

## Example of assert: int\_puzzle

- recall `int_puzzle(High, Number, List)`: unique integers between 1 and High that add up to Number are returned in List
  - problem: permutations of solutions were returned
- Solution: keep track of previous solutions
  - Do this by sorting a potential solution, and returning it only if it wasn't returned before
  - sorted integer list represents all its permutations

## int\_puzzle version 2

?- dynamic oldsolution/1. % needed in Sicstus and others:  
% any pred asserted or retracted

```
int_puzzle(High, Number, SortedSoln) :-  
    retractall(oldsolution(_)), % deleted previous runs solutions  
    make_intlist(1, High, All),  
    make_soln(Number, All, Unsorted), % (... as before)  
    sort(Unsorted, SortedSoln),  
    check_unique_and_save(SortedSoln).
```

```
check_unique_and_save(Soln) :-  
    \+ oldsolution(Soln), % if used before, fail  
    assertz(oldsolution(Soln)). % save and return
```

## int\_puzzle: another variation

- This variation lets you find the unique solutions by finding the list of all solutions all at once. No assert is needed...

```
int_puzzle(High, Number, AllSolns) :-  
    int_puzzle2(High, Number, [ ], AllSolns).
```

```
int_puzzle2(High, Number, OldSolns, AllSolns) :-  
    make_intlist(1, High, All),  
    make_soln(Number, All, Unsorted),  
    sort(Unsorted, SortedSoln),  
    \+ member(SortedSoln, OldSolns),  
    int_puzzle(High, Number, [SortedSoln|OldSolns], AllSolns).  
int_puzzle2(_, _, AllSolns, AllSolns).
```

## One more int\_puzzle variation

- This version does the same as the previous, except the builtin setof untility is used:
- setof(X, Goal, L) executes Goal exhaustively, saving the values of X, and putting them in list L

```
int_puzzle(High, Number, Solns) :-  
    setof(List, int_puzzle2(High, Number, List), Solns).
```

```
int_puzzle2(High, Number, SortedList) :-  
    make_intlist(1, High, All),  
    make_soln(Number, All, List),  
    sort(List, SortedList).
```

# Testing & manipulating terms

- Here are a selection of builtin's; see your implementation manual for more...

**var(X)** - succeeds if argument X is an uninstantiated variable

**nonvar(X)** - succeeds if argument X is instantiated to something

- eg. **nonvar(s(1,X))** succeeds

**atom(X)** - succeeds if X is a non-numeric constant

**integer(X)** - succeeds if X is an integer

**ground(X)** - succeeds if X is instantiated to something that has no uninstantiated variables

- eg. **ground(s(1,X))** fails, but **ground(s(1,2))** succeeds

**functor(T, F, N)**: succeeds if term T has functor name F and arity N

- eg. **functor(f(a,2,v(F)), F, N) --> F=f, N=3**

- **functor([a,b,c], F, N) --> F = '.', N = 2** (because **[a,b,c] = ','(a, [b,c])**)

# Manipulating terms

**F =.. L : (called 'univ') breaks term F into a list L, where first list member is functor name, and tail are arguments**

– also works other way: given list, create the term

– eg. **s(a,b,c) =.. L --> L = [s, a, b, c]**

– **T =.. [append,a,[ ], eee(4)] --> T = append(a, [ ], eee(4) )**

**% a predicate that adds an argument value to any given term**

**add\_arg(Term, Arg, Newterm) :-**

**Term =.. List,**

**append(List, [Arg], List2),**

**Newterm =.. List2.**

**?- add\_arg(parent(john, bill), zebra, T).**

**T = parent(john, bill, zebra)**

## Manipulating structures

**name(A, X) - converts atom A into list of ascii values (integers); works backwards too**

- name(apple, X) --> X = [97, 112, 112, 108, 101]**
- name(A, [108, 101, 97, 112]) --> A = leap**
- name(A, "leap") --> A = leap**
- Note: string notation is a shorthand for list of ascii values**

**% a predicate that takes an atom, and numbers it via a global counter**

**% we presume counter is initialized to '1'**

```
set_tag(A, NewA) :-  
    retract(count(N)),  
    name(N, Nlist),  
    name(A, Alist),  
    append(Alist, Nlist, NewAlist),  
    name(NewA, NewAlist),  
    N2 is N + 1,  
    assert(count(N2)).
```

# negation

**`\+ X` - succeeds if X fails when called; fails if X succeeds**

- **This is not a true implementation of logical negation, but is “close enough” for programming purposes**
- **permits new programs to be written more concisely; without ‘not’, you’d have to rewrite a lot of code**

**% rem\_dups(A, B) remove the duplicates in list A: version 1**

**rem\_dups(A, B) :- rem\_dups2(A, [ ], B).**

**rem\_dups2([ ], B, B).**

**rem\_dups2([A|T], B, C) :- \+ member(A, B), rem\_dups(T, [A|B], C).**

**rem\_dups2([\_|T], B, C) :- rem\_dups(T, B, C).**

**% version 2**

**rem\_dups([ ], [ ]).**

**rem\_dups([A|T], [A|C]) :- \+ member(A, T), rem\_dups(T, C).**

**rem\_dups([\_|T], C) :- rem\_dups(T, C).**

# Equality

- You've seen unification: =
- Other types of equalities:

**$X == Y$ : succeeds if X and Y are identically equal**

- == will not unify them
- if == succeeds, then = would have succeeded; but not necessarily vice versa

eg. ?- X == cat --> fails

?- X = a, Y = b, s(a,Y) == s(X,b). --> succeeds

?- s(a,Y) == s(X,b). --> fails

**$X \neq Y$ : same as not(X=Y)**

**$X \neq= Y$ : same as not(X == Y)**

**$X ::= Y$  : numeric values of X and Y are equal**

- same as: T is X, T is Y. (X, Y must be fully instantiated!)

**$X \neq= Y$ : numeric values are not equal**

# Control

- **Prologs left-to-right goal selection and first-to-last clause selection are limited in many programming situations**
- **There are various tools that let you control Prolog execution more precisely**
- **Some tools are simple to use; others can lead to trouble**

## **1. iteration (“failure-driven loops”)**

**repeat - a builtin that always succeeds when backtracked to (same as ‘true’)**

**fail - a builtin that always fails**

- **let you write conventional-style do-loops**
- **can lead to messy programs if you aren’t careful!**
- **can also lead to infinite loops: ?- repeat, write(‘boo!’), fail.**

# Failure-driven loops

**% program which reads a command from user from a limited selection,  
% and performs that task.**

**driver :-**

```
repeat,  
write('Commands: [listing, system, clear, end] ?'),  
read(C),  
process(C).
```

**process(listing) :- listing, fail.**

**process(system) :- write('unix:'), read(X), system(X), fail.**

**process(clear) :- cls, fail.**

**process(end) :- write('Goodbye.').**

```
process(X) :- \+ member(X, [listing, shell, clear, end]),  
write('Unrecognized command.'). fail.
```

# Control

## 2. Disjunction (logical 'or'):

- **(P ; Q) - P is first executed; backtracking out of reverts to Q**
- **this can always be done via multiple clauses; however, it is often useful if these clauses share a lot of common goals**
- **if you use this too much, your programs become very hard to read and debug**

**% an error occurs if a list is empty, or has > 100 members, or contains an integer; otherwise it's OK**

```
test_list(L) :- ( L = [ ] ; length(L) > 100 ; (member(X,L), integer(X)) )  
                write('Bad list, '), nl, fail.
```

```
test_list(L) :- \+ ( L = [ ] ; length(L) > 100 ; (member(X,L), integer(X)) ),  
                write('Good list. '), nl.
```

# Disjunction

```
test_list([ ]) :- write('Bad list, '), nl, fail.
```

```
test_list(L) :- length(L) > 100, write('Bad list, '), nl, fail.
```

```
test_list(L) :- member(X,L), integer(X), write('Bad list, '), nl, fail.
```

```
test_list(L) :- length(L) >= 1, length(L) =< 100,  
                \+ (member(X,L), integer(X)),  
                write('Good list. '), nl.
```

- Note how the 'good test' must perform all the checks all over again.
  - this can be very expensive... (see next slide)
- Note how backtracking works:

```
parent(X, Y) :- (mother(X,Y) ; father(X,Y) ).
```

same as....  
mother

```
parent(X,Y) :- mother(X,Y).
```

```
parent(X,Y) :- father(X,Y).
```

father is reached only after

exhausted

## Control: if-then-else

3. implication or if-then-else:  $(P \rightarrow A ; B)$   
 $(P \rightarrow A)$

- in  $P \rightarrow A ; B$ :
  - (i)  $P$  is executed until it either succeeds or fails
  - (ii) if  $P$  succeeded, then execute  $A$   
otherwise if  $P$  failed, then execute  $B$
  - (iii) backtracking within  $A$  and  $B$  work as usual; however, backtracking will never revert back to  $P$ , nor from  $A$  to  $B$
- $P$  directs to control to either  $A$  or  $B$ , and backtracking to  $P$  disappears
- $P \rightarrow A$  : like the above, except that if  $P$  fails, whole expression fails; and failure of  $A$  will not cause backtracking into  $P$

# Control: if-then-else

**% rewrite previous example:**

**test\_list(L) :-**

```
( L = [ ] ; length(L) > 100 ; (member(X,L), integer(X)) )
```

```
->
```

```
(write('Bad list, '), nl, fail) % note: fail here makes whole clause fail
```

```
;
```

```
(write('Good list. '), nl).
```

- **note the use of parentheses to group goals**
  - **not always required, but it prevents misunderstandings**
- **good idea to use clear indentation to improve readability**
- **it is possible to nest ( ; -> not , ), and if you do it too much, your program will be unintelligible**
- **if you find yourself writing lots of nested constructs**
  - > create new predicates!**

# Control

**% yet another rewrite:**

```
test_list(L) :-  
    test_if_bad(L)  
    ->  
    write('Bad list, '), nl, fail  
    ;  
    write('Good list'), nl.
```

```
test_if_bad([ ]).  
test_if_bad(L) :-  
    length(L) > 100.  
test_if_bad(L) :-  
    member(X, L),  
    integer(X).
```

- **probably better style to put bad tests into separate clauses**