

# Debugging Prolog

- **advantage of interpreted language: readily available debugging facilities!**
- **trace - step-by-step view of execution**
  - ?- trace.     % trace on
  - ?- notrace.   % turn off trace
- **4 types of events are shown:**
  - 1. **CALL** : is about to invoke a new goal/ match a new clause
  - 2. **RETRY** : is doing backtracking(match another clause)
  - 3. **EXIT** : a predicate call is successful, and is returning
  - 4. **FAIL** : a goal cannot be satisfied, and backtracking goes to another part of tree

# Debugging

- **typical commands when in trace mode:**
  - **c or ENTER:creep** - single stepper (call goal, or exit predicate)
  - **s** : skips entering that goal - calls it, and returns after goal completed
  - **l** : leap - resumes execution, stopping at a spy point
  - **f** : forces that goal to fail
  - **g**: lists the ancestors (waiting goals)
  - **h**: help

**?- spy pred.** : as soon as pred is called, it puts you into trace mode  
– turn off: **nospy**

**?- debugging.** : shows the predicates being spied upon

**?- nodebug.** turns off all spy points

# Example trace

```
plan_date(Guy, Gal, Food) :-  
    likes(Guy, Food), likes(Gal, Food).
```

```
likes(tom, sushi).  
likes(tom, pasta).  
likes(tom, bbq).  
likes(sue, pasta).  
likes(sue, bbq).
```

| ?- trace.

yes

| ?- plan\_date(tom, sue, X).

```
1 1 Call: plan_date(tom,sue,_87) ?  
2 2 Call: likes(tom,_87) ?  
2 2 Exit: likes(tom,sushi) ?  
3 2 Call: likes(sue,sushi) ?  
3 2 Fail: likes(sue,sushi) ?  
2 2 Redo: likes(tom,sushi) ?  
2 2 Exit: likes(tom,pasta) ?  
3 2 Call: likes(sue,pasta) ?  
3 2 Exit: likes(sue,pasta) ?  
1 1 Exit: plan_date(tom,sue,pasta) ?
```

X = pasta? ;

```
1 1 Redo: plan_date(tom,sue,pasta) ?  
3 2 Redo: likes(sue,pasta) ?  
3 2 Fail: likes(sue,pasta) ?  
2 2 Redo: likes(tom,pasta) ?  
2 2 Exit: likes(tom,bbq) ?  
3 2 Call: likes(sue,bbq) ?  
3 2 Exit: likes(sue,bbq) ?  
1 1 Exit: plan_date(tom,sue,bbq) ?
```

X = bbq ? ;

```
1 1 Redo: plan_date(tom,sue,bbq) ?  
3 2 Redo: likes(sue,bbq) ?  
3 2 Fail: likes(sue,bbq) ?  
2 2 Redo: likes(tom,bbq) ?  
2 2 Fail: likes(tom,_87) ?  
1 1 Fail: plan_date(tom,sue,_87) ?
```

no

## Other debugging techniques

- **Pay attention to message: “singleton variables” (paraphrase)**
  - means that a variable has been defined in only one place in a predicate
  - could be safely replaced by an anonymous variable: `_`
  - message useful because you might have misspelled it

**grandfather(Granddad, Grandkid) :-  
parent(Grandfather, parent),  
parent(Parent, Grandkid).**

**--> singleton defined: Granddad, Grandfather, Parent**

- **good old “write” debug statements within your program**
  - useful when tracing gives too much information
- **always debug modularly**
  - predicates can be tested by themselves before being used

# Returning values

- **Arguments in predicates used to read input values and return output back**

- Prolog differs from most languages in that arguments can be both input and output, depending on calling pattern to predicate

- eg. `?- plan_date(tom, sue, Food).`

- `?- plan_date(Guy, Gal, sushi).`

- `?- plan_date(tom, Gal, bbq).      etc.`

- **In a typical problem, you may want to return an answer in response to some other input**

- eg. `% divisible(X, Y, Ans)    Ans is 'yes' if X is divisible by Y; else 'no'`

- `divisible(X, Y, Ans) :-`

- `X is integer(X/ Y) * Y,`

- `Ans = yes.`

- `divisible(X, Y, Ans) :-`

- `\+ (X is integer(X/Y)*Y),`

- `Ans = no.`

- note: doesn't matter where 'Ans=yes' is put in the body.

# Returning values

- **Better version: pass data directly using unification**

`divisible(X, Y, yes) :-`

`X is integer(X/Y) * Y.`

`divisible(X, Y, no) :-`

`\+ (X is integer(X/Y)*Y).`

- **Another version: if divisible succeeds, then numbers are divisible...**

`divisible(X, Y) :-`

`X is integer(X/Y) * Y.`

- **Could add more error checking too...**

`divisible(X, Y) :-`

`number(X), number(Y), Y \= 0,`

`X is integer(X/Y)*Y.`

# Recursion

- **some (most!) problems can be easily solved by defining a solution that refers to itself**
- **recursive predicate: one that calls itself in a rule body**
  - recursion may be indirect: predicate p may call q, which calls p
- **characteristics of recursive algorithms:**
  1. **exist exit condition(s) which terminate the recursion**
    - eg. have processed to a limit value, or an empty data structure
  2. **intermediate expression which defines a partial solution in terms of the recursive program itself**
- **recursion is naturally done in Prolog**
  - eg 'family tree' question of assignment 1

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

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

# Recursion

- As with any goal, each recursive goal:
  - a) calls a 'fresh copy' of the predicate, ie. all logic variables in the clauses to be called are renamed
  - b) attempts unification of the goal with the clause head
  - c) during backtracking, these variables are 'undone' and next clause is tried

**Example: integer exponents : compute  $y = x^n$  ( $n \geq 0$ )**

```
%  $y = x^n = x * [x^{(n-1)}]$ 
```

```
exp(1, _, N) :- N =< 0. ← exit case
```

```
exp(Y, X, N) :-
```

```
    N > 0,
```

```
    N2 is N - 1,
```

```
    exp(Y2, X, N2), ← recursive call
```

```
    Y is X * Y2.
```

## A somewhat “buggy” version of exp

```
exp(1, _, 0).  
exp(Y, X, N) :-  
    N2 is N - 1,  
    exp(Y2, X, N2),  
    Y is Y2 * X.
```

- “Hey, but what’s wrong with it!?!?”

Let’s run it...

?- exp(Y, 5, 3).

Y = 125 ;

(oops... never comes back!)

- problem: on backtracking from exit case, negative N occurs in recursion and never is caught by exit case
- likewise , all negative ‘n’ cause infinite recursion

## More recursion: Boolean logic

**boolean(t, t).**

**boolean(f, f).**

**boolean(and(A, B), Out) :-**

**boolean(A, OutA),**

**boolean(B, OutB),**

**and\_table(OutA, OutB, Out).**

***/\* similarly for Or, Not \*/***

**and\_table(f, f, f).**

**and\_table(f, t, f).**

**and\_table(t, f, t).**

**and\_table(t, t, t).**

***/\* similarly for Or, Not \*/***

# Boolean logic

- **boolean/2: 1st arg is boolean expression, 2nd arg is boolean value**
- **we will permit variable arguments to expressions**
  - eg. ?- boolean(and(and(X, t), Y), Out).
- **boolean/2 recursively decomposes expression until it reduces to either t, f, or a variable**
  - t, f terms returned as their own values; variable will unify to t, f, and an and term
- **problem: variable terms are useful in some cases (can generate entire truth tables)**
  - but a variable term will unify with AND, and this causes infinite recursion in the 2nd clause for boolean
- **solution: only permit unification with 'and' term for non-variable args**

```
boolean(X, Out) :-      % new 3rd clause:
    nonvar(X),
    X = and(A, B),
    boolean(A, OutA), boolean(B, OutB), and_table(OutA, OutB, Out).
```

# Lists revisited

- Recall builtin list data structure:

[ ] - empty list

[ a, b, c, d ] - list with 4 elements

[ H | T ] - list with first element H, tail T

– eg. [ a, b, c, d ] = [ H | T ] ---> H=a, T= [b,c,d]

- Recursion on lists is straight-forward

% member(M, L) succeeds if element M is in list L

member( X, [ X | T ] ). % found it!

member( X, [ Y | T ] ) :- member( X, T ). %recurse on tail

note: can rewrite as:

member( X, [ X | \_ ] ).

member( X, [ \_ | T ] ) :- member( X, T ).

# Lists

```
member( X, [ X | _ ] ).  
member( X, [_ | T] ) :- member( X, T ).
```

**Note that this program is equivalent to:**

```
member( X, Y ) :- Y = [X | _].  
member( X, Y ) :- Y = [_ | T], member( X, T ).
```

- **We can always put terms in predicate heads as goals**
- **However no advantage, and in fact makes program less efficient (more goals to execute)**
- **Letting program execution perform unification is preferred.**

## Example Prolog programs

?- member(a, [b, c, d]).

no

?- member(c, [b, c, d]). ← note 2 different uses of member!

yes

?- member (X, [a, b, c]). ←

X = a ;

X = b ;

X = c ;

no

?- member(A, [a, b, c]), member(A,[ b, c, d]). ( hmmm... like set intersection)

A = b ;

A = c ;

no

?- member(a, L).

L = [ a | \_A] ;

L = [\_A, a | \_B] ;

L = [\_A, \_B, a | \_C] ; etc

## List programs: append

`append( [ ], L, L).`

`append( [X | R] S, [X | T]) :- append(R, S, T).`

`?- append([ a, b, c], [1, 2, 3], L).`

`L = [a, b, c, 1, 2, 3] .`

`?- append( Y, [1, 2, 3], [a, b, c, 1, 2, 3]).`

`Y = [a, b, c]`

`?- append(X, Y, [1,2,3]).`

`X=[ ], Y = [1, 2, 3] ;`

`X=[1], Y = [2, 3] ;`

`X=[1,2], Y = [3] ;`

`X = [1,2,3], Y = [ ];`

`no`

`?- append( _, [Y|_], [1, 2, 3]). % a way to use append like 'member'!`

`Y = 1; Y = 2; Y = 3; no`

## Testing out list notation

?- [a, b] = [X, Y].

X = a, Y = b

?- [a, b] = [X | Y].

X = a, Y = [b]

?- [a] = [X | Y].

X = a, Y = [ ]

?- [a, b, c, d] = [X, Y | Z].

X = a, Y = b, Z = [c, d, e]

?- [a, b, c, d] = [X, Y | W, Z].

\*\*\* ERROR \*\*\* (bad syntax in RHS)

?- X = [a, b, c, d], Y = [cat | X].

X = [a,b,c,d], Y = [cat,a,b,c,d]

?- Y = [a | Y].

Y = [a,a,a,a,a,a,a,.... □ ]

## Lists: set programs

**intersection([ ], \_, [ ]).**

**intersection([X | R], Y, [X | Z]) :-**

**member(X, Y),**

**intersection(R, Y, Z).**

**intersection([X | R], Y, Z) :-**

**not member(X, Y),**

**intersection(R, Y, Z).**

**union([ ], X, X).**

**union([X | R], Y, Z) :-**

**member(X, Y),**

**union(R, Y, Z).**

**union([X | R], Y, [X | Z]) :-**

**not member(X, Y),**

**union(R, Y, Z).**

# Lists: processing elements

- **General pattern:**
- **list\_proc([ ],TermValue). % exit for recursion**  
**list\_proc([Item | Tail], Soln) :-**  
    **{do processing on Item, obtaining Soln1},**  
    **list\_proc(Tail, Soln2),**  
    **{combine Soln1 and Soln2 to get Soln}.**
- **eg. Add up the elements in a list**

```
sum([ ], 0).  
sum([N | T], Sum) :-  
    sum(T, Sum2),  
    Sum is Sum2 + N.
```

# Lists: processing elements

- eg. double all the numbers in a list

```
double([ ], [ ]).
```

```
double([N | T], Soln) :-  
    N2 is N*2,  
    double(T, Soln2),  
    Soln = [N2 | T2].
```

or better...

```
double([ ], [ ]).
```

```
double([N | T], [N2|T2]) :-  
    N2 is N*2,  
    double(T, Soln2).
```

## Generating lists as results

- example: given an integer  $N > 0$ , create a list of integers from 0 to  $N$

```
intlist(N, L) :- N > 0, make_intlist(0, N, L).
```

```
make_intlist(M, N, [ ]) :-
```

```
    M > N.                                % exit case
```

```
make_intlist(M, N, L) :-
```

```
    M =< N,
```

```
    M2 is M + 1,                            % bump up M
```

```
    make_intlist(M2, N, L2),                % make list for M+1,...,N
```

```
    append( [M], L2, L).                    % add M to start of it
```

- if we remove test “ $M =< N$ ”, then the first solution returned is correct; but backtracking will cause additional numbers  $> N$  to be added.

# Generating Lists as results

version 2:

```
make_intlist(M, N, [ ]) :-
```

```
    M > N.
```

```
    % exit case
```

```
make_intlist(M, N, L) :-
```

```
    M =< N,
```

```
    M2 is M + 1,
```

```
    % bump up M
```

```
    make_intlist(M2, N, L2),
```

```
    % make list for M+1,...,N
```

```
    L = [M | L2].
```

```
    % 'append' is overkill
```

# Generating Lists as results

version 3:

```
make_intlist(M, N, [ ]) :-  
    M > N.
```

```
make_intlist(M, N, [M | L2]) :-  
    M =< N,  
    M2 is M + 1,  
    make_intlist(M2, N, L2).
```

- Note how unification initially binds the answer list with `[M|L2]`, even though `L2` doesn't have a value at first
- However, after `make_intlist` succeeds, `L2` will have a value
- `L2`'s result automatically gets transferred to whatever unified with `[M|L2]`

# Generating Lists as Output

version 5:

```
intlist(N, L) :- make_intlist(0, N, [ ], L).
```

```
make_intlist(M, N, L, L) :- M > N.
```

```
make_intlist(M, N, OldL, L) :-  
    M =< N,  
    M2 is M + 1,  
    make_intlist(M2, N, [M | OldL], L).
```

- This version starts with an empty list, and adds an item in each recursive call. This list is returned as an answer in the exit case.
- note: `[M | OldL]` is a shortcut for “`append([M], OldL, L2)`”.

## Another list example: even or odd list?

**% even succeeds if there are an even number of entries in a list**

**even([ ]).**

**even([\_, \_|T]) :- even(T).**

**% odd succeeds if there are an odd number of list entries**

**odd([ \_ ]).**

**odd([\_, \_ | T]) :- odd(T).**

**% an alternate version of odd...**

**odd2(List) :- \+ even(List). % not even? then it's odd!**

**% even\_odd gives descriptive results...**

**even\_odd(List, even) :- even(list).**

**even\_odd(List, odd) :- odd(list).**

## List example: reverse a list

```
reverse([ ], [ ]).
```

```
reverse([A|T], Reversed) :-  
    reverse(T, TRev),  
    append(TRev, [A], Reversed).
```

% an alternate, more efficient reverse...

```
reverse2(List, Reversed) :-  
    rev2(List, [ ], Reversed).
```

```
rev2([ ], Rev, Rev).
```

```
rev2([A|T], L, Rev) :- rev2(T, [A|L], Rev).
```

## Another list example: deletion

***/\* delete(A, L, M) deletes the first instance of element A from L,  
resulting in M \*/***

```
delete(A, [A|T], T).                % found A, so return the tail  
delete(A, [B|T], [B|T2]) :- delete(A, T, T2). % del A from rest, but add B  
delete(_, L, L).                    % A not found: return L
```

***/\* delete\_all(A, L, M) deletes all of A from L, giving M***

```
delete_all(A, [A|T], T2) :- delete_all(A, T, T2).  
delete_all(A, [B|T], [B|T2]) :- delete(A, T, T2).  
delete_all(_, L, L).
```