



# CSC384: Intro to Artificial Intelligence

## Prolog Tutorials 3&4

# Debugging Programs in Prolog

- We talked about the graphical debugger under Windows.
- Now, the text-based debugger:
  - You can put a breakpoint on a predicate:
    - ?- `spy(female)`.
  - You can remove a breakpoint:
    - ?- `nosp(female)`.
  - To start debugging use “trace” before the query:
    - ?- `trace, male(X)`.
  - While tracing you can do the following:
    - `creap`: **step inside the current goal** (press c/enter/or space)
    - `leap`: **run up to the next spypoint** (press l)
    - `skip`: **step over the current goal without debugging** (press s)
    - `abort`: **abort debugging** (press a)
    - And much more... press h for help

# Text Debugger Example

- `?-spy(female/1).`  
yes
- `?-mother(X,Y). %starts debugging!`  
Call: (9) female(albert) ?
- `?-nospy(female/1).`  
% Spy point removed from female/1
- `trace, father(X,Y). %let's debug!`  
Call: (9) father(\_G305, \_G306) ?
- Let's debug this!

```

male(albert).
male(edward).

female(alice).
female(victoria).

parent(albert,edward).
parent(victoria,edward).

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

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

# List Processing in Prolog

- Much of Prolog's computation is organized around lists. Two key things we do with a list is iterate over them and build new ones.

- Many built-in predicates: **member**, **append**, **length**, **reverse**.

- checking membership:

`member(?X,?Y)` holds iff X is a member of list Y.

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

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

- E.g. building a list of integers in range [i, j].

`build(+from, +to, ?NewList)`

`build (I,J,[ ]) :- I > J.`

`build (I,J,[I | Rest]) :- I =< J, N is I + 1,  
build(N,J,Rest).`

# Lists: extracting elements

- Extracting all elements satisfying a condition:  
`extract(+Cond, +List, ?Newlist)`

`extract(_,[ ], [ ]).`

`extract(Condname, [X | Others], [ X | Rest]):-`

`Cond =.. [Condname,X], %building cond predicate CondName(X)`

`Cond, %now, calling Cond predicate to see if it holds`

`extract(Condname, Others, Rest).`

`extract(Condname, [X | Others], Rest):- Cond =.. [Condname,X],`

`\+ Cond, extract(Condname, Others, Rest).`

- `=..` is used to build predicates on the fly. Note that you cannot call `Condname(X)` directly!
- `\+` is *negation as failure*. We will get back to this soon.

# Lists: append

- Appending two lists:

append(?X, ?Y, ?Z)

Holds iff Z is the result of appending lists X and Y.

Examples:

- *append([a,b,c],[1,2,3,4],[a,b,c,1,2,3,4])*
- Extracting the third element of L: *append([\_,\_,X],\_,L)*
- Extracting the last element of L: *axppend(\_,[X],L)*
- Finding two consecutive elements X&Y in L:  
append(\_,[X,Y|\_],L)



# Implementing append

definition: `append(?X, ?Y, ?Z)`

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

`append([H|T],L,[H|L2]):-`

`append(T,L,L2).`

- What are **all** the answers to `append(_, [X,Y], [1,2,3,4,5])` ?
- What are **all** the answers to `append(X, [a], Y)` ?
- What is the answer to `append(X,Y,Z)`? How many answers?

# Lists: reversing

- Reversing a list:  $[1, 2, [a, b], 3] \rightarrow [3, [a, b], 2, 1]$   
`reverse(?L, ?RevL)`

`reverse([ ], [ ]).`

`reverse([H|T], RevL):-`

`reverse(T, RevT), append(RevT, [H], RevL).`

- This is not efficient! Why?  $O(N^2)$

# Efficiency issues: Fibonacci

- Fibonacci numbers: 
$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n-1) + fib(n-2) & n > 1 \end{cases}$$

- Consider the following implementation:

fib(0,0).

fib(1,1)

fib(N,F):- N>1,

N1 is N-1, fib(N1,F1), N2 is N-2, fib(N2, F2),

F is F1+F2.

- This is very inefficient (exponential time)! Why?
- Solution: use **accumulator!**

# Fibonacci using accumulators

Definition: `fibacc(+N,+Counter,+FibNminus1,+FibNminus2,-F)`

We start at counter=2, and continue to reach N. FibNminus1 and FibNminus2 are accumulators and will be update in each recursive call accordingly.

`fibacc(N,N,F1,F2,F):-` %basecase: the counter reached N, we are done!

`F is F1+F2.`

`fibacc(N,I,F1,F2,F):- I<N,` %the counter < N, so updating F1&F2

`Ipls1 is I +1, F1New is F1+F2, F2New is F1,`

`fibacc(N,Ipls1,F1New,F2New,F).`

- This is  $O(N)$ .
- Now we define `fib(N,F)` for  $N>1$  to be `fibacc(N,2,1,0,F)`.

# Accumulators: reverse

- Efficient List reversal using accumulators:  $O(n)$

```
reverse(L,RevL):-  
    revAcc(L,[ ], RevL).
```

```
revAcc([ ],RevSoFar, RevSoFar).
```

```
revAcc([H|T],RevSoFar, RevL):-  
    revAcc(T,[H|RevSoFar], RevL).
```

# Negation As Failure

- Prolog cannot assert something is false.
- Anything that cannot be proved from rules and facts is considered to be false (hence the name Negation as Failure)
- Note that this is different than logical negation!
- Represented in Prolog by symbols **not** or **\+**
  - **\+** member(X,L) %this holds if it cannot prove X is a member of L
  - **not** (A<B) %this holds if it cannot prove A is less than B
- $X \neq Y$  is shorthand for  $\neq (X=Y)$

# NAF examples

Defining **disjoint sets**:

```
overlap(S1,S2):- %S1 &S2 overlap if they share an element.
```

```
    member(X,S1),member(X,S2).
```

```
disjoint(S1,S2):- \+ overlap(S1,S2).
```

```
?- disjoint([a,b,c],[2,c,4]).
```

no

```
?- disjoint([a,b],[1,2,3,4]).
```

yes

```
?- disjoint([a,c],X).
```

No ←this is not what we wanted it to mean!

# Proper use of NAF

- $\setminus +$  Goal works properly only in the following two cases:
  - When G is fully instantiated at the time of processing  $\setminus +$ . In this case, the meaning is straightforward.
  - When there are uninstantiated variables in G but they do not appear elsewhere in the same clause. In this case, it means there are no instantiations for those variables that make the goal true. e.g.  $\setminus +$ Goal(X) means there is no X such that Goal(X) succeeds.

# Guiding the Search Using Cut !

- The goal “!”, pronounced **cut**, always succeeds immediately but just **once**.
- It has an important side-effect: once it is satisfied, it **disallows** (just for the current call to predicate containing the cut):
  - backtracking **before** the cut in that clause
  - Using **next rules** of this predicate
- So, below, before reaching cut, there might be backtracking on b1 and b2 and even trying other rules for p if b1&b2 cannot be satisfied.

p:- b1,b2,!,a1,a2,a3. **%after reaching !, no backtracking on b1&b2**

p:- r1,...,rn. **%also this rule won't be searched**

p:- morerules. **%this one too!**

- See cut slides for more details.