The current topic: Prolog

✓ Introduction
✓ Object-oriented programming: Python
✓ Functional programming: Scheme
✓ Python GUI programming (Tkinter)
✓ Types and values
• Logic programming: Prolog
  ✓ Introduction
  ✓ Rules, unification, resolution, backtracking, lists.
  ✓ More lists, math, structures.
  ✓ More structures, trees, cut.
  – Next up: Negation.
• Syntax and semantics
• Exceptions

Announcements

• Term Test 2 has been marked.
  – Handed back at the end of class today.
  – The deadline for submitting a re-mark request is the end of class, Friday November 28th. Make sure you understand the posted solutions before submitting a re-mark request.
  – Average: 68.4%

Using not instead of cut to avoid wrong answers

• Prolog has a not operator, but its behaviour is more subtle than in other languages.

• Example: Replacing a cut with not:
  – With cut:
    A :- B, !, C.
    A :- D.
  
  – With not:
    A :- B, C.
    A :- not(B), D.

• Observe that not can be more cumbersome than cut.
  – repetitive if-then-else

Using not for inequality

• Example:
crispy(snap).
crispy(crackle).
crispy(pop).
breakfast(A,B,C) :- crispy(A), crispy(B), crispy(C).

?- breakfast(A,B,C).
A = snap
B = snap
C = snap ;

A = snap
B = snap
C = crackle ;
...

• But we really want A, B, and C to be different from each other.
Using not for inequality

```prolog
crispy(snap).
crispy(crackle).
crispy(pop).
breakfast(A,B,C) :- crispy(A), crispy(B), crispy(C), not(A=B),
                 not(A=C), not(B=C).
```

?- breakfast(A,B,C).
A = snap
B = crackle
C = pop ;
A = snap
B = pop
C = crackle ;
...

Negation in Prolog

- `not(B)` can also be written as `\+ B`.
- And `not(X=Y)` can also be written as `X \= Y`.
- The goal `\+ X` succeeds iff `X` fails.
- Examples:
  ```prolog
  ?- \+ member(b, [a,b,c]).
  No
  ?- \+ member(x, [a,b,c]).
  Yes
  ?- \+ member(X, [a,b,c]).
  No
  ```

Negation as failure

- Prolog assumes that if it can’t prove an assertion, then the assertion is false.
  - And Prolog assumes that if it can prove an assertion, then the assertion is true.
- This is the "closed world assumption": in the universe of facts Prolog knows about, failure to prove is proof of failure.
  - But if we know something Prolog doesn’t, this can lead to surprises: things that Prolog thinks are false when we know they’re true, and the opposite.
  - Example:
```
university(uoft).
?- university(york).
No
?- \+ university(york).
Yes
```
Be careful with negation

sad(X) :- \+ happy(X).
happy(X) :- beautiful(X), rich(X).
rich(bill).
beautiful(michael).
rich(michael).
beautiful(cinderella).

?- sad(michael).
No
?- sad(jim).
Yes
?- sad(Someone).
No

• Isn’t anyone sad?
• No, that just means that it’s not true we can’t find anyone happy.
  – In other words, there exists someone who is happy.

Set overlap

• Write a predicate overlap(S1, S2) that succeeds if lists S1 and S2 have a common element. Then write a predicate disjoint(S1, S2) that succeeds if S1 and S2 have no common element.

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

?- overlap([a,b,c], [c,d,e]).
Yes
?- disjoint([a,b,c], [c,d,e]).
No
?- overlap([a,b,c], [d,e,f]).
No
?- disjoint([a,b,c], [d,e,f]).
Yes
?- disjoint([a,b,d], S).
No

What does that mean?

?- disjoint([a,b,d], S).
No

• The query should mean "can you find a list S that is disjoint from the list [a,b,d] (and if so what is it)?".
• Obviously there are many such sets, so why "No"?
• Answer: because Prolog succeeded in finding a set that did overlap with S, so it announced failure of the original query.

?- overlap([a,b,d], S).
S = [a|_G226] ;
S = [_G225, a|_G229] ;
S = [_G225, _G228, a|_G232] ;
...
**Safe use of negation**

- The goal \( \text{not}(G) \) is **safe** if either:
  - \( G \) is fully instantiated when \( \text{not}(G) \) is processed, or
  - \( G \) has uninstantiated variables, but they don’t appear anywhere else in the clause.

- **Safe example:**
  ```prolog
  childlessMan(X) :- male(X), \+ parent(X,Y).
  ```
  - \( X \) is instantiated in \( \text{male}(X) \), and \( Y \) isn’t used elsewhere.

- **Unsafe example:**
  ```prolog
  childlessMan(X) :- \+ parent(X,Y), male(X).
  ```
  - \( X \) is not instantiated before the negation, and is used elsewhere.

- If necessary, add a precondition to warn the programmer.
  - recall that \( +\text{Var} \) means that \( \text{Var} \) must be instantiated.

% disjoint(+S1, +S2) succeeds if...
```prolog
  disjoint(S1, S2) :- \+ overlap(S1, S2).
```  - When the precondition is satisfied, this negation is safe.

**Double-negation doesn't "cancel out"**

- In other languages, \( \text{not} (\text{not}(<\text{expression}>)) \) is equivalent to \(<\text{expression}>\).
  - But not in Prolog.

```prolog
?- member(X,[a,b,c]).
X = a ;
X = b ;
X = c ;
No
?- not(not(member(X,[a,b,c]))).
X = _G166 ;
No
```

- Why is \( X \) uninstantiated in this example?
  - Since \( \text{member}(X, [a,b,c]) \) succeeds (by instantiating \( X \) to, say, \( a \)),
    \( \text{not}(\text{member}(X, [a,b,c])) \) fails.
  - When a goal fails, the variables it instantiated get uninstantiated. So \( X \) gets uninstantiated.
  - But since \( \text{not}(\text{member}(X, [a,b,c])) \) fails, \( \text{not}(\text{not}(\text{member}(X, [a,b,c]))) \) succeeds.

**fail**

- The \( \text{fail} \) predicate fails immediately. Example:

```prolog
p(X) :- fail.
?- p(csc326).
No
```

- We can use \( \text{fail} \) to state that something is false.
• Example: We want to represent "Colbert does not like bears (regardless of whatever else he likes)."
  – One solution: Add "\texttt{not(bear(X))}" to every rule describing what Colbert likes. For example:
    
    \begin{verbatim}
likes(colbert, X) :- animal(X), not(bear(X)).
likes(colbert, X) :- toy(X), not(bear(X)).
likes(colbert, X) :- livesInArctic(X), not(bear(X)).
    \end{verbatim}
  ...
  – Let's try to use \texttt{fail} instead.
  – First attempt:
    \begin{verbatim}
bear(yogi).
animal(yogi).
likes(colbert, X) :- bear(X), fail.
likes(colbert, X) :- animal(X).
    \end{verbatim}
    
    ?- likes(colbert, yogi).
    Yes

• We need to add a cut to prevent other rules from being tried after the first rule reaches \texttt{fail}.
  – Second attempt:
    \begin{verbatim}
bear(yogi).
cat(tom).
animal(yogi).
animal(tom).
likes(colbert, X) :- bear(X), !, fail.
likes(colbert, X) :- animal(X).
    \end{verbatim}
    
    ?- likes(colbert, yogi).
    No
    ?- likes(colbert, tom).
    Yes
    ?- likes(colbert, X).
    No
  – Downside: This solution only works when \(X\) is instantiated.

• Another example: Define a predicate \texttt{different(X, Y)} that succeeds if \(X\) and \(Y\) don't unify.
  \begin{verbatim}
different(X, Y) :- X=Y, !, fail.
different(_, _).
    \end{verbatim}
    
    ?- different(a, b).
    Yes
    ?- different(a, a).
    No

• Notice that the above definition is equivalent to:
    \begin{verbatim}
different(X, Y) :- not(X=Y).
    \end{verbatim}

• We can define the \texttt{not} predicate as follows:

    \begin{verbatim}
not(X) :- X, !, fail.
not(_).
    \end{verbatim}

  (To test this out, use a name other than "\texttt{not}", since Prolog won't let you redefine the built-in "\texttt{not}".)
fail

- Recall the original version of `bstmem(Tree, X)`:  
  \[
  \begin{align*}
  &\text{bstmem}(\text{node}(X, _, _), X). \\
  &\text{bstmem}(\text{node}(K, L, _), X) :\text{=} \ X < K, \text{bstmem}(L, X). \\
  &\text{bstmem}(\text{node}(K, _, R), X) :\text{=} \ X > K, \text{bstmem}(R, X).
  \end{align*}
  \]

- Recall that this version was inefficient.

Inefficiency in `bstmem`

[trace]  
\[- \text{bstmem}(\text{node}(5, \text{node}(3,\text{empty},\text{empty}), \text{empty}), 1). \]

Call: (8) \text{bstmem}(\text{node}(5, \text{node}(3,\text{empty},\text{empty}), \text{empty}), 1) ? creep
  \^ Call: (9) \text{bstmem}(\text{node}(3,\text{empty},\text{empty}), 1) ? creep
    \^ Exit: (9) \text{bstmem}(\text{empty}, 1) ? creep
      \^ Fail: (10) \text{bstmem}(\text{empty}, 1) ? creep
        \^ Redo: (9) \text{bstmem}(\text{node}(3,\text{empty},\text{empty}), 1) ? creep
          \^ Call: (10) \text{bstmem}(\text{empty}, 1) ? creep
            \^ Fail: (10) \text{bstmem}(\text{empty}, 1) ? creep
              \^ Redo: (8) \text{bstmem}(\text{node}(5, \text{node}(3,\text{empty},\text{empty}), \text{empty}), 1) ? creep
                \^ Call: (9) \text{bstmem}(\text{node}(3,\text{empty},\text{empty}), 1) ? creep
                  \^ Exit: (9) \text{bstmem}(\text{empty}, 1) ? creep
                    \^ Fail: (10) \text{bstmem}(\text{empty}, 1) ? creep
                      \^ Redo: (8) \text{bstmem}(\text{node}(5, \text{node}(3,\text{empty},\text{empty}), \text{empty}), 1) ?

fail

- We solved the inefficiency illustrated on the previous slide as follows:
  
  \[
  \begin{align*}
  &\text{bstmem}(\text{node}(X, _, _), X). \\
  &\text{bstmem}(\text{node}(K, L, _), X) :\text{=} \ X < K, !, \text{bstmem}(L, X). \\
  &\text{bstmem}(\text{node}(K, _, R), X) :\text{=} \ X > K, \text{bstmem}(R, X).
  \end{align*}
  \]

- What if we try to instead solve this inefficiency by using `fail`:

  \[
  \begin{align*}
  &\text{bstmem}(_, _) :\text{=} !, \text{fail}. \\
  &\text{bstmem}(\text{node}(X, _, _), X). \\
  &\text{bstmem}(\text{node}(K, L, _), X) :\text{=} \ X < K, \text{bstmem}(L, X). \\
  &\text{bstmem}(\text{node}(K, _, R), X) :\text{=} \ X > K, \text{bstmem}(R, X).
  \end{align*}
  \]

Tracing the new `bstmem`

[trace]  
\[- \text{bstmem}(\text{node}(5, \text{node}(3,\text{empty},\text{empty}), \text{empty}), 1). \]

Call: (8) \text{bstmem}(\text{node}(5, \text{node}(3,\text{empty},\text{empty}), \text{empty}), 1) ? creep
  \^ Call: (9) \text{bstmem}(\text{node}(3,\text{empty},\text{empty}), 1) ? creep
    \^ Exit: (9) \text{bstmem}(\text{empty}, 1) ? creep
      \^ Fail: (10) \text{bstmem}(\text{empty}, 1) ? creep
        \^ Redo: (9) \text{bstmem}(\text{node}(3,\text{empty},\text{empty}), 1) ? creep
          \^ Call: (10) \text{bstmem}(\text{empty}, 1) ? creep
            \^ Fail: (10) \text{bstmem}(\text{empty}, 1) ? creep
              \^ Redo: (8) \text{bstmem}(\text{node}(5, \text{node}(3,\text{empty},\text{empty}), \text{empty}), 1) ? creep
                \^ Call: (9) \text{bstmem}(\text{node}(3,\text{empty},\text{empty}), 1) ?

No
fail

• What went wrong?
  – fail only affects the present goal (bstmem(empty, 1) in the example).

  – It does not directly cause the failure of a previous goal (so, in the example, Prolog still looks for other rules for the goal bstmem(node(3, empty, empty), 1)).

Advice on writing Prolog

To minimize bugs, especially with cut and not:

• Use cut and not as necessary to avoid wrong answers.

• Follow the rules for safe use of not.

• Follow the rules for doing arithmetic.

• Always use “;” when testing to check all possible answers.
  – It’s easy to get first answer right and rest wrong if "else" misused.

• Test with variables in every combination of positions.

• Use preconditions to state where variables are disallowed.

• Use cut to avoid duplicate answers.

• Use cut where possible for efficiency.

• Use _ where possible for efficiency.

Summary: logic programming and Prolog

• Logic programming:
  – Unification, resolution, backtracking.
  – Specify kind of result wanted (what you want), not how to get it.

• Prolog:
  – The major logic programming language.
  – Efficiency can be a worry:
    • cut
    • ordering the predicates

Bubble sort

• Write a predicate bsort(+Before, ?After) that succeeds if After is a sorted version of Before. bsort should use bubble sort to sort the list.

bsort(Before, After) :- bsortaux(Before, [], After).

• Helper predicate bsortaux(+Prelower, +Preupper, ?Sorted) succeeds if Sorted is a list that consists of a sorted version of Prelower followed by (an unchanged) Preupper.

bsortaux([], Preupper, Preupper) :- !.
bsortaux(Prelower, Preupper, Sorted) :-
  bubble(Prelower, Preupper, Postlower, Postupper),
  bsortaux(Postlower, Postupper, Sorted).
• Helper predicate `bubble(+Prelower, +Preupper, ?Postlower, ?Postupper)` succeeds if performing one round of bubble sort on unsorted portion `Prelower` and sorted portion `Preupper` results in unsorted portion `Postlower` and sorted portion `Postupper`.

```prolog
bubble([X, Y | Rest], Preupper, [X | Bubbled], Postupper) :-
  X =< Y, % No swap needed.
  !,
  bubble([Y | Rest], Preupper, Bubbled, Postupper).

bubble([X, Y | Rest], Preupper, [Y | Bubbled], Postupper) :-
  bubble([X | Rest], Preupper, Bubbled, Postupper).

bubble([X], Preupper, [], [X|Preupper]) :- !.

bubble([], Preupper, [], Preupper). % not needed, we hope
```

```
[trace]  ?- bsort([2,1], S).
Call: (7) bsort([2, 1], _G290) ? creep
Call: (8) bsormaux([2, 1], [], _G290) ? creep
Call: (9) bubble([2, 1], [], _L206, _L207) ? creep
  ^ Call: (10) 2=<1 ? creep
  ^ Fail: (10) 2=<1 ? creep
  Redo: (9) bubble([2, 1], [], _L206, _L207) ? creep
  Call: (10) bubble([2, 1], [], _G346, _L207) ? creep
  Exit: (10) bubble([2], [], [], [2]) ? creep
  Exit: (9) bsormaux([1, 2], _G290) ? creep
  Call: (10) bubble([1, 2], [], _L245, _L246) ? creep
  Exit: (10) bubble([1], [], [], [1, 2]) ? creep
  Exit: (9) bsormaux([1], [], _G290) ? creep
  Exit: (8) bsormaux([2, 1], [], [1, 2]) ? creep
  Exit: (7) bsort([2, 1], [1, 2]) ? creep
Exit: (S = [1, 2]) ? creep
```

```
[trace]  ?- bsort([3,2,1], S).
Call: (7) bsort([3, 2, 1], _G293) ? creep
Call: (8) bsormaux([3, 2, 1], [], _G293) ? creep
Call: (9) bubble([3, 2, 1], [], _L206, _L207) ? creep
  ^ Call: (10) 3=<2 ? creep
  ^ Fail: (10) 3=<2 ? creep
  Redo: (9) bubble([3, 2, 1], [], _L206, _L207) ? creep
  Call: (10) bubble([3, 1], [], _G352, _L207) ? creep
  ^ Call: (11) 3=<1 ? creep
  ^ Fail: (11) 3=<1 ? creep
  Redo: (10) bubble([3, 1], [], _G352, _L207) ? creep
  Call: (11) bubble([3], [], _G358, _L207) ? creep
  Exit: (11) bubble([3], [], [], [3]) ? creep
  Exit: (10) bubble([3, 1], [], [], [3]) ? creep
  Exit: (9) bubble([3, 2, 1], [], [2, 1], [3]) ? creep
Exit: (S = [1, 2, 3]) ? creep
```
Exercises

• Fix the sibling predicate (that we previously defined) so that it doesn’t consider a person to be their own sibling. Then make sure that this fix has eliminated any unusual behaviour in the aunt, uncle, nephew, and niece predicates that you defined in a previous set of exercises.

• Trace bsort on more interesting (and larger) examples. For example, trace the call:
  
  bsort([1,5,2,6,3,4], S).

• Challenge: Recall that the efficiency of bubble sort can be improved by halting after the first iteration during which no swaps are performed (we can halt at that point since if no swaps are performed, the list must be already sorted). Modify bsort by adding this improvement.