The current topic: Prolog

✓ Introduction
✓ Object-oriented programming: Python
✓ Functional programming: Scheme
✓ Python GUI programming (Tkinter)
✓ Types and values

- Logic programming: Prolog
  ✓ Introduction
  - Next up: Rules, unification, resolution, backtracking, lists.
- Syntax and semantics
- Exceptions

Announcements

- Reminder: The deadline for Lab 2 re-mark requests is **Friday.**
- Reminder: The project is due on **November 17th** at 10:30 am.
  - Make sure you carefully follow the submission instructions.

Prolog syntax

- Variables are capitalized.
- Constants and predicate names begin with a lower-case letter.
- A predicate is identified by its name and the number of parameters it takes.
  - `sibling(A,B)` is called "sibling/2", because it has two parameters.
  - You can define different predicates that have the same name but take a different number of parameters. For example, you can define both `sibling(A,B)` and `sibling(A,B,P)`; the first one is "sibling/2" and the second one is "sibling/3" – these are two distinct predicates that happen to share a name.

Meaning of a Prolog rule

- A rule:

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

- Meaning:

$$\text{parent}(x, y) \wedge \text{female}(x) \supset \text{isMother}(x)$$

- But you really need quantifiers:

$$\forall x\ [\ \ [\exists y\ [\text{parent}(x, y) \wedge \text{female}(x)]] \supset \text{isMother}(x)]$$

How to think in Prolog

- In C, Java, Python, …, you program with instructions.
  - What should we do next, and where should we store the value calculated?
- In Scheme, you program with functions.
  - Given arguments, what's the function value?
  - A function takes you from arguments to result.
- In Prolog, you program with relations.
  - (A function is a mapping from a domain to a range, and each value in the domain is associated with just one value in the range. In a relation, there may be multiple range values for each domain value.)
  - All arguments are at the same level: There is no distinction between "in" and "out" values.

Thinking and style in Prolog

- Although in a Prolog relation, there may be "no distinction between in and out values", some predicates (rules) may have parameters that must have values, or that are always given values. Other parameters may sometimes be given values, and at other times receive them.
- The *documentation* preceding a predicate should specify what's in and what's out:

```
% myRule(+Given, -Deduced, ?Other) does something ...
```

  - This comment implies that `Given` must be set before `myRule` is called, and that `Deduced` receives a value as the result of the call. `Other` may either receive a value or start with an existing value.
  - '+' requires the variable to be instantiated by the caller.
  - '-' requires the variable to be un-instantiated.
  - '?' says that either is acceptable.

How Prolog answers a query

- Unification.
- Resolution.
- Backtracking.

Unification

- Unification succeeds if two expressions can be made to have "the same structure" through variable instantiation (giving a variable a value).
  - An instantiated variable will not change its value. However, it can become un-instantiated when backtracking occurs.
- Unification examples:
  - parent(X,Y) and parent(albert, edward)
These unify: X=albert, Y=edward
  - parent(X,edward) and parent(albert,Y)
These unify: X=albert, Y=edward
  - parent(albert,edward) and parent(victoria,Y)
These do **not** unify, since albert and victoria don't unify.
  - parent(X, edward) and parent(Y, edward)
These unify: X=Y (or Y=X, or creating a variable Z and letting X=Z and Y=Z).

Unification

- More unification examples:
  - parent(X,Y) and female(X)
    These do **not** unify, since parent and female don't unify.
  - parent(albert,edward) and parent(X,Y,Z)
    These do **not** unify, since the first expression has 2 arguments and the second expression has 3 arguments.
- Observe that in order for two expressions to unify, they must have the same functor and the same number of arguments (or else there's no way for them to have the same structure).

What = does

- In Prolog, = calls for unification.
  - **Not** assignment.
  - **Not** equality testing.

```
?- parent(X,Y) = parent(albert,Z).
X = albert
Y = _G158
Z = _G158

?- parent(X,Y) = parent(A,B).
X = _G157
Y = _G158
A = _G157
B = _G158
```

What = does

```
?- parent(X,Y) = course(A,B).
No

?- parent(X,X) = parent(albert,Y).
X = albert
Y = albert

?- parent(X,Y,Z) = parent(A,B).
No
```

Resolution

- Resolution involves combining information from separate rules.
  - This is the way Prolog tries to prove that a query succeeds.
- The general idea:
  - If we have rules of the form
    `A :- B.`
    `C :- D.`
    where `A`, `B`, `C`, `D` are expressions, **and** if `B` and `C` unify, say to `U`, then we have
    `A :- U.`
    `U :- D.`
    and hence we have
    `A :- D.`
    That is, to prove `A`, it suffices to prove `D`.
  - So given a query `Q`, Prolog iterates through its rules (and facts) until it finds one whose left side unifies with `Q`, and then it tries to prove (in order) **each** *sub-query* given by the right side of the rule. For example, given the two rules above, if query `Q` unifies with `A`, then Prolog will try to prove `B` (by, again, iterating through its rules and facts, looking for one whose left side unifies with `B`).

Backtracking

- Backtracking involves trying another set of possible variable instantiations, when the previous set fails or is rejected.
  - This is possibly the hardest part of Prolog for imperative programmers to understand.
- Why does Prolog need to backtrack?
  - Recall (from the previous slide) that when trying to prove a query `Q`, Prolog selects the first rule R whose left side unifies with `Q`.
  - Even if this first rule ultimately leads to failure, there may be other rules that lead to success.
  - So Prolog needs to "undo" the unification of `Q` with the left side of R, and then look for the next rule whose left side unifies with `Q`.
- How much does backtracing "undo"?
  - As little as possible.
  - Just like depth-first search.
- Interactively, typing **;** calls for backtracking.

Tracing

- Use the "`trace.`" command to enter tracing mode.

```
?- trace.
Yes
[trace]  ?- parent(Person, edward).
   Call: (7) parent(_G283, edward) ? creep
   Exit: (7) parent(albert, edward) ? creep
Person = albert ;
   Redo: (7) parent(_G283, edward) ? creep
   Exit: (7) parent(victoria, edward) ? creep
Person = victoria ;
   Redo: (7) parent(_G283, edward) ? creep
   Fail: (7) parent(_G283, edward) ? creep
No
[debug]  ?- nodebug.
Yes
?-
```

A query to trace

```
male(tom).
male(peter).
male(doug).
female(susan).
male(david).
parent(doug, susan).
parent(tom, william).
parent(doug, david).
parent(doug, tom).
grandfather(GP, GC) :- male(GP), parent(GP, A), parent(A, GC).

?- grandfather(X,Y).
```

Let Prolog trace it

```
[trace]  ?- grandfather(X,Y).
  Call: (7) grandfather(_G283, _G284) ? creep
  Call: (8) male(_G283) ? creep
  Exit: (8) male(tom) ? creep
  Call: (8) parent(tom, _G353) ? creep
  Exit: (8) parent(tom, william) ? creep
  Call: (8) parent(william, _G284) ? creep
  Fail: (8) parent(william, _G284) ? creep
  Fail: (8) parent(tom, _G353) ? creep
  Redo: (8) male(_G283) ? creep
  Exit: (8) male(peter) ? creep
  Call: (8) parent(peter, _G353) ? creep
  Fail: (8) parent(peter, _G353) ? creep
  Redo: (8) male(_G283) ? creep
  Exit: (8) male(doug) ? creep
  Call: (8) parent(doug, _G353) ? creep
  Exit: (8) parent(doug, susan) ? creep
  Call: (8) parent(susan, _G284) ? creep
  Fail: (8) parent(susan, _G284) ? creep
  Redo: (8) parent(doug, _G353) ? creep
  Exit: (8) parent(doug, david) ? creep
  Call: (8) parent(david, _G284) ? creep
  Fail: (8) parent(david, _G284) ? creep
  Redo: (8) parent(doug, _G353) ? creep
  Exit: (8) parent(doug, tom) ? creep
  Call: (8) parent(tom, _G284) ? creep
  Exit: (8) parent(tom, william) ? creep
  Exit: (7) grandfather(doug, william) ? creep

X = doug
Y = william
```

The anonymous variable

- A variable that appears only once is called a *singleton variable*.

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

  - Y's value doesn't matter; it doesn't have to match anything else in the rule.

- Singletons consume run-time resources.

- Use "_" instead -- the *anonymous* variable.

```
isaMother(X) :- female(X), parent(X, _).
```

  - _ matches anything.
  - But don't use two _'s hoping they'll match each other.

Lists in Prolog

- Two ways to describe a list.

- Element by element, with commas:

```
[a, b, c]
[ ]
[a, [b, c], d, [ ], e]
[a, X, c, d]
```

- [first | rest] where "rest" must itself be a list:

```
[a | [b, c] ]      (same as [a, b, c])
[a | Rest ]
```

Why two ways?

```
?- [a, b, c, d] = [ H | T ].

H = a
T = [b, c, d]
```

- In Prolog, you don't need car and cdr functions to break up a list!

Unifying lists

```
?- [X, Y, Z] = [bob, likes, bananas].
X = bob
Y = likes
Z = bananas

?- [1, 2] = [X | Y].
X = 1
Y = [2]

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

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

Unifying lists

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

?- [First | [a, b]] = [z, a, b].
First = z

?- [[the | Y] | Z] = [[X, hare] | [is, here]].
Y = [hare]
Z = [is, here]
X = the

?- Y = [Y].
Y = [[[[[[[[[[...]]]]]]]]]]
```

Writing list predicates

- You need recursion if the list length isn't fixed. The usual considerations are relevant:

1. Choose suitable names for the predicate and the arguments.
2. Write specifications in the form "predicate succeeds if …" *and think of it that way*.
3. Write the base cases first.
   - Why?

4. There may be several non-trivial cases, each requiring a rule.
   - You may need to carefully order the rules.

Two list predicates

- Consider the following predicate:

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

- Observe that `member(X,L)` means that `X` is an element of `L`. (Prolog also has a built-in version of member/2.)

- Another list predicate:

```
bigger(_, [ ]).
bigger(X, [First | Rest]) :- X > First, bigger(X, Rest).
```

- What does bigger/2 mean?
  - `bigger(X,L)` means that `X` is bigger than every element in `L`.

append: many predicates in one

- Build a list:

```
?- append([a], [b], Y).
Y = [a, b] ;
No
```

- Break a list apart:

```
?- append(X, [b], [a, b]).
X = [a] ;
No

?- append([a], X, [a, b]).
X = [b] ;
No
```

append, with more than one variable

```
?- append(X, Y, [a, b]).
X = [ ]
Y = [a, b] ;

X = [a]
Y = [b] ;

X = [a, b]
Y = [ ] ;
No
```

append, generating a list

```
?- append(X, [a], Y).
X = [ ]
Y = [a] ;

X = [_G230]
Y = [_G230, a] ;

X = [_G230, _G236]
Y = [_G230, _G236, a] ;

X = [_G230, _G236, _G242]
Y = [_G230, _G236, _G242, a] ;
```

… and so on …

Writing myAppend

- There is a built-in `append`. Let's write our own version, called `myAppend`.

- Things that might be useful:

```
myAppend(X, [ ], X).
myAppend([ ], Y, Y).
myAppend([H | R], Y, [H | New]) :- myAppend(R, Y, New).
```

- Is that enough? too much?
  - That is, (1) does it work? (2) could we delete one or two of the rules and still have a functioning `myAppend`?

Writing myAppend

- Testing `myAppend`:

```
?- myAppend([a], [b], Y).
Y = [a, b] ;
No

?- myAppend(X, [b], [a,b]).
X = [a] ;
No

?- myAppend([a], X, [a,b]).
X = [b] ;
No

?- myAppend([a], [], Y).
Y = [a] ;
Y = [a] ;
Y = [a] ;
No.
```

Writing myAppend

- More testing:

```
?- myAppend(X, Y, [a,b]).
X = [a, b]
Y = [] ;

X = []
Y = [a, b] ;

X = [a]
Y = [b] ;

X = [a, b]
Y = [] ;

X = [a, b]
Y = [];
No
```

Writing myAppend

- Our `myAppend` works, but repeats answers. Let's try removing one of the rules. Suppose we re-write `myAppend` as follows:

```
myAppend(X, [ ], X).
myAppend([H | R], Y, [H | New]) :- myAppend(R, Y, New).
```

- Testing the new version:

```
?- myAppend([a], [b], Y).
No
```

  – Why? To prove `myAppend([a], [b], Y)`, Prolog uses the second rule and tries to prove `myAppend([], [b], Y)`. But `myAppend([], [b], Y)` doesn't unify with the left side of either of the two rules.

Writing myAppend

- Let's try again:

```
myAppend([ ], Y, Y).
myAppend([H | R], Y, [H | New]) :- myAppend(R, Y, New).
```

- Testing:

```
?- myAppend([a], [b], Y).
Y = [a, b] ;
No

?- myAppend([a], [], Y).
Y = [a] ;
No

?= myAppend(X, [b], [a, b]).
X = [a];
No
```

- This version works, and doesn't repeat answers.

Exercises

- Using the `parent`, `male`, `female`, and `sibling` predicates as a starting point, write the following predicates. Recall that our `sibling` predicate behaves somewhat strangely (since it considers a person to be their own sibling), and this may cause similar strange behaviour in the predicates you define – don't worry about this for now (until we discuss negation).
  – `uncle`
  – `aunt`
  – `nephew`
  – `niece`
  – `grandparent`
  – `ancestor`