The current topic: Prolog

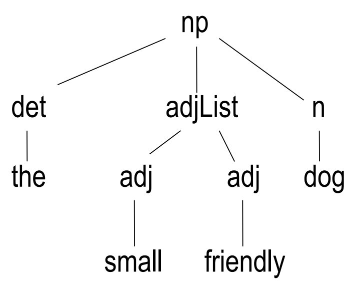
✓ Introduction
✓ Object-oriented programming: Python
✓ Functional programming: Scheme
✓ Python GUI programming (Tkinter)
✓ Types and values
- Logic programming: Prolog
  ✓ Introduction
  ✓ Rules, unification, resolution, backtracking, lists.
  ✓ More lists, math, structures.
  – More structures, trees,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

Announcements

- The project was due **today** at 10:30 am.
- Reminder: Lab 3 has been posted.

Parsing to a Prolog structure

- A parse tree from natural-language computing:



- As a Prolog structure:

```
np(det(the), adjList(adj(small), adj(friendly)), n(dog)).
```

Trees

- Let's represent a binary tree like this:
  – `node(K,L,R)` is a tree with key `K` (an integer) at the root, left subtree `L`, and right subtree `R`.
  – the atom "`empty`" (which is **not** a special part of Prolog) represents an empty binary tree.
- You could also use Prolog lists to represent trees in the same way we did in Scheme, using [ ] to represent an empty tree.

Tree predicates

- We'll write the following predicates:

`treemem(Tree, X)`
  - Succeeds if `X` is contained in `Tree`.

`bstmem(Tree, X) /* for a binary search tree */`
  - Succeeds if `X` is contained in `Tree`, which we assume is a BST.

`bstinsert(Tree, X, Newtree)`
  - Succeeds if `Newtree` is the same as `Tree`, but with `X` added if it is not already present.

treemem(Tree, X)

```
treemem(node(X, _, _), X).
treemem(node(_, L, _), X) :- treemem(L, X).
treemem(node(_, _, R), X) :- treemem(R, X).
```

- Examples:

```
?- treemem(empty, 3).
No

?- treemem(node(5,node(8,empty,empty),empty),8).
Yes

?- treemem(node(4,node(6,empty,empty),node(3,empty,empty)), 3).
Yes

?- treemem(node(4,node(6,empty,empty),node(3,empty,empty)), 0).
No
```

Tracing treemem

```
[trace]  ?-
   treemem(node(4,node(6,empty,empty),node(3,empty,empty)), 0).
   Call: (8) treemem(node(4, node(6, empty, empty), node(3,
   empty, empty)), 0) ? creep
   Call: (9) treemem(node(6, empty, empty), 0) ? creep
   Call: (10) treemem(empty, 0) ? creep
   Fail: (10) treemem(empty, 0) ? creep
   Redo: (9) treemem(node(6, empty, empty), 0) ? creep
   Call: (10) treemem(empty, 0) ? creep
   Fail: (10) treemem(empty, 0) ? creep
   Redo: (8) treemem(node(4, node(6, empty, empty), node(3,
   empty, empty)), 0) ? creep
   Call: (9) treemem(node(3, empty, empty), 0) ? creep
   Call: (10) treemem(empty, 0) ? creep
   Fail: (10) treemem(empty, 0) ? creep
   Redo: (9) treemem(node(3, empty, empty), 0) ? creep
   Call: (10) treemem(empty, 0) ? creep
   Fail: (10) treemem(empty, 0) ? creep
No
```

bstmem(Tree, X)

```
bstmem(node(X, _, _), X).
bstmem(node(K, L, _), X) :- X < K, bstmem(L, X).
bstmem(node(K, _, R), X) :- X > K, bstmem(R, X).
```

- Examples:

```
?- bstmem(empty, 27).
No
?- bstmem(node(5, node(2,empty,empty), node(7,empty,empty)), 5).
Yes
?- bstmem(node(5, node(3,node(1, empty,empty),empty), empty),3).
Yes
?- bstmem(node(5, node(3,node(1, empty,empty),empty), empty),8).
No
```

bstinsert(Tree, X, Newtree)

```
bstinsert(empty, X, node(X, empty, empty)).
bstinsert(Tree, X, Tree) :- Tree = node(X, _, _).
bstinsert(node(K, L, R), X, node(K, Lnew, R)) :-
  X < K, bstinsert(L, X, Lnew).
bstinsert(node(K, L, R), X, node(K, L, Rnew)) :-
  X > K, bstinsert(R, X, Rnew).
```

- Examples:

```
?- bstinsert(empty, 3, Newtree).
Newtree = node(3, empty, empty) ;
No

?- bstinsert(node(3, empty, empty), 2, Newtree).
Newtree = node(3, node(2, empty, empty), empty) ;
No
```

bstinsert(Tree, X, Newtree)

- More examples:

```
?- bstinsert(node(3, node(2, empty, empty), empty), 2, Newtree).
Newtree = node(3, node(2, empty, empty), empty) ;
No

?- bstinsert(node(3, node(2, empty, empty), empty), 4, Newtree).
Newtree = node(3, node(2, empty, empty), node(4, empty, empty));
No
```

Max again

```
max(X, X, X).
max(X, Y, X) :- X > Y.
max(X, Y, Y) :- Y > X.
?- max(2, N, 2).
N = 2 ;
ERROR: Arguments are not sufficiently instantiated
```

- We'd like max to stop after it has succeeded.

Max again, with cut

```
max(X, X, X) :- !.
max(X, Y, X) :- X > Y.
max(X, Y, Y) :- Y > X.

?- max(2, N, 2).
N = 2 ;
No
```

- "!" is the *cut* predicate.

Controlling reasoning with cut

- ! always succeeds immediately.
- Side effects: once satisfied, cut disallows both:
  - Backtracking back over the cut during the application of the **current** rule.
  - Backtracking and applying a different clause of the same predicate to satisfy the **present goal**.
- Cut means "commit".
- ! makes two commitments:
  - To the variable bindings made so far during this application of the **current** rule.
  - To the current rule itself.
  - For example, in the rule
    `p(...) :- q1(A), q2(B), !, q3(C), q4(D).`
    If we reach the cut: we commit to the this rule for `p` (we won't try other rules if this one ends up failing), and we commit to the values of `A` and `B` (we won't try other values as backtracking normally would); if this rule fails, backtracking will just directly return to whatever rule "called" `p` and continue from there.

Example

```
banana(X) :- yellow(X), !, tropical(X).
banana(X) :- red(X), painted(X).
yellow(one).
red(one).
painted(one).
red(two).
painted(two).
tropical(two).

?- banana(one).
No

?- banana(two).
Yes

?- banana(X).
No
```

Example

```
banana(X) :- yellow(X), !, tropical(X).
banana(X) :- red(X), painted(X).
yellow(one).
red(one).
painted(one).
red(two).
painted(two).
tropical(two).

[trace]  ?- banana(one).
   Call: (7) banana(one) ? creep
   Call: (8) yellow(one) ? creep
   Exit: (8) yellow(one) ? creep
   Call: (8) tropical(one) ? creep
   Fail: (8) tropical(one) ? creep
   Fail: (7) banana(one) ? creep
No
```

Example

```
banana(X) :- yellow(X), !, tropical(X).
banana(X) :- red(X), painted(X).
yellow(one).
red(one).
painted(one).
red(two).
painted(two).
tropical(two).
[trace]  ?- banana(two).
   Call: (7) banana(two) ? creep
   Call: (8) yellow(two) ? creep
   Fail: (8) yellow(two) ? creep
   Redo: (7) banana(two) ? creep
   Call: (8) red(two) ? creep
   Exit: (8) red(two) ? creep
   Call: (8) painted(two) ? creep
   Exit: (8) painted(two) ? creep
   Exit: (7) banana(two) ? creep
Yes
```

Example

```
banana(X) :- yellow(X), !, tropical(X).
banana(X) :- red(X), painted(X).
yellow(one).
red(one).
painted(one).
red(two).
painted(two).
tropical(two).

[trace]  ?- banana(X).
   Call: (7) banana(_G283) ? creep
   Call: (8) yellow(_G283) ? creep
   Exit: (8) yellow(one) ? creep
   Call: (8) tropical(one) ? creep
   Fail: (8) tropical(one) ? creep
   Fail: (7) banana(one) ? creep
No
```

Another example

```
max(X, X, X) :- !.
max(X, Y, X) :- X > Y.
max(X, Y, Y) :- Y > X.

?- member(X, [1,2,3]), max(X,X,X).
X = 1 ;
X = 2 ;
X = 3 ;
No
```

- Observe that in this example, the cut in the first rule of `max` does **not** prevent backtracking from finding different instantiations of `X` that satisfy our query.
  - Remember that cut commits to the current rule for satisfying the current goal, and commits to bindings made so far by the current rule.

allHave(List, Key)

- Write a predicate `allHave(List, Key)` that succeeds if `List` is a list and every element of `List` is itself a list that contains `Key`. For example:

```
?- allHave([[a,b], [2, a, [d]], [a]], a).
Yes
?- allHave([a], a).
No
```

- Defining `allHave(List, Key)`:

```
allHave([],_).
allHave([H|T], Key) :- member(Key, H), allHave(T, Key).
```

allListsHave(List, Key)

- Now write a predicate `allListsHave(List, Key)` that succeeds if `List` is a list and every element of `List` **that is itself a list** contains `Key`. (Notice how this is different from `allHave`.)
- We will use the built-in `is_list/1`, that does exactly what its name suggests.

- First ("bad") attempt:

```
badALH([Head | Tail], Key) :- is_list(Head), member(Key, Head),
   badALH(Tail, Key).
badALH([_ | Tail], Key) :- badALH(Tail, Key).
badALH([ ], _).
```

Why did we call it "bad"?

```
?- allListsHave([[a,b], marie, [a]], a).
Yes

?- badALH([[a,b], marie, [a]], a).
Yes

?- allListsHave([[b], marie, [a]], a).
No

?- badALH([[b], marie, [a]], a).
Yes
```

And why *is* it bad?

```
badALH([Head | Tail], Key) :- is_list(Head), member(Key, Head),
   badALH(Tail, Key).
badALH([_ | Tail], Key) :- badALH(Tail, Key).
badALH([ ], _).
```

- In "`badALH([[b], marie, [a]], a)`", when `Head` is `[b]`, the first rule fails.
- Then Prolog tries the second rule, and it succeeds.
- We need to commit to using the first rule as soon as we know that `Head` is a list.

Good allListsHave, with cut

```
allListsHave([Head | Tail], Key) :- is_list(Head), !,
   member(Key, Head), allListsHave(Tail, Key).
allListsHave([_ | Tail], Key) :- allListsHave(Tail, Key).
allListsHave([], _).
```

- Effectively, cut behaves here like an "if/else".
  - That is, if "`is_list(Head)`" is true, the cut commits to the first rule (and hence prevents the second rule from being used).
  - On the other hand, if "`is_list(Head)`" is false, the cut is never reached, so then the second rule is used.

Uses for cut

- Avoid wrong answers.
  - example: `allListsHave`.
- Avoid duplicate answers.
  - example: next slide
- Avoid unnecessary work.
  - Prevent Prolog from using rules that can't possibly lead to success.

Duplicate answers

```
female(ann).
male(ben).
male(bob).
parent(ann, ben).
parent(ann, bob).
isMother(X) :- female(X), parent(X, _).

?- isMother(X).
X = ann ;
X = ann ;
No
```

Avoiding duplicate answers with cut

```
isMotherWithCut(X) :- female(X), parent(X, _), !.

?- isMotherWithCut(X).
X = ann ;
No
```

- The cut commits us to the instantiations used in the rule.
- But that means it commits us both for `X` and for `X`'s child. What if there's another parent?

Missing answers with misplaced cut

```
female(ann).
female(aya).
male(ben).
male(bob).
parent(ann, ben).
parent(ann, bob).
parent(aya, jerry).
parent(aya, sue).

?- isMotherWithCut(X).
X = ann ;
No
```

- Where's `aya`?
  - `X` is instantiated within the rule for `isMotherWithCut`, and this happens before the cut. So after `X` is instantiated to `ann`, the cut commits `X` to this value.

Moving the cut

```
isMotherWithCut2(X) :- female(X), p2(X).
p2(X) :- parent(X, _), !.

?- isMotherWithCut2(X).
X = ann ;
X = aya ;
No
```

- Why does this work?
  - The cut prevents Prolog from looking for other ways to satisfy `p2(X)`, but it does not prevent Prolog from looking for other ways to satisfy `isMotherWithCut2(X)`.

What if we move the cut again?

```
isMotherWithCut3(X) :- female(X), p3(X).
p3(X) :- !, parent(X, _).

?- isMotherWithCut3(X).
X = ann ;
X = ann ;
X = aya ;
X = aya ;
No
```

- Since the cut appears before `parent(X,_)`, it does not prevent Prolog from looking for different ways to satisfy `parent(X,_)`.
  - The cut only prevents Prolog from using any other rules for `p3` (and since there are no such rules, the cut really has no effect in this example).

Unnecessary work

- Recall the BST search predicate we defined earlier:

```
bstmem(node(X, _, _), X).
bstmem(node(K, L, _), X) :- X < K, bstmem(L, X).
bstmem(node(K, _, R), X) :- X > K, bstmem(R, X).
```

- Let's trace a call to see the unnecessary work that occurs:

Unnecessary work

```
[trace]  ?- bstmem(node(5, node(3,empty,empty), emtpy), 1).
   Call: (8) bstmem(node(5, node(3, empty, empty), emtpy), 1) ?
  creep
^  Call: (9) 1<5 ? creep
^  Exit: (9) 1<5 ? creep
   Call: (9) bstmem(node(3, empty, empty), 1) ? creep
^  Call: (10) 1<3 ? creep
^  Exit: (10) 1<3 ? creep
   Call: (10) bstmem(empty, 1) ? creep
   Fail: (10) bstmem(empty, 1) ? creep
   Redo: (9) bstmem(node(3, empty, empty), 1) ? creep
^  Call: (10) 1>3 ? creep
^  Fail: (10) 1>3 ? creep
   Redo: (8) bstmem(node(5, node(3, empty, empty), emtpy), 1) ?
  creep
^  Call: (9) 1>5 ? creep
^  Fail: (9) 1>5 ? creep
No
```

Avoiding unnecessary work with cut

- When searching a BST for a value X, we know that if X is less than the value of the current node then we only need to search in the node's left subtree.
  - However, when this search of the left subtree fails, Prolog still attempts to use the other rule (the one for searching the right subtree), but this (always) ends up failing. To prevent this unnecessary work, we can add a cut.

```
bstmem(node(X, _, _), X).
bstmem(node(K, L, _), X) :- X < K, !, bstmem(L, X).
bstmem(node(K, _, R), X) :- X > K, bstmem(R, X).
```

- Observe that once we know `X < K`, we can commit to the current rule (that is, there's no point trying the rule for `X > K`).

- Let's trace the new version.

Avoiding unnecessary work with cut

```
[trace]  ?- bstmem(node(5, node(3,empty,empty), emtpy),1).
   Call: (8) bstmem(node(5, node(3, empty, empty), emtpy), 1) ?
   creep
^  Call: (9) 1<5 ? creep
^  Exit: (9) 1<5 ? creep
   Call: (9) bstmem(node(3, empty, empty), 1) ? creep
^  Call: (10) 1<3 ? creep
^  Exit: (10) 1<3 ? creep
   Call: (10) bstmem(empty, 1) ? creep
   Fail: (10) bstmem(empty, 1) ? creep
   Fail: (9) bstmem(node(3, empty, empty), 1) ? creep
   Fail: (8) bstmem(node(5, node(3, empty, empty), emtpy), 1) ?
   creep

No
```

Avoiding unnecessary work with cut

- We can do even better. Consider the following trace of the new version:

```
[trace]  ?- bstmem(node(5, node(2,empty,empty),
   node(7,empty,empty)), 7).
   Call: (8) bstmem(node(5, node(2, empty, empty), node(7,
   empty, empty)), 7) ? creep
^  Call: (9) 7<5 ? creep
^  Fail: (9) 7<5 ? creep
   Redo: (8) bstmem(node(5, node(2, empty, empty), node(7,
   empty, empty)), 7) ? creep
^  Call: (9) 7>5 ? creep
^  Exit: (9) 7>5 ? creep
   Call: (9) bstmem(node(7, empty, empty), 7) ? creep
   Exit: (9) bstmem(node(7, empty, empty), 7) ? creep
   Exit: (8) bstmem(node(5, node(2, empty, empty), node(7,
   empty, empty)), 7) ? creep
Yes
```

- Observe that the check 7 > 5 is unnecessary (it has to be true).

Avoiding unnecessary work with cut

- Let's remove this inefficiency:

```
bstmem(node(X, _, _), X) :- !.
bstmem(node(K, L, _), X) :- X < K, !, bstmem(L, X).
bstmem(node(K, _, R), X) :- bstmem(R, X).
```

- The cuts are effectively creating "if/elseif/else" behaviour in this example. Tracing the new version:

```
[trace]  ?- bstmem(node(5, node(2,empty,empty), node(7,empty,empty)),
   7).
   Call: (8) bstmem(node(5, node(2, empty, empty), node(7, empty,
   empty)), 7) ? creep
^  Call: (9) 7<5 ? creep
^  Fail: (9) 7<5 ? creep
   Redo: (8) bstmem(node(5, node(2, empty, empty), node(7, empty,
   empty)), 7) ? creep
   Call: (9) bstmem(node(7, empty, empty), 7) ? creep
   Exit: (9) bstmem(node(7, empty, empty), 7) ? creep
   Exit: (8) bstmem(node(5, node(2, empty, empty), node(7, empty,
   empty)), 7) ? creep
Yes
```

Green cuts vs. red cuts

- In some of the examples we've seen, cuts only prevent duplicate answers or inefficiency.
  - That is, if these cuts are removed, we do **not** get any incorrect answers.
- In other examples, cuts prevent us from getting to incorrect answers (e.g. by behaving like "if/else").
- Cuts that don't change answers (that is, cuts that only prevent inefficiency or duplication) are known as ***green*** cuts.
- Cuts that do change answers (e.g. cuts that behave like "if/else") are known as ***red*** cuts.
  - Such cuts violate the main idea of logic programming, since they specify answers based on *how* these answers are found.
  - Use such cuts carefully. Make sure your code is still readable.

Exercises

- Recall that the `sibling` predicate defined in the Nov. 7th lecture produces duplicate answers on a query like "`sibling(edward, Sib).`". Using cut, fix this predicate to prevent such duplicates. Make sure that it still produces every correct answer (rather than just a single correct answer). Hint: It might be helpful to define a predicate `person(X)` that succeeds when `X` is a person.

- Make `bstinsert` more efficient using cuts (just as we made `bstmem` more efficient). Trace calls to `bstinsert` to identify inefficiency, and trace again after you make your changes to make sure you've removed the inefficiency.