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.
  - Next up: More lists, math, structures.
- Syntax and semantics
- Exceptions

Announcements

- Reminder: The project is due on **Monday** at 10:30 am.
  - Make sure you carefully follow the submission instructions.
- Lab 3 has been posted.

swapFirstTwo

- We want to write a predicate `swapFirstTwo(List1, List2)` that succeeds if `List1` and `List2` are lists of length at least 2 that are the same except the first two elements of `List1` are in reverse order in `List2`. Examples:

```
?- swapFirstTwo([a, b], [b, a]).
Yes
?- swapFirstTwo([a, b], [b, c]).
No
?- swapFirstTwo([a, b, c], [b, a, c]).
Yes
?- swapFirstTwo([a, b, c], [b, a, d]).
No
?- swapFirstTwo([a, b, c], X).
X = [b, a, c] ;
No
```


swapFirstTwo

- More examples:

```
?- swapFirstTwo([a, b | Y], X).
Y = _G161
X = [b, a|_G161] ;
No

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

?- swapFirstTwo(X, Y).
X = [_G225, _G228|_G229]
Y = [_G228, _G225|_G229] ;
No
```

swapFirstTwo

- Defining `swapFirstTwo`:

```
swapfirsttwo([X, Y | R], [Y, X | R]).
```

- Only one rule is needed!

isPrefix

- Write a predicate `isPrefix(Little, Big)` that succeeds if `Big` is a list beginning with all the members of `Little`, in order.

```
isPrefix([],[]).
isPrefix([],[_|_]).
isPrefix([H|T], [H|Rest]) :- isPrefix(T, Rest).
```

- Testing `isPrefix`:

```
?- isPrefix([1,2], [1,2,3,4]).
Yes
?- isPrefix(L, [1,2,3,4]).
L = [] ;
L = [1] ;
L = [1, 2] ;
L = [1, 2, 3] ;
L = [1, 2, 3, 4] ;
No
```


occursIn

- Write a predicate `occursIn(Little, Big)` that succeeds if `Little` is a sublist of `Big` (this means that the elements of `Little` appear together, in order, within `Big`).

```
occursIn(Little, Big) :- isPrefix(Little, Big).
occursIn(Little, [_|T]) :- occursIn(Little, T).
```

- Testing `occursIn`:

```
?- occursIn([1,2], [1,2,3]).
Yes
?- occursIn([2,3], [1,2,3,4]).
Yes
?- occursIn([A], [1,2,3,4]).
A = 1 ;
A = 2 ;
A = 3 ;
A = 4 ;
No
```


length(List, N)

- The built-in predicate `length(List,N)` succeeds if `List` is a list of length `N`.
- Let's try to define our own version, which we'll call `len` instead.
- First attempt:

```
len([ ], 0).
len([_ | Rest], N) :- len(Rest, N - 1).
```

- Testing `len`:

```
?- len([], Val).
Val = 0 ;
No
?- len([a,b,c], Val).
No
?- len([a,b,c], 3).
No
```

- What's going on?

Tracing len

- Let's trace a call to `len`:

```
[trace]  ?- len([a,b,c], 3).
   Call: (7) len([a, b, c], 3) ? creep
   Call: (8) len([b, c], 3-1) ? creep
   Call: (9) len([c], 3-1-1) ? creep
   Call: (10) len([], 3-1-1-1) ? creep
   Fail: (10) len([], 3-1-1-1) ? creep
   Fail: (9) len([c], 3-1-1) ? creep
   Fail: (8) len([b, c], 3-1) ? creep
   Fail: (7) len([a, b, c], 3) ? creep

No
```

- We'll later see how to fix the problem.

Math in Prolog

- Let's try to do some math in Prolog.

```
?- X = 14 - 2, Y = 12 - 0, X = Y.
No

?- X = 14 - 2, Y = 2, Z = 14 - Y, X = Z.
X = 14-2
Y = 2
Z = 14-2 ;
No
```

- Recall that = calls for unification, not assignment.

For math, use 'is', not '='

- `X is expression` causes `expression` to be evaluated and then tries to unify the result with `X`.
- In "`X is expression`", `expression` must be:
  - an arithmetic expression
  - fully instantiated
- Examples:

```
?- X is 10 + 17.
X = 27 ;
No
?- Y is 7, Z is 3 + 4, Y = Z.
Y = 7
Z = 7 ;
No
```


For math, use 'is', not '='

- More examples:

```
?- Y is 7, X is Y+2.
Y = 7
X = 9 ;
No

?- X is Y+2, Y is 7.
ERROR: Arguments are not sufficiently instantiated
```

Fixing len

- We can now try to fix `len` using `is`:

```
len([ ], 0).
len([_ | Rest], N) :- len(Rest, M), M is N-1.
```

- Testing `len`:

```
?- len([a,b,c], Val).
ERROR: Arguments are not sufficiently instantiated

?- len([a,b,c], 3).
ERROR: Arguments are not sufficiently instantiated
```


Tracing len

- Let's figure out what's going wrong:

```
[trace]  ?- len([a,b,c], Val).
   Call: (8) len([a, b, c], _G296) ? creep
   Call: (9) len([b, c], _L191) ? creep
   Call: (10) len([c], _L208) ? creep
   Call: (11) len([], _L225) ? creep
   Exit: (11) len([], 0) ? creep
^  Call: (11) 0 is _G360-1 ? creep
ERROR: Arguments are not sufficiently instantiated
```


Fixing len (again)

- We need to fix the `is` so that the right side is always instantiated:

```
len([ ], 0).
len([_ | Rest], N) :- len(Rest, M), N is M+1.
```

- Testing `len`:

```
?- len([a,b,c], Val).
Val = 3 ;
No

?- len(List, 3).
List = [_G216, _G219, _G222] ;
```

...(Non-terminating computation – do a trace to see why)...

max

- We want to write a predicate `max(X, Y, Z)` that succeeds if `Z` is the maximum of `X` and `Y`.

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

- Testing `max`:

```
?- max(2, 3, N).
N = 3 ;
No
?- max(2, 2, N).
N = 2 ;
No
?- max(3,2,N).
N = 3 ;
No
```

max

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

- Observe that one correct answer is provided before the error. We'll see later how to use `cut` to get Prolog to stop looking for answers after the first one (and hence prevent the error).

factorial(N, Ans)

- Write a predicate `factorial(N, Ans)` that succeeds if `Ans` is `N!`:

```
factorial(0, 1).
factorial(N, Ans) :- M is N - 1, factorial(M, A), Ans is N*A.
```

- Testing `factorial`:

```
?- factorial(0, F).
F = 1 ;
ERROR: Out of local stack

?- factorial(5, F).
F = 120 ;
ERROR: Out of local stack
```

- What causes the error? Consider what happens when the second rule is used to answer `factorial(0,F)`.

factorial(N, Ans)

- More testing:

```
?- factorial(69, F).
F = 1.71122e+98
Yes

?- factorial(70, F).
F = 1.19786e+100
Yes

?- factorial(-1, F).
ERROR: Out of local stack

?- factorial(N, 6).
ERROR: Arguments are not sufficiently instantiated
```


sumlist(List, Total)

- Write a predicate `sumlist(List, Total)` that succeeds if `Total` is the sum of the numbers in `List`.

```
sumlist([ ], 0).
sumlist([H | Rest], Total) :- sumlist(Rest, S), Total is S + H.
```

- Testing `sumlist`:

```
?- sumlist([3, 7], X).
X = 10 ;
No

?- sumlist(X, 3).
ERROR: Arguments are not sufficiently instantiated
```

Arithmetic predicates may not be invertible

- You may not be able to supply a variable for some of the parameters.
  - For example, `f(X, 3).` might be OK, while `f(3, X).` is not.

- Every time you use "`is`", you must be sure the expression to the right will be fully instantiated.
  - If necessary, add a precondition to the predicate so that the user knows what is required, including which of the predicate's variables must be instantiated.

Univ

- `=..` is called "univ". Use it to build queries:

```
check(Val1, Val2, Comp) :- Query =.. [Comp, Val1, Val2], Query.
```

- `Query =.. [Comp, Val1, Val2]` succeeds when `Query` is `Comp(Val1, Val2)`.
  - That is, it unifies `Query` with `Comp(Val1, Val2)`.
  - `Comp` is the *functor*.

- In the above example, the last predicate "executes" `Query`: it looks to see if `Query` succeeds after univ has built it.

- Example:

```
?- check([a,b,c], L, length).
L = 3 ;
No
```


univ

- Examples:

```
?- check(3, 5, <).
Yes

?- check(5, 3, <).
No

?- check([a,b,c], L, length).
L = 3 ;
No
```


Programs vs. data

```
check(Val1, Val2, Comp) :- Query =.. [Comp, Val1, Val2], Query.
```

- We're building a data structure and executing it.
  - This should remind you of `eval` in Scheme:
    `(eval '(Comp Val1 Val1))`

Programs vs. data

- Program (query):
  `parent(X, edward).`
- Data:
  `parent(victoria, edward).`
- There is no structural difference between a query and data.
  - But we can execute a query.
- So we can build up a query, or modify it, and then execute the result.

Structures in Prolog

- An example of a structure:

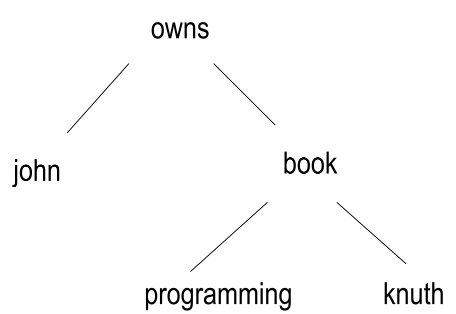
`mother(elizabeth, charles)`

- "`mother`" is the *functor*.
  - "`elizabeth`" and "`charles`" are the *components*.
- A predicate is a structure that you think of as code:
  - `mother(elizabeth, charles).` states a fact that Prolog can reason with, so it's code.
- Structures of the same form can also be used as data structures.
  - Whether a particular structure is a predicate or a data structure depends on context.
  - Structures can be nested.

Structure as data structure

```
owns(john, book(programming, knuth)).
```

- Think of it as a tree:



Unification with structures

```
owns(john, book(programming, knuth)).

?- owns(john, X).
X = book(programming, knuth)

?- owns(john, X), X = book(Y, Z).
X = book(programming, knuth)
Y = programming
Z = knuth

?- owns(john, book(Y,Z)).
Y = programming
Z = knuth

?- owns(X, book(Y,Z)).
X = john
Y = programming
Z = knuth
```

"Prolog doesn't ***think***!"

```
mother(elizabeth, charles).
happy(elizabeth).

?- happy(mother(X, charles)).
No
```

- We don't have a structure that matches the query.
  - That is, we don't have a fact stating that `mother(elizabeth, charles)` is happy.
- But we can ask who is happy and is also the mother of `charles`:

```
?- happy(X), mother(X, charles).
X = elizabeth ;
No
```


Exercises

- Write a predicate `allLists(List)` that succeeds if every element of `List` is itself a list. For example:
```
?- allLists([[a], [b], []]).
Yes
?- allLists([[a], b]).
No
```
- Write a predicate `dotProduct(X,Y,D)` that succeeds if `X` and `Y` are lists of integers, and `D` is the dot product of `X` and `Y` (when these lists are viewed as vectors). Determine appropriate preconditions. Examples:
```
?- dotProduct([1,2,3], [4,5,6], D).
D = 32 ;
No
?- dotProduct([],[], D).
D = 0 ;
No
```