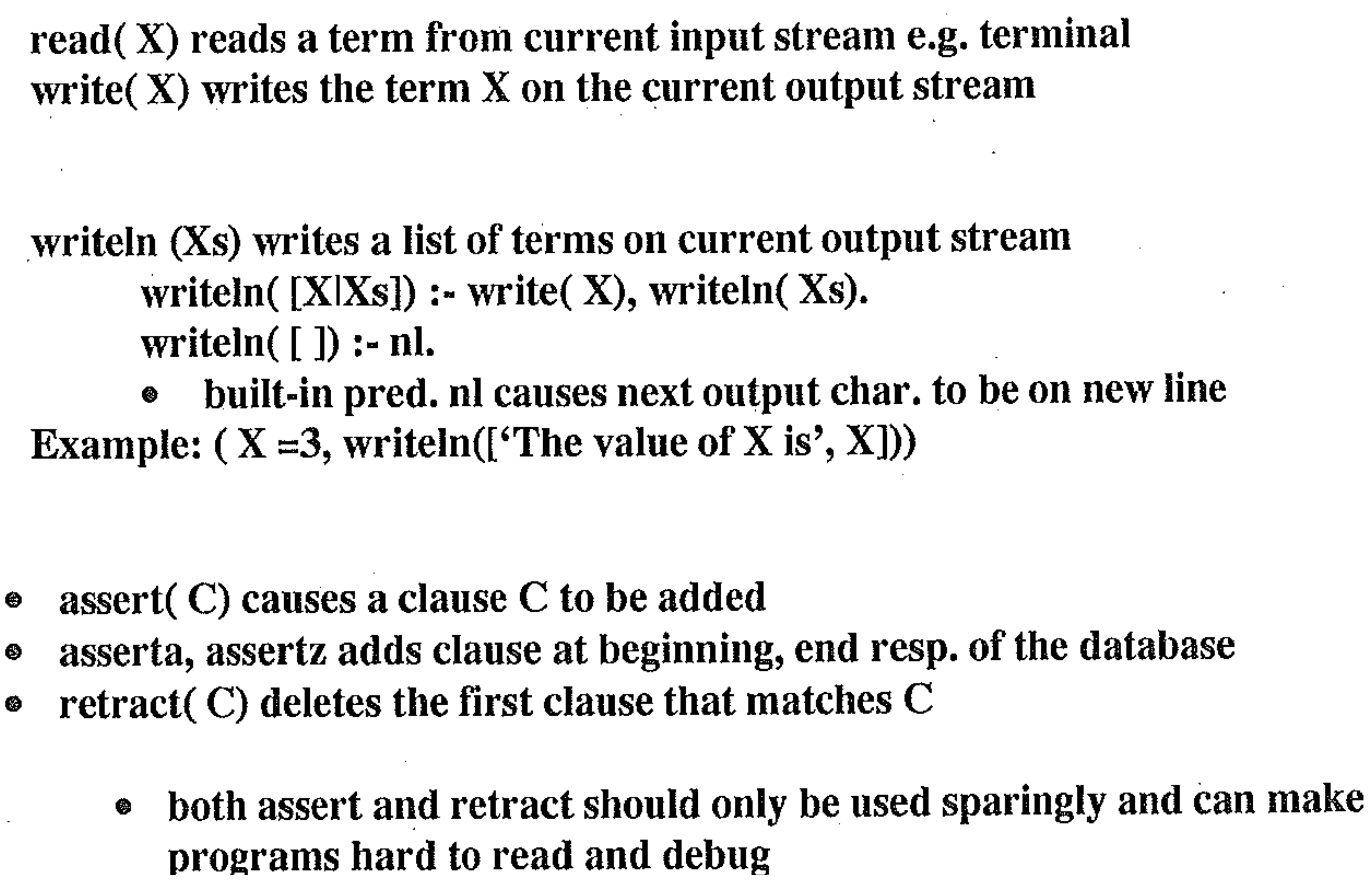
```
unmarried_student( X) :- not married( X), student(X).
student( bill).
married( joe).
```

?- unmarried_student(X). fails since *not married(X)* fails ; with *X* = *joe*, and the ‘expected’ solution *X* = *bill* not found

Problem is that unmarried_student(bill) succeeds

SWI Prolog alternative to **not X**
\+ X True if X is a goal that cannot be proven (mnemonic: + refers to provable)

Input/Output and modifying database



read(X) reads a term from current input stream e.g. terminal
write(X) writes the term X on the current output stream

writeln (Xs) writes a list of terms on current output stream

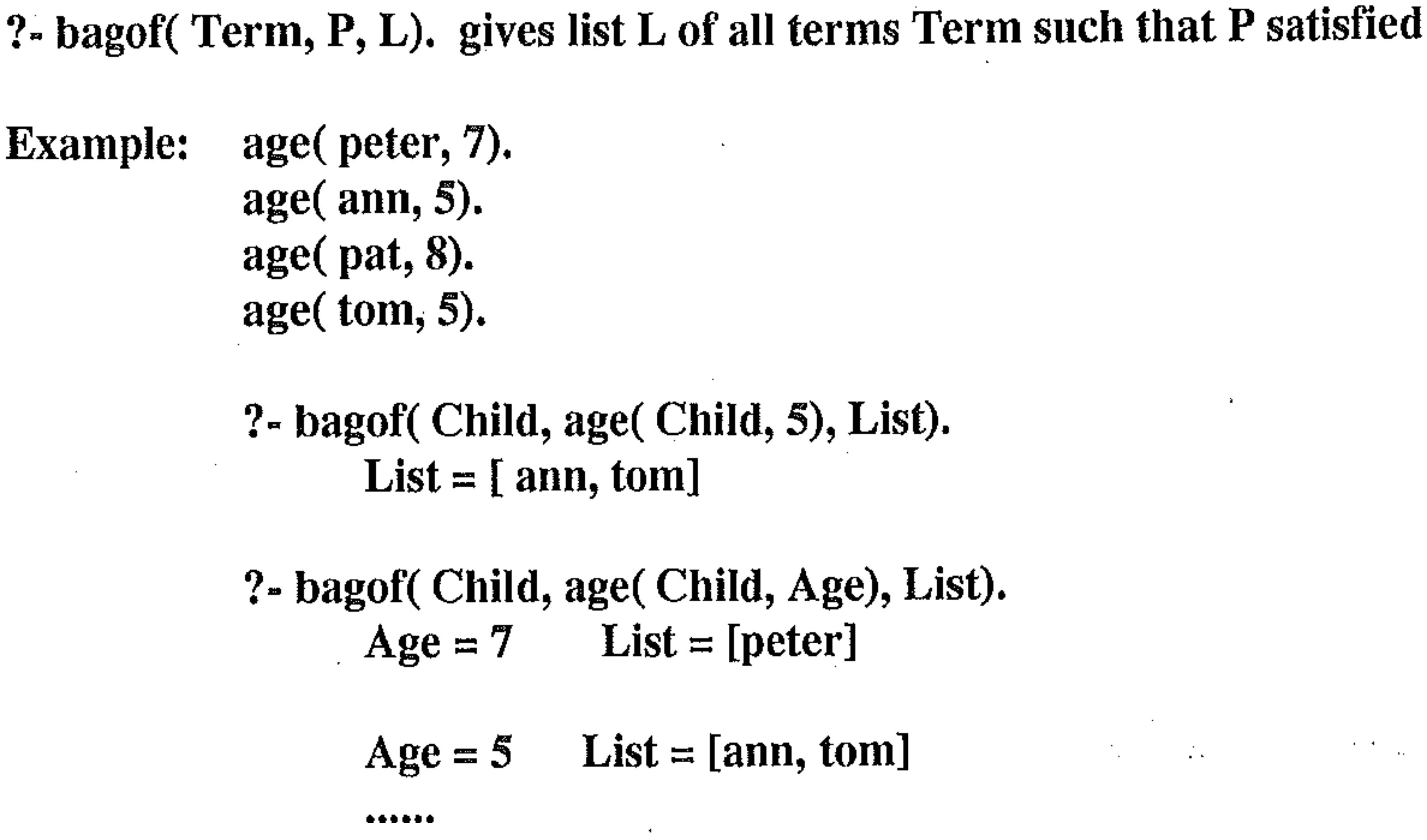
```
        writeln( [X|Xs]) :- write( X), writeln( Xs).
        writeln( [ ]) :- nl.
```

- built-in pred. nl causes next output char. to be on new line

Example: (X =3, writeln([‘The value of X is’, X]))

- assert(C) causes a clause C to be added
- asserta, assertz adds clause at beginning, end resp. of the database
- retract(C) deletes the first clause that matches C
    - both assert and retract should only be used sparingly and can make programs hard to read and debug

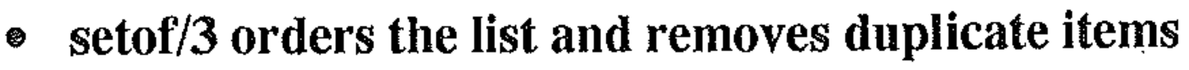
Useful set predicate



?- bagof(Term, P, L). gives list L of all terms Term such that P satisfied

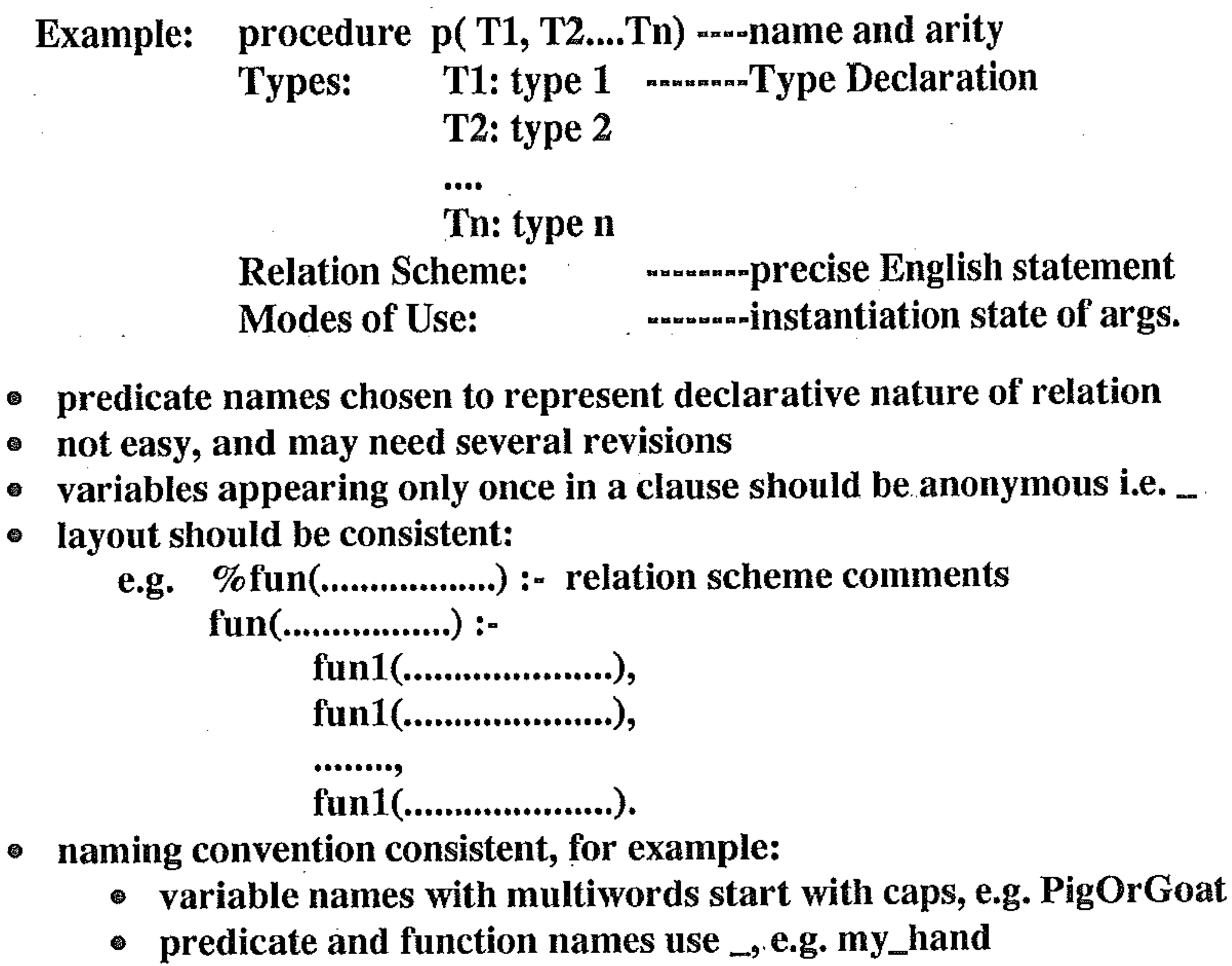
```
Example:    age( peter, 7).
            age( ann, 5).
            age( pat, 8).
            age( tom, 5).

            ?- bagof( Child, age( Child, 5), List).
                    List = [ ann, tom]

            ?- bagof( Child, age( Child, Age), List).
                    Age = 7       List = [peter]

                    Age = 5      List = [ann, tom]
                    ......
```

- findall in SWI Prolog similar to bagof



- setof/3 orders the list and removes duplicate items

Specification and Style (for assignment)



```
Example:   procedure  p( T1, T2....Tn) ----name and arity
           Types:      T1: type 1   ---------Type Declaration
                       T2: type 2
                       ....
                       Tn: type n
           Relation Scheme:         ---------precise English statement
           Modes of Use:            ---------instantiation state of args.
```

- predicate names chosen to represent declarative nature of relation
- not easy, and may need several revisions
- variables appearing only once in a clause should be anonymous i.e. _
- layout should be consistent:

```
    e.g.   %fun(..................) :-  relation scheme comments
           fun(.................) :-
                   fun1(....................),
                   fun1(....................),
                   ........,
                   fun1(....................).
```

- naming convention consistent, for example:
  - variable names with multiwords start with caps, e.g. PigOrGoat
  - predicate and function names use _, e.g. my_hand

- initial comments can include:
    - what the program is about, how to use it, perhaps with exs.
    - identity of top-level predicates
    - how main concepts or objects are represented

- Some heuristics of good style:
    - clauses short, consisting of a few goals
    - use with care:
        - cut operator, red cuts within clearly defined constructs
        - not procedure, especially if variables not instantiated
        - assert/retract, with purpose well documented
    - Logical Or (;) replaced by separate clauses where possible