CS540 Introduction to Artificial Intelligence: **Programming in Logic**

Michael Coen

Week of Wednesday, November 11, 2015

(Week #10)

Declarative Programming

- A powerful technique
  - Rely on our programming language to do most of the work

- Do you recall this quote?

  Underlying our approach to this subject is our conviction that "computer science" is not a science and that its significance has little to do with computers. The computer revolution is a revolution in the way we think and in the way we express what we think… Mathematics provides a framework for dealing precisely with notions of "what is." Computation provides a framework for dealing precisely with notions of "how to."

  Abelson and Sussman, *Structure and Interpretation of Computer Programs* (1984).

Declarative Programming

- A powerful technique
  - Rely on our programming language to do most of the work

- Do you recall this quote?

  Underlying our approach to this subject is our conviction that "computer science" is not a science and that its significance has little to do with computers. The computer revolution is a revolution in the way we think and in the way we express what we think… Mathematics provides a framework for dealing precisely with notions of "what is." Computation provides a framework for dealing precisely with notions of "how to."

  Abelson and Sussman, *Structure and Interpretation of Computer Programs* (1984).

Programming in logic blurs this distinction.

Programming Styles

- Imperative:
  - Procedural

    Step -by-step instructions for doing something

    C, C++, Java, Python, Fortran, etc.

  - Functional

    Solve problems by reducing them to simpler problems

    Recursion is a basic motif

    Scheme, Lisp, Algol, Haskel, etc.

- Declarative:

  $\sqrt{x}$ is the y such that $y^2 = x$ and $y \geq 0$

  Um, ok? But what's $\sqrt{7}$?

  Prolog, Aleph, Golem, etc.

Preliminary Prolog

- This will be a gentle introduction to Prolog
  - The ideas are important
  - **You should know this stuff is possible**

- Prolog = "Programming in Logic"
  - Developed by Colmerauer, Roussel, Kowalski, and others around 1972
    "Programmation en Logique"

- Stems from two intellectual threads:
  - Logic as the theoretical foundation for Computer Science
  - Strong focus in AI to use logic to represent and reason about problems.

    Early AI focused almost exclusively on symbolic logic. **Probabilistic logics have made a strong resurgence in recent years, particularly with the popularity of *Bayesian* methods.**

Some Comments about Logic Programming

- Prolog is the most popular logic programming language
  - Many versions of Prolog
  - SWI-Prolog is free, runs on all platforms, and very popular: http://www.swi-prolog.org
  - We'll be using SWI-Prolog for demos and the little bit of Prolog you may be playing with.

- Other popular implementations:
  - BProlog: Provides memoization and all the mathematical functions missing in ISO Prolog
  - Datalog: Russell and Norvig seem to like this. (Should have replaced SQL years ago…)
    - Not Turing-complete! (This is not a "real" programming language)
  - Many, many others… (Yap = "Yet Another Prolog" is popular here)

- Major programming languages have Prolog extensions, libraries…
    - Java, C, C++, Matlab, Scheme, Python, Haskell, SQL, Perl, Fortran, etc…

- Side note:
  - I program a lot in a language called Prism, a probabilistic version of Prolog.
  - Derives the *most likely* inference.
  - Wonderful tool for modeling human reasoning, especially "weird" human reasoning

Knowledge Representation

Towards a definition of "family!"

Motherhood: $\forall m,c\ Mother(c) = m \Leftrightarrow Female(m) \wedge Parent(m,c).$

Husband: $\forall w,h\ Husband(h,w) \Leftrightarrow Male(h) \wedge Spouse(h,w).$

Disjoint: $\forall x\ Male(x) \Leftrightarrow \neg Female(x).$

Inverse: $\forall p,c\ Parent(p,c) \Leftrightarrow Child(c,p).$

Sibling: $\forall x,y\ Sibling(x,y) \Leftrightarrow x \neq y\ \wedge\ \exists p\ Parent(p,x)\ \wedge\ Parent(p,y)$

Transitivity: $\forall g,c\ Grandparent(g,c) \Leftrightarrow \exists p\ Parent(g,p)\ \wedge\ Parent(p,c).$

Recursion: $\forall a,c\ Ancestor(a,c)\ \Leftrightarrow\ \exists p\ Parent(p,c)\ \wedge\ Ancestor(a,p).$

Recursion: $\forall a,c\ Ancestor(a,c)\ \Leftrightarrow\ \exists r\ Ancestor(r,c)\ \wedge\ Parent(a,r).$

Can we prove things in this system?

$\therefore\ \forall x,y\ Sibling(x,y) \Leftrightarrow\ Sibling(y,x)$

Knowledge Representation in Prolog

Fatherhood: $\forall d,c\ Father(c) = d \Leftrightarrow Male(d) \wedge Parent(d,c).$

```
father(Dad,Child) :- male(Dad), parent(Dad,Child).
```

So, what this rule really says is:

Fatherhood: $\forall d,c\ Male(d) \wedge Parent(d,c) \Rightarrow Father(c) = d$

How would I say "Mike is male" ?

```
male(mike).
```

Things to remember:

- Variables are always **upper-cased**.
- Constants are **lower-cased**.
- Commas mean "**and**."
- Everything ends in a **period**.

Defining a family

```
mother(P,C) :- female(P), parent(P,C).
father(P,C) :- male(P), parent(P,C).

wife(W,H)   :- female(W), spouse(W,H).
husband(H,W):- male(H),   spouse(H,W).
```

Defining a family

```
mother(P,C) :- female(P), parent(P,C).
father(P,C) :- male(P), parent(P,C).

wife(W,H)   :- female(W), spouse(W,H).
husband(H,W):- male(H),   spouse(H,W).
```

Rules

```
male(mike).
female(aimee).
female(lila).
spouse(mike, aimee).
spouse(aimee, mike).
parent(mike, lila).
parent(aimee, lila).
```

Facts

Prolog programs consist of rules, facts, and queries!

```
Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 7.3.11)
Copyright (c) 1990-2015 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).


% /Volumes/MacintoshHD2/Users/mhcoen/Dropbox/FileS/Teaching/CS540-15/Lectures/Week
10/family-examples.pl compiled 0.00 sec, 20 clauses
1 ?- father(mike, lila).    % Is mike the father of lila?
true.

2 ?- mother(aimee, lila).   % Is aimee the  mother of lila?
true.

3 ?- father(mike, X).       % Who is mike the father of?
X = lila.

4 ?- mother(X, aimee).      % Who is the mother of aimee?
false.                      % The system doesn't know!

5 ?- mother(X, lila).       % Who is the mother of lila?
X = aimee
false.                      % What's this??

6 ?- father(X, Y).          % Who is the father of who?
X = mike,
Y = lila.

7 ?- father(mike,X), mother(aimee,X).  % Who are aimee and mike the mother and father of?
X = lila.

8 ?-
```

Programs are written in an editor.

Queries are inside the program or typed at the read-eval-print loop.

Defining a family

```
mother(P,C) :- female(P), parent(P,C).
father(P,C) :- male(P), parent(P,C).

wife(W,H)   :- female(W), spouse(W,H).
husband(H,W):- male(H),   spouse(H,W).
```

Rules

```
male(mike).
female(aimee).
female(lila).
spouse(mike, aimee).
spouse(aimee, mike).
parent(mike, lila).
parent(aimee, lila).
```

Facts

Quick check…

```
18 ?- father(X, lila).
X = mike.
```

Okay, that's fine...

Defining a family

```
mother(P,C) :- female(P), parent(P,C).
father(P,C) :- male(P), parent(P,C).

wife(W,H)   :- female(W), spouse(W,H).
husband(H,W):- male(H),   spouse(H,W).

male(X)      :- not(female(X)).
```

Rules

```
female(aimee).
female(lila).
spouse(mike, aimee).
spouse(aimee, mike).
parent(mike, lila).
parent(aimee, lila).
```

Facts

```
1 ?- male(mike).
true.
```

Query

But????

```
1 ?- father(mike,lila).
true.

2 ?- father(X,lila).
false.
```

What's going on here? Why can it prove I am Lila's father,
but it can't derive who the father of Lila is anymore???

It's because this rule

```
father(P,C) :- male(P), parent(P,C).
```

never unifies in the second query!! What fact or goal would P unify with?

All we have is this:

```
male(X)       :- not(female(X)).
```

We can't unify a variable through a "negative" goal.
There is nothing left to do! Prolog proves through *negation*!
We can't prove through a "negative negation!" (Oy! Does my head hurt!!)

But????

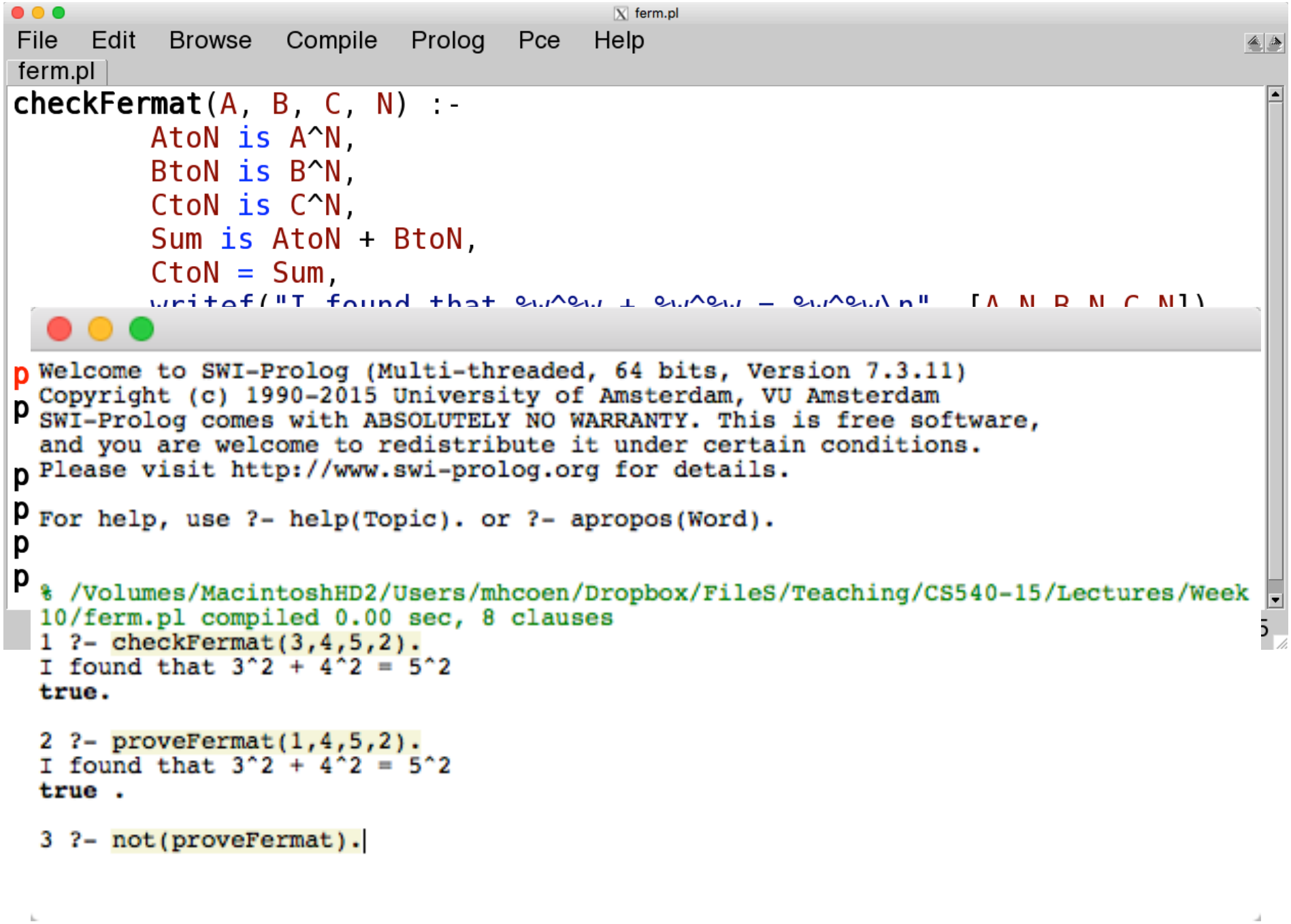
If this had worked, you could write an evil rule like:

```
true(X) :- not(untrue(X)).
```

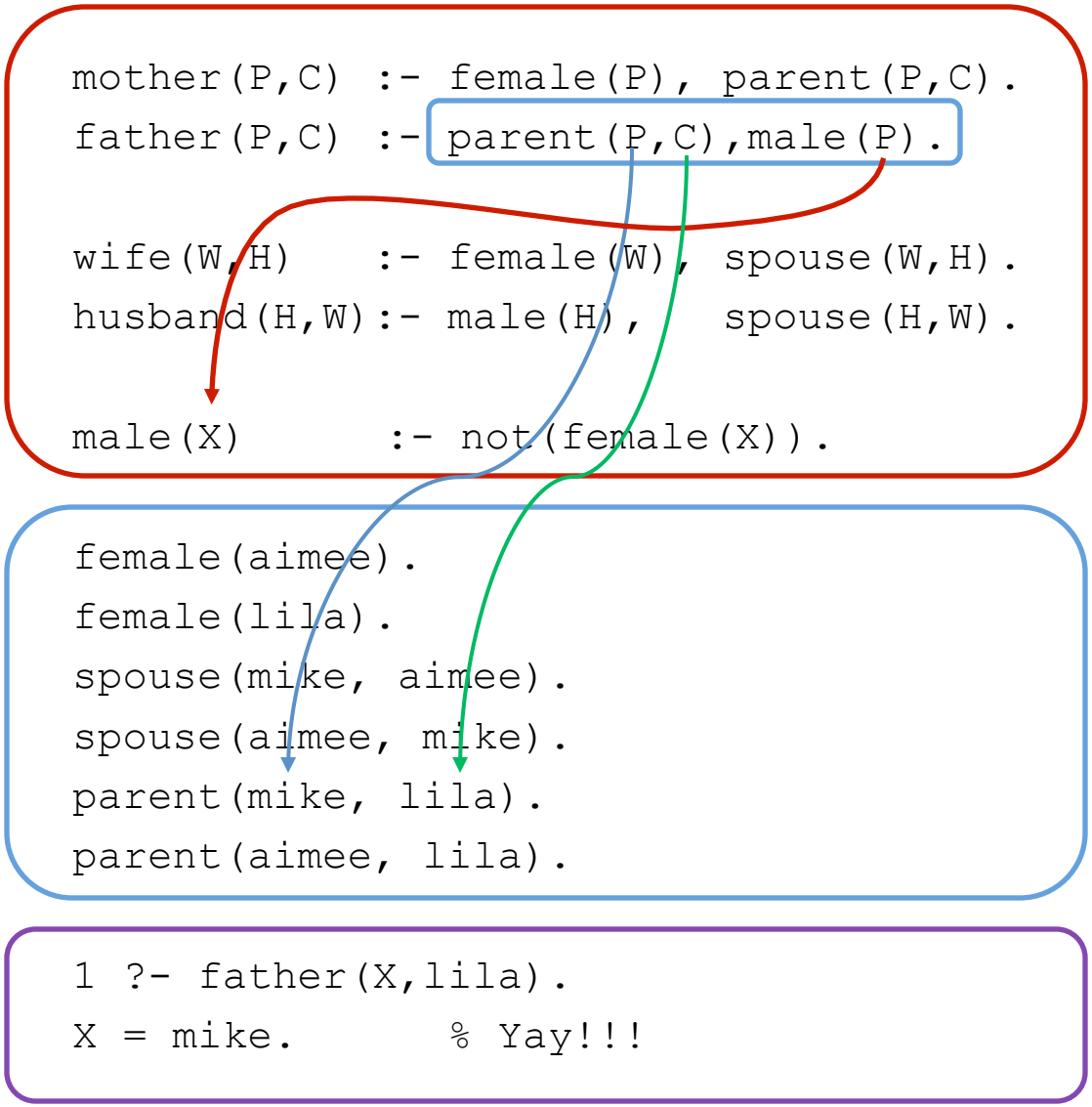
This would **enumerate through all true interpretations of `X`.**

`Not` is unsound in Prolog!

The ghosts of numerous mathematicians would come yell at you if it were.


ferm.pl
File Edit Browse Compile Prolog Pce Help
ferm.pl
checkFermat(A, B, C, N) :-
AtoN is A^N,
BtoN is B^N,
CtoN is C^N,
Sum is AtoN + BtoN,
CtoN = Sum,
Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 7.3.11)
Copyright (c) 1990-2015 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.
For help, use ?- help(Topic). or ?- apropos(Word).
% /Volumes/MacintoshHD2/Users/mhcoen/Dropbox/FileS/Teaching/CS540-15/Lectures/Week 10/ferm.pl compiled 0.00 sec, 8 clauses
1 ?- checkFermat(3,4,5,2).
I found that 3^2 + 4^2 = 5^2
true.
2 ?- proveFermat(1,4,5,2).
I found that 3^2 + 4^2 = 5^2
true .
3 ?- not(proveFermat).

Defining a family



Defining a family

```
mother(P,C) :- female(P), parent(P,C).
father(P,C) :- parent(P,C),male(P).

wife(W,H)   :- female(W), spouse(W,H).
husband(H,W):- male(H),   spouse(H,W).

male(X)      :- not(female(X)).
```

Rules

```
female(aimee).
female(lila).
spouse(mike, aimee).
spouse(aimee, mike).
parent(mike, lila).
parent(aimee, lila).
```

Facts

Defining a family

```
mother(P,C) :- female(P), parent(P,C).
father(P,C) :- parent(P,C),male(P).

wife(W,H)   :- female(W), spouse(W,H).
husband(H,W):- male(H),   spouse(W,H).

male(X)      :- not(female(X)).

spouse(X,Y) :- spouse(Y,X).    % Good?
```

Rules

```
female(aimee).
female(lila).
spouse(mike, aimee).
parent(mike, lila).
parent(aimee, lila).
```

Facts

Defining a family

```
mother(P,C) :- female(P), parent(P,C).
father(P,C) :- parent(P,C),male(P).

wife(W,H)   :- female(W), spouse(W,H).
husband(H,W):- male(H),   spouse(W,H).

male(X)      :- not(female(X)).

spouse(X,Y) :- spouse(Y,X).    % Good?
```

Rules

```
female(aimee).
female(lila).
spouse(mike, aimee).
parent(mike, lila).
parent(aimee, lila).
```

Facts

```
1 ?- spouse(aimee,mike).
ERROR: Out of local stack
   Exception: (246,482) spouse(mike, aimee) ?
```

Query

Defining a family

```
mother(P,C) :- female(P), parent(P,C).
father(P,C) :- parent(P,C),male(P).

wife(W,H)   :- female(W), spouse(W,H).
husband(H,W):- male(H),   spouse(W,H).

male(X)      :- not(female(X)).

spouse(X,Y)  :- married(Y,X). % Good!
spouse(X,Y)  :- married(X,Y). % Good!
```

Rules

```
female(aimee).
female(lila).
married(mike, aimee).
parent(mike, lila).
parent(aimee, lila).
```

Facts

```
1 ?- spouse(aimee,mike).
true .
```

Query

Towards a definition of a family…

Motherhood: $\forall m,c\ Mother(c) = m \Leftrightarrow Female(m) \wedge Parent(m,c).$

Husband: $\forall w,h\ Husband(h,w) \Leftrightarrow Male(h) \wedge Spouse(h,w).$

Disjoint: $\forall x\ Male(x) \Leftrightarrow \neg Female(x).$

Inverse: $\forall p,c\ Parent(p,c) \Leftrightarrow Child(c,p).$

Sibling: $\forall x,y\ Sibling(x,y) \Leftrightarrow x \neq y\ \wedge\ \exists p\ Parent(p,x)\ \wedge\ Parent(p,y)$

Transitivity: $\forall g,c\ Grandparent(g,c) \Leftrightarrow \exists p\ Parent(g,p)\ \wedge\ Parent(p,c).$

Recursion: $\forall a,c\ Ancestor(a,c)\ \Leftrightarrow\ \exists p\ Parent(p,c)\ \wedge\ Ancestor(a,p).$

Recursion: $\forall a,c\ Ancestor(a,c)\ \Leftrightarrow\ \exists r\ Ancestor(r,c)\ \wedge\ Parent(a,r).$

Can we prove things in this system?

$\therefore\ \forall x,y\ Sibling(x,y) \Leftrightarrow\ Sibling(y,x)$

More definitions

```
sibling(X,Y) :- mother(M,X), mother(M,Y), father(F,X),
   father(F,Y), different(X,Y).
different(X,Y) :- not(X = Y).

sisters(X,Y)  :- sibling(X,Y), female(X), female(Y).
brothers(X,Y) :- sibling(X,Y), male(X), male(Y).
```

Rules

```
parent(roberta,jenny).
parent(stanley,jenny).
parent(roberta, mike).
parent(stanley, mike).
```

Facts

```
1 ?- sibling(X,Y).
X = jenny,
Y = mike .
```

Query

Spanning the generations…

```
grandparent(X,Z) :- parent(X,Y), parent(Y,Z).

ancestor(X,Z) :- parent(X,Z).
ancestor(X,Z) :- parent(X,Y), ancestor(Y,Z).
```

Rules

```
parent(stanley, mike).
parent(avraham, stanley).
parent(yitzhok, avraham).
```

Facts

Querying the Generations

1 ?- ancestor(yitzhok, mike).
true .

2 ?- ancestor(yitzhok, lila).
true .

3 ?- ancestor(lila, yitzhok).
false.

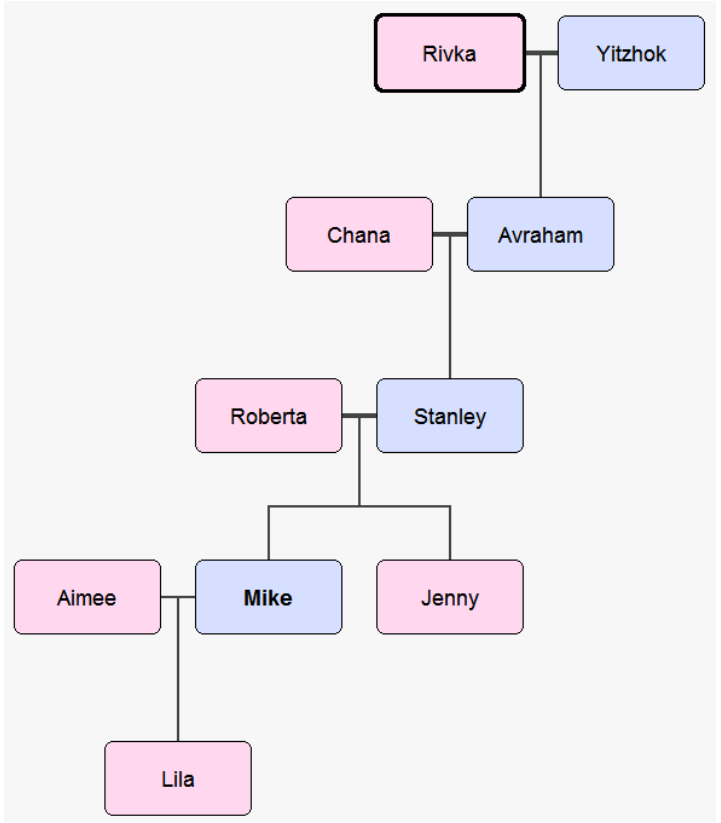
4 ?- ancestor(X,mike).
X = roberta ; % A semicolon triggers a cut!
X = stanley ;
X = avraham ;
X = chana ;
X = yitzhok ;
X = rivka ;
false.

5 ?- ancestor(rivka, X).
X = avraham ;
X = stanley ;
X = jenny ;
X = mike ;
X = lila ;
false.

6 ?- ancestor(X, avraham).
X = yitzhok ;
X = rivka ;
false.

7 ?- ancestor(aimee,X).
X = lila .

8 ?- ancestor(X,lila), ancestor(Y,lila), spouse(X,Y).
X = aimee,
Y = mike ;
X = mike,
Y = aimee ;
false.



Important!

- **Variables are universally quantified**

```
spouse(X,Y)   :- married(Y,X).
```

$$\forall x, y \ \text{married}(x, y) \Rightarrow \text{spouse}(x, y)$$

- **Unless they only appear in the clause of the rule:**

```
ancestor(X,Z) :- parent(X,Y), ancestor(Y,Z).
```

$$\forall x,z\ (\exists y\ \text{parent}(x,y) \wedge \text{ancestor}(y,z) \Rightarrow \text{ancestor}(x,z))$$

Prolog will make up an internal variable to represent y.

What expressive power is this exactly?!

Spanning the generations…

```
ancestor(X,Z) :- parent(X,Z).
ancestor(X,Z) :- parent(X,Y), ancestor(Y,Z).
```

Rules

```
ancestor(X,Z) :- parent(X,Z).
ancestor(X,Z) :- ancestor(X,Y), parent(Y,Z).
```

$Rules_2$

```
ancestor(X,Z) :- parent(X,Z).
ancestor(X,Z) :- ancestor(X,Y),ancestor(Y,Z).
```

$Rules_3$

Which way is better?

Spanning the generations…

```
ancestor(X,Z) :- parent(X,Z).
ancestor(X,Z) :- parent(X,Y), ancestor(Y,Z).
```

Rules

```
ancestor(X,Z) :- parent(X,Y), ancestor(Y,Z).
ancestor(X,Z) :- parent(X,Z).
```

$Rules_{Yuk}$?

What's wrong with this?

Just replacing the operator names…

```
path(X,Z)   ≡ ancestor(X,Z)
link(X,Z)   ≡ parent(X,Z)
```

```
ancestor(X,Z) :- parent(X,Z).
ancestor(X,Z) :- parent(X,Y), ancestor(Y,Z).
```

```
path(X,Z) :- link(X,Z).
path(X,Z) :- link(X,Y), path(Y,Z).
```

Rules

```
path(X,Z) :- link(X,Y), path(Y,Z).
path(X,Z) :- link(X,Z).
```

$Rules_{Yuk}$?

What's wrong with this?

Suppose we have:

```
link(a,b), link(b,c)
```

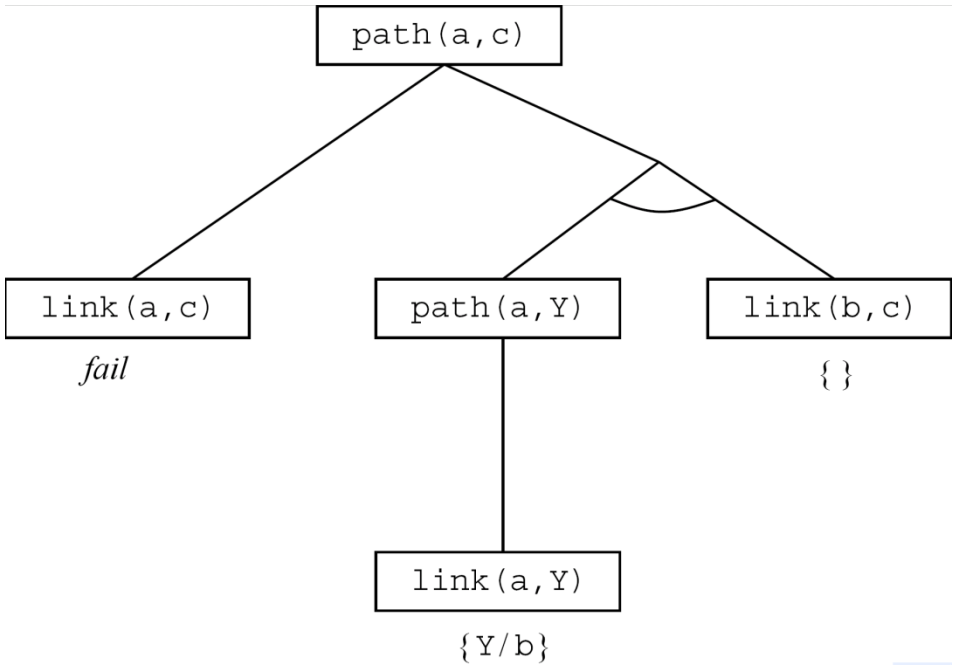
NB: Russell & Norvig book gets this wrong!

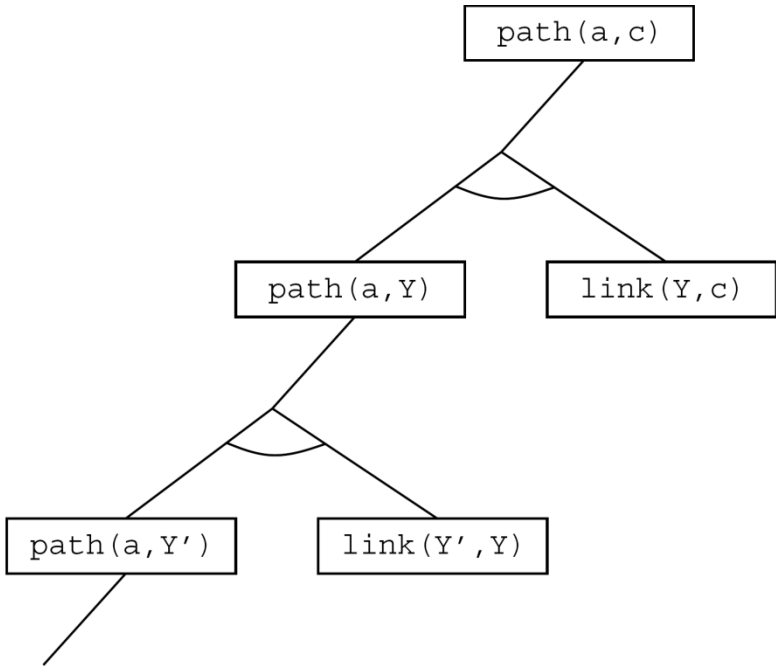
```
path(X,Z) :- path(X,Y), link(Y).
```

Prolog Does Backward Chaining Depth First Search

```
path(X,Z) :- link(X,Z).
path(X,Z) :- link(X,Y), path(Y,Z).
link(a,b), link(b,c)
?- path(a,c).
```

```
path(X,Z) :- link(X,Y), path(Y,Z).
path(X,Z) :- link(X,Z).
link(a,b), link(b,c)
?- path(a,c).
```





Remember this?

```
(define (factorial x)
  (if (<= x 1)
      1
      (* x (factorial (- x 1)))))
```

vs.

```
(define (factorial x)
  (* x (factorial (- x 1)))
  (if (<= x 1)
      um, done???))
```

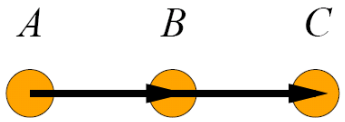
This is the same issue…

```
(define (ancestor X Z)
  (if (parent? X Z)
      true
      (and ∃Y (parent X Y) (ancestor Y Z))))
```

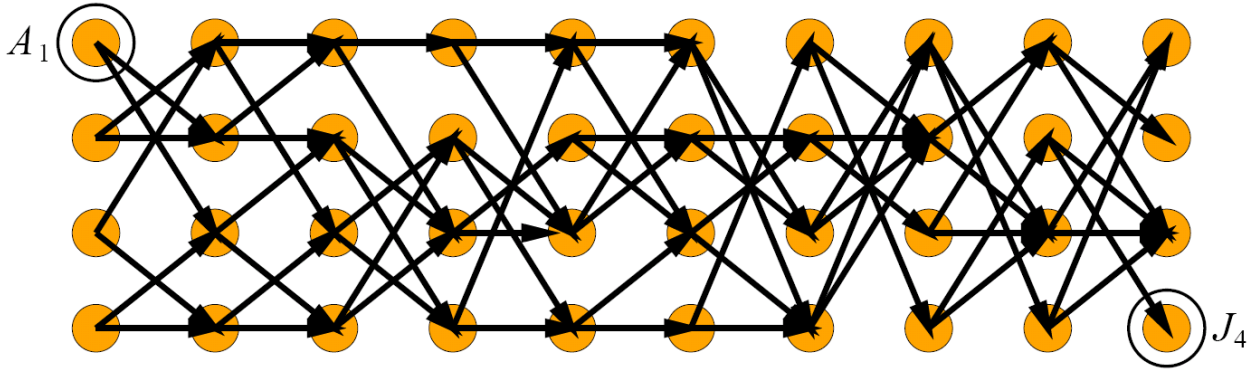
vs.

```
(define (ancestor X Z)
  (and ∃Y(parent X Y) (ancestor Y Z))
  (if (parent? X Z)
      true))
```

Caveats



(a)



(b)

Finding a path from a to c can cause an infinite loop.

Finding a path from a to node J_4 can be very inefficient. Requires 877 inferences in this case because we try to find paths from nodes that can't reach the goal! Forward chaining would only look at 62 inferences.

Memoization (*tabling*) fixes this!
It's like adding dynamic programming.

More issues

- Most Prologs are not sound!
  - Missing something called occur check during unification.
  - Shouldn't unify variable x with something containing variable x, but Prolog doesn't check!

```
data(X, name(X)).
```

Fact

```
1 ?- data(X,Y).
Y = name(X).

2 ?- data(Y,Y).
Y = name(**).
```

Queries

Flags in SWI-Prolog:

```
occurs_check(true).
occurs_check(error).
occurs_check(false).
```

This is like saying $Y = \text{name}(\text{name}(\text{name}(\text{name}(\ldots\text{name}(Y)\ldots))))$ $\to\infty$ number of ti

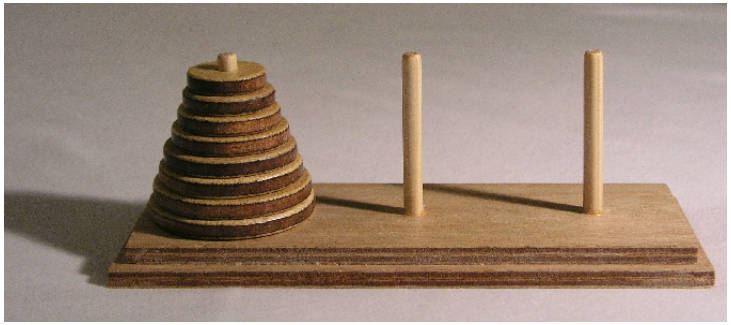
More thoughts

- Prolog allows both side-effects and backtracking.
  - What happens to side-effects when you backtrack???

- Equality (=) in Prolog means unifiable. It **does not** mean logical equality!
  - E.g., **friend(bart, *X*)** ***unifies*** with **friend(*Y*, Milhouse)**
  - But 2.99999999…. might not unify with 3, even though we know 2.99999… = 3
  - And in Prolog, foo ≠bar, even though they might actually be equal in FOL.

- Prolog is a compromise between representational power, inferential power, and efficiency.

Towers of Hanoi



```
move(1,X,Y,_) :- write('Move top disk from '), write(X), write(' to '), write(Y), nl.
move(N,X,Y,Z) :- N>1, M is N-1, move(M,X,Z,Y), move(1,X,Y,_), move(M,Z,Y,X).
```

```
(define (hanoi n)
  (hanoi-helper 'A

(define (hanoi-hel
  (cond ((= n 1)
         (printf "]
        (else
         (hanoi-he
         (hanoi-he
         (hanoi-he
```

?- move(3,left,right,center).
Move top disk from left to right
Move top disk from left to center
Move top disk from right to center
Move top disk from left to right
Move top disk from center to left
Move top disk from center to right
Move top disk from left to right

Lists in Prolog

A list is built with brackets: […]

e.g., [apple, pear, banana, peach], [] (empty list)

```
1 ?- member(2, [1,2,3]).% (What would happen with member(2,X)?)
true .

2 ?- member(X, [1,2,3]).
X = 1 ;
X = 2 ;
X = 3 ;
false.

3 ?- member(2, [1, X, 3]).
X = 2 .
```

First element

```
first([X|_], X).   % _ (underscore) means ignore the value

1 ?- first([1,2,3], X).
X = 1.

2 ?- first([X,2,3], 1).
X = 1.

3 ?- first([1, X, 2,3], 1).
true.

4 ?- first([2, X, 2,3], 1).
false.
```

Lists in Prolog

Append two lists:

append(X, Y, **Z**) is true if Z = [X | Y]. | partitions a list into *head* and *tail*

```
append([],Y,Y).
append([A|X], Y, [A|Z]) :- append(X,Y,Z).
```

```
?- append([1,2], [3,4], Z).
   Z = _G2044
   Z = [1, 2, 3, 4]

   append([1, 2], [3, 4], _G2044)
   _G2044 = [1 | _G2126]
   _G2044 = [1, 2, 3, 4]

   append([2], [3, 4], _G2126)
   _G2126 = [2 | _G2129]
   _G2126 = [2, 3, 4]

   append([], [3, 4], _G2129)
   _G2129 = [3,4]
```

Lists in Prolog

Append two lists:

append(X, Y, **Z**) is true if Z = [X | Y]. | partitions a list into *head* and *tail*

```
append([],Y,Y).
append([A|X], Y, [A|Z]) :- append(X,Y,Z).
```

```
1 ?- append([1,2], [3,4], Z).
T Call: (7) append([1, 2], [3, 4], _G2044)
T Call: (8) append([2], [3, 4], _G2126)
T Call: (9) append([], [3, 4], _G2129)
T Exit: (9) append([], [3, 4], [3, 4])
T Exit: (8) append([2], [3, 4], [2, 3, 4])
T Exit: (7) append([1, 2], [3, 4], [1, 2, 3, 4])
Z = [1, 2, 3, 4].
```

Lists in Prolog

Append two lists:

append(X, Y, **Z**) is true if Z = [X | Y]. | partitions a list into *head* and *tail*

```
append([],Y,Y).
append([A|X], Y, [A|Z]) :- append(X,Y,Z).
```

```
1 ?- append([X, Y, [1,2]).
X = [],
Y = [1, 2] ;
X = [1],
Y = [2] ;
X = [1, 2],
Y = [] ;
false.
```

Reversing a list

Reverse a list:

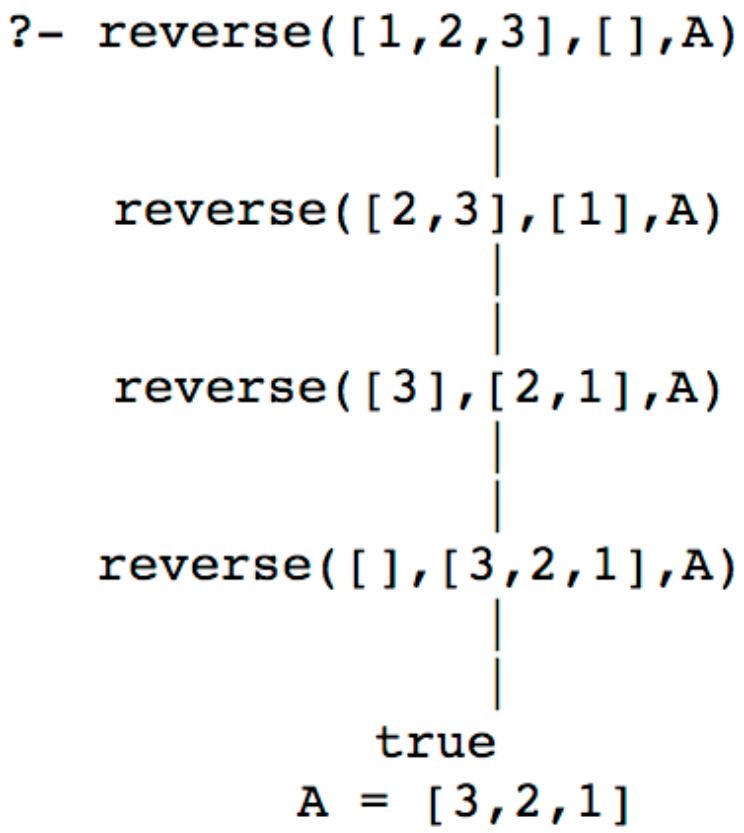
```
reverse([],X,X).
reverse([X|Y],Z,W) :- reverse(Y,[X|Z],W).
reverse(A,R) :- reverse(A,[],R).

1 ?- reverse([1,2,3], X). (or reverse(X, [1,2,3]).)
X = [3, 2, 1].

last(List, Last) :- reverse(List,R), first(R, Last).
1 ?- last([1,2,3], X).
X = 3.
2 ?- last([1,2,X], 17).
X = 17.
3 ?- last([1,2],2).
true.
```

```
last([Elem], Elem).
last([_|Tail], Elem) :- last(Tail,Elem).
```

Visualizing This

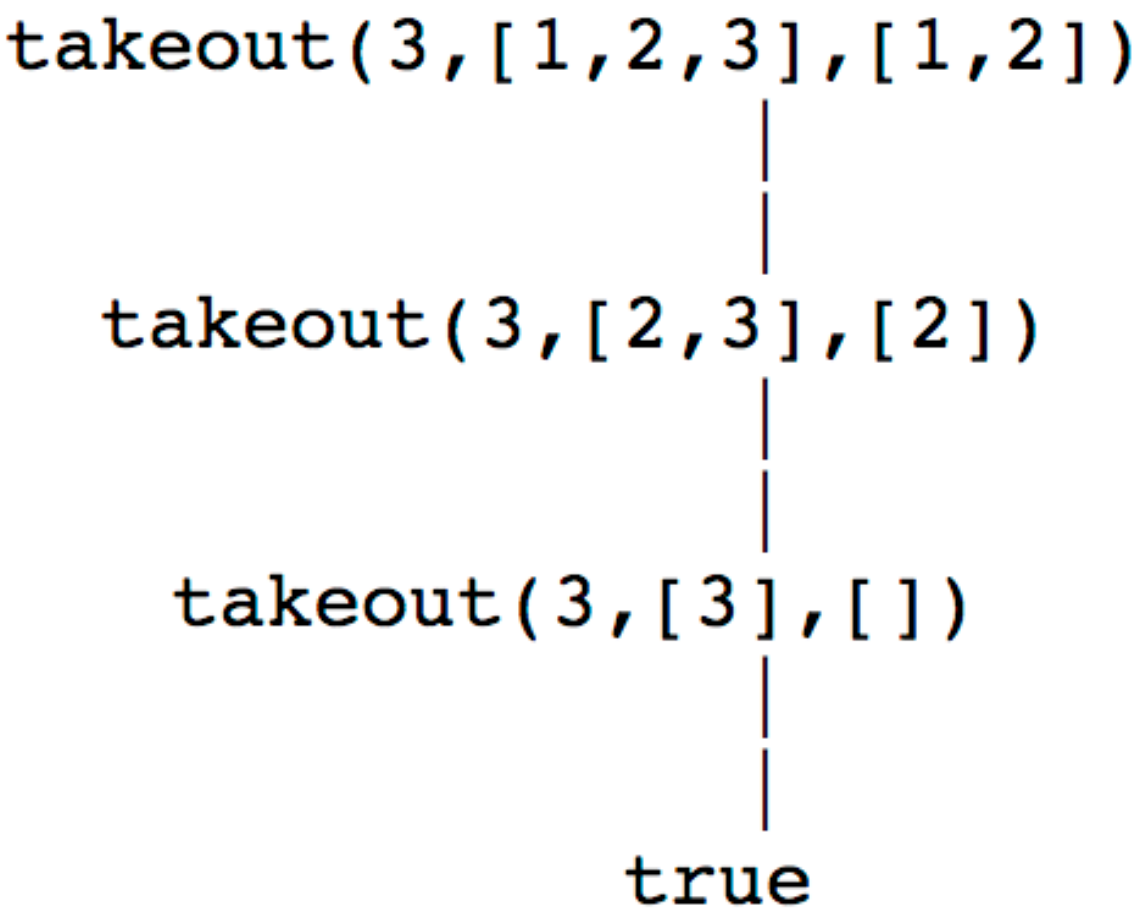


Permutations

```
takeout(X,[X|R],R).
takeout(X,[F|R],[F|S]) :- takeout(X,R,S).
perm([],[]).
perm([X|Y],Z) :- perm(Y,W), takeout(X,Z,W).

1 ?- perm([1,2,3], P).
P = [1, 2, 3] ;
P = [2, 1, 3] ;
P = [2, 3, 1] ;
P = [1, 3, 2] ;
P = [3, 1, 2] ;
P = [3, 2, 1] ;
false.
```

Verifying rather than computing



Permutations

```
takeout(X,[X|R],R).
takeout(X,[F|R],[F|S]) :- takeout(X,R,S).
perm([],[]).
perm([X|Y],Z) :- perm(Y,W), takeout(X,Z,W).

1 ?- perm([1,2,3,4, 5], P), first(P, 2), last(P, 4).
P = [2, 1, 3, 5, 4] ;
P = [2, 3, 1, 5, 4] ;
P = [2, 3, 5, 1, 4] ;
P = [2, 1, 5, 3, 4] ;
P = [2, 5, 1, 3, 4] ;
P = [2, 5, 3, 1, 4] ;
false.
```

Permutations

```
sumlist([],0).
sumlist([X|Rest],Sum) :- sumlist(Rest, Sum1), Sum is X + Sum1.

1 ?- perm([1,2,3,4], P), first(P,X), last(P,Y), sumlist([X, Y],
   5).
P = [1, 2, 3, 4];
P = [1, 3, 2, 4];
P = [3, 1, 4, 2];
P = [3, 4, 1, 2];
P = [2, 1, 4, 3];
P = [2, 4, 1, 3];
P = [4, 2, 3, 1];
P = [4, 3, 2, 1];

false.
```

Revisiting Lists & Append

A list is built with brackets: […]

[apple, pear, banana, peach], [] (Ø or the empty list)

The | operator unifies this way:

[Head | Tail] = [apple | pear, banana, peach]

Examining unification:

[X|Y] unifies with [a,b,c] with the unifier {X = a, Y = [b,c]}.

[X|Y] unifies with [a,b,c,d] with the unifier {X = a, Y = [b,c,d]}.

[X|Y] unifies with [a] with the unifier {X = a, Y = []}.

[X|Y] does not unify with [].

Revisiting Append

Append two lists:

append(X,Y,Z) is true if Z = [X | Y]. | partitions a list into *head* and *tail*

```
append([],List,List).
append([H|T], List, [H|New]) :- append(T,List,New).
```

Let's examine: **`append([],List,List).`**

Query	Unifier	Prolog Output
?- append([],[b,c,d],[b,c,d]).	{List = [b,c,d]}	True.
?- append([],[b,c,d],X).	{List = [b,c,d], X = [b,c,d]}	X = [b,c,d]
?- append(X,Y,Z).	{X = [], Y = List, Z = List}	X = [], Y = Z
?- append([],[b,c],[b,c,d]).	{}	False.

Revisiting Append

Append two lists:

append(X,Y,Z) is true if Z = [X | Y]. | partitions a list into *head* and *tail*

```
append([],List,List).
append([H|T], List, [H|New]) :- append(T,List,New).
```

Let's examine:

```
append([H|T], List, [H|New]) :- append(T,List,New).
```

One way to think about this:

> If we have **`append(T,List,New)`** is true, meaning **New = [T, List]** , then adding something to the beginning of **T** and **New** doesn't change the interpretation, so **`append([H|T], List, [H|New])`** is also true, implying **`[H|New] = [[H|T], List].`**

If this makes sense, then just think about it backwards!

Revisiting Append

Append two lists:

append(X,Y,Z) is true if Z = [X | Y]. | partitions a list into *head* and *tail*

```
append([],List,List).
append([H|T], List, [H|New]) :- append(T,List,New).
```

Let's examine:

```
append([H|T], List, [H|New]) :- append(T,List,New).
```

```
?- append([1,2,3], [4,5,6], Result).
```

Unifies: { H = 1, T = [2,3], List = [4,5,6], Result = [1|New] }

```
?- append([2,3], [4,5,6], New).
```

Unifies: { H = 2, T = [3], List = [4,5,6], Result = [2|New] }

```
?- append([3], [4,5,6], New).
```

Unifies: { H = 3, T = [], List = [4,5,6], Result = [3|New] }

```
?- append([], [4,5,6], New).
```

Revisiting Append

```
(1) append([],List,List).
(2) append([H|T], List, [H|New]) :- append(T,List,New).
?- append([1,2,3], [4,5,6], Result).
```

Unifies: { H = 1, T = [2,3], List = [4,5,6], Result = [1|New] }

```
?- append([2,3], [4,5,6], New).
```

Unifies: { H = 2, T = [3], List = [4,5,6], Result = [2|New'] }

```
?- append([3], [4,5,6], New').
```

Unifies: { H = 3, T = [], List = [4,5,6], Result = [3|New''] }

```
?- append([], [4,5,6], New'').  (This now matches (1)!)
```

Unifies: {New'' = [4,5,6] }

```
?- append([], [4,5,6], [4,5,6]).
?- append([3], [4,5,6], [3,4,5,6]).
?- append([2,3], [4,5,6], [2,3,4,5,6]).
?- append([1,2,3], [4,5,6], [1,2,3,4,5,6]).
```

Lists in Prolog

Append two lists:

append(X,Y,Z) is true if Z = [X | Y]. | partitions a list into *head* and *tail*

```
append([],List,List).
append([H|T], List, [H|New]) :- append(T,List,New).

1 ?- append([1], [2,3], Z).
Z = [1, 2, 3].

2 ?- append([1], [2,3], Z).
 T Call: (7) append([1], [2, 3], _G751)
 T Call: (8) append([], [2, 3], _G826)
 T Exit: (8) append([], [2, 3], [2, 3])
 T Exit: (7) append([1], [2, 3], [1, 2, 3])
Z = [1, 2, 3].
```

Lists in Prolog

Append two lists:

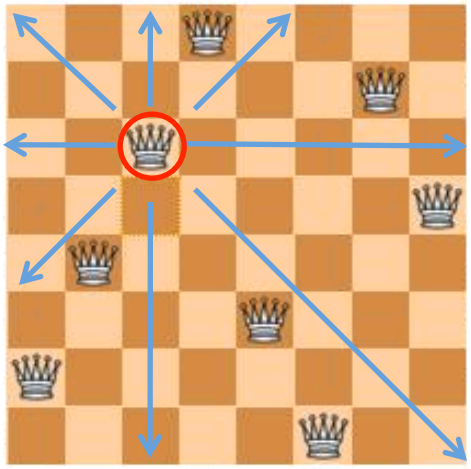
append(X,Y,Z) is true if Z = [X | Y]. | partitions a list into *head* and *tail*

```
append([],List,List).
append([H|T], List, [H|New]) :- append(T,List,New).

1 ?- append([1,2,3], [4,5,6], Z).
Z = [1, 2, 3, 4, 5, 6].

2 ?- append([1,2,3],[4,5,6],Z).
 T Call: (7) append([1, 2, 3], [4, 5, 6], _G777)
 T Call: (8) append([2, 3], [4, 5, 6], _G861)
 T Call: (9) append([3], [4, 5, 6], _G864)
 T Call: (10) append([], [4, 5, 6], _G867)
 T Exit: (10) append([], [4, 5, 6], [4, 5, 6])
 T Exit: (9) append([3], [4, 5, 6], [3, 4, 5, 6])
 T Exit: (8) append([2, 3], [4, 5, 6], [2, 3, 4, 5, 6])
 T Exit: (7) append([1, 2, 3], [4, 5, 6], [1, 2, 3, 4, 5, 6])
Z = [1, 2, 3, 4, 5, 6].
```

N-Queens



If two queens can attack each other, then one of the following must be the case:

- They must be on the same column.
- They must be on the same row.
- The sum of the row and column must be the same for both.
- The difference of the row and column must be the same for both.

To solve N-Queens, we simply need to be able to describe a bad solution.
We don't need to know how to "generate" a good one!

Check that a position is safe

```
queenDoesntHit(_,_,[]) :- !.

queenDoesntHit(Col,Row_dist,[A_queen_col|OtherQueens]) :- !,
    Diag_hit_col1 is Col + Row_dist, A_queen_col =\= Diag_hit_col1,
    Diag_hit_col2 is Col - Row_dist, A_queen_col =\= Diag_hit_col2,
    Row_dist1 is Row_dist + 1, queenDoesntHit(Col,Row_dist1,OtherQueens).


safePosition([_]) :- !. % A single queen position is always safe

safePosition([A_queen|OtherQueens]) :- !, queenDoesntHit(A_queen,
    1,OtherQueens), safePosition(OtherQueens).
```

How many ways to make change from a dollar?

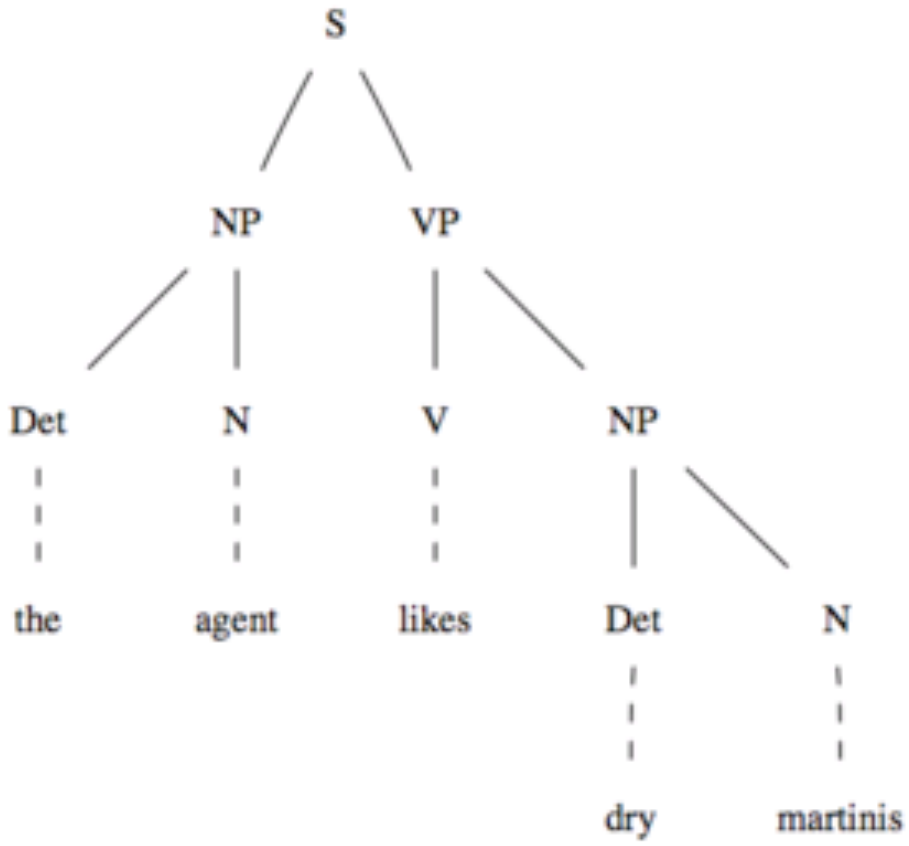
```
change([H,Q,D,N,P]) :-
    member(H,[0,1,2]),                          /* Half-dollars */
    member(Q,[0,1,2,3,4]),                      /* quarters     */
    member(D,[0,1,2,3,4,5,6,7,8,9,10]) ,        /* dimes        */
    member(N,[0,1,2,3,4,5,6,7,8,9,10,           /* nickels      */
              11,12,13,14,15,16,17,18,19,20]),
    S is 50*H + 25*Q +10*D + 5*N,
    S =< 100,
    P is 100-S.
```

```
?- findall(T, change([H,Q,D,N,P]), B), length(B,C).
```

Note: It's easier to show one solution exists than to count all possible solutions!

Parsing in Prolog

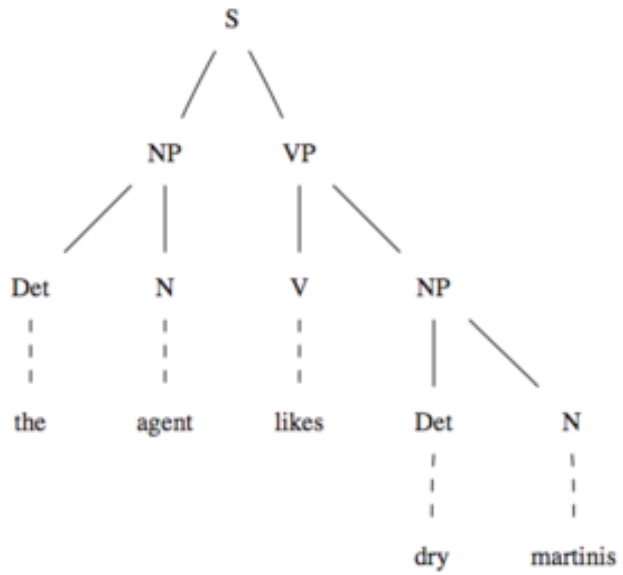
“The agent likes dry martinis.”



A *Very* Simple Grammar

```
S      → np, vp.
np     → det, n.
vp     → v, np.
vp     → v.
det    → [the].
det    → [a].
det    → [dry].
n      → [agent].
n      → [hero].
n      → [martinis].
v      → [likes].
v      → [drinks].
```

This is called a definite clause grammar (DCG). Prolog can automatically turn these into Prolog programs!




```
?- parse(Structure, [the,agent,likes,dry,martinis]).
Structure = s(np(the, agent), vp(likes, np(dry, martinis)))
```

Using Probabilistic Prolog

- Suppose we are less interested in whether something is true but rather **how likely** it is to be true.

- We want to attach probabilities to our rules in Prolog and guide the search along the probabilistic most likely path!

- So, we assign every rule α_i :- β_i a unique probability (p_i)

  α_i :- β_i (p_i)

- Two questions:
  - When would we want to do this?
  - How do we know what the probabilities should be?
  - How does this change Prolog's semantics?

Remember this game? ?

We'll call it the *next number* game:

Given an ordered sequence of numbers, come up with the next number!

1, 2, 4, ?

6? (Even numbers) No, how do you explain the 1?

8? (Powers of 2) Seems good!

Joint work with Yue Gao.

(To appear in *Proceedings of the Thirty-First Annual Conference of the Cognitive Science Society*. Amsterdam, Netherlands. 2009.)

1,8,27,…

64

Why? This is the sequence x^3.

2, 3, 5, 7,…

11

Why? These are the prime numbers.

But is it always so obvious?

1, 2, 4, ?

6, n = prime numbers – 1
1, 2, 4, 6, 10, 12, 16, …

7, $n = i(i+1)/2+1$
1, 2, 4, 7, 11, 16, 22, 29, …

8, $n = 2^i$
1, 2, 4, 8, 16, 32, 64, …

9, n = partial sums of Catalan numbers
1, 2, 4, 9, 23, 65, 197, …

The Problem of Inductive Bias

- How do you make a guess with so little data?!
  - Classic problem in philosophy and AI!
- We *innately* prefer certain explanations to others.
  - Called the ***inductive bias***
- For computers to reason as people do, they need to share our innate preferences!
- But what are our innate cognitive biases?!

A Grammar for Mental Arithmetic

- Expression → PrefixOp (Expression) p_1
- Expression → Expression InfixOp Expression p_2
- Expression → $Previous_{i-1}$ p_3 | $Previous_{i-2}$ p_4 | $Previous_{i-3}$ p_5
- Expression → Number p_6
- Expression → Index p_7
- PrefixOp → exp p_8 | log p_9 | sin p_10 | cos p_11 | tan p_12
- PrefixOp → floor p_13 | ceiling p_14 | mod p_15 | rem p_16 | prime p_17
- InfixOp → + p_18 | − p_19 | × p_20 | ÷ p_21 | ^ p_22
- Number → SmallNum | LargeNum | SpecialNum
- SmallNum → [-9 p_26,…,9 p_45]
- LargeNum → [-50,…, -11, 11, …, 50] p_46
- SpecialNum → -100 p_47 | -10 p_48 | ¼ p_49 | ½ p_50
- SpecialNum → π p_51 | 10 p_52 | 100 p_53
- Index → [1,…,10] p_54

Collect Data of People Solving Problems

- Consider the sequence [1, 4, 9]. We found that all subjects predicted the next number would be *16*, but provided two syntactically different but numerically equivalent generating formulae.

  40% guessed: f(*index*) = $index^2$

  60% guessed: f(*index*) = $Previous_{index-1} + 2 \times index + 1$

Who would have imagined this??

Collect Data of People Solving Problems

- Consider the sequence [1,2,10]. That this sequence is in some sense more difficult was apparent because subjects spent more time studying it, often commenting it felt "difficult" or under constrained.

  All but one: $f(index) = Previous_{index\text{-}1} + (index - 1)^3$

  Yielding: [1, 2, 10, 37, 101, …]

  One guess: $f(index) = Previous_{index\text{-}1} + (Previous_{index\text{-}1})^3$

  Yielding: [1, 2, 10, 1010, 1.0303×10^9,…]

Structurally quite similar.

Collect Data of People Solving Problems

- Consider the sequence [0,7,26], where all the subjects agreed on the next element (*63*) and on the generating formula:

  $f(index) = index^3 - 1$

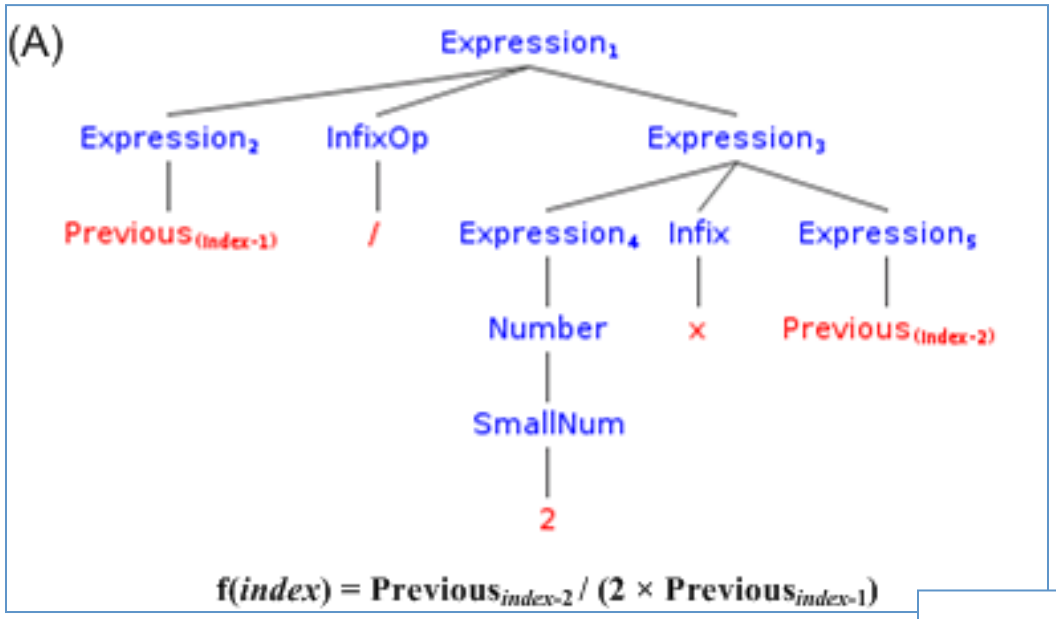
Here, cubing is the "simplest" explanation for people.
We had to pick a sequence to force them to cube.

Deriving the Probabilities

Production Rule	Probability
Expression → PrefixOp (Expression)	0.00402
Expression → Expression InfixOp Expression	0.349
Expression → Previous_{i-1}	0.177
Expression → Previous_{i-2}	0.0321
Expression → Number	0.317
Expression → Index	0.104
InfixOp → +	0.388
InfixOp → −	0.143
InfixOp → ×	0.263
InfixOp → ÷	0.0388
InfixOp → ^	0.163
SmallNum → -1	0.04
SmallNum → 1	0.24
SmallNum → 2	0.40
SmallNum → 3	0.08
SmallNum → 4	0.04
…	…

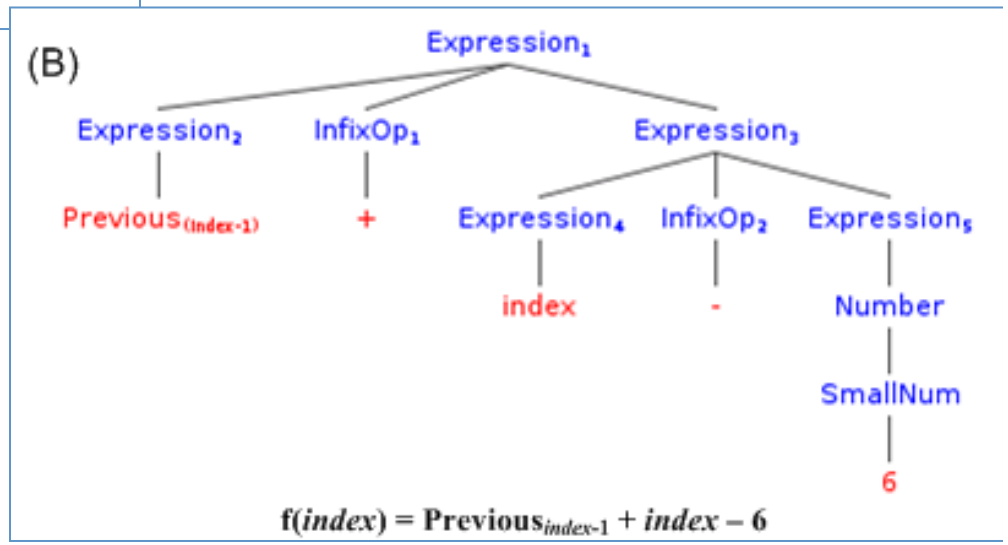
From a *corpus* of solutions, we can derive the probabilities on each rule using the inside-outside algorithm.

The sequence [8,4,1]

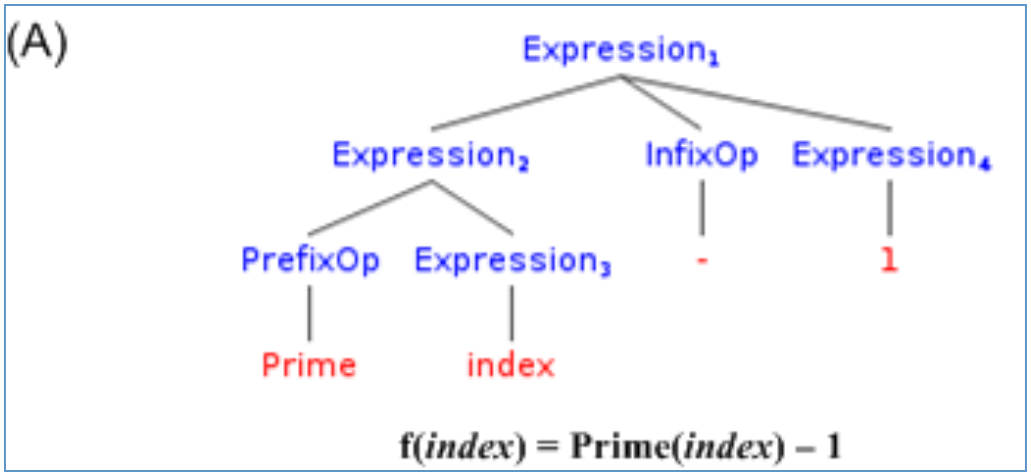


(A) is before training

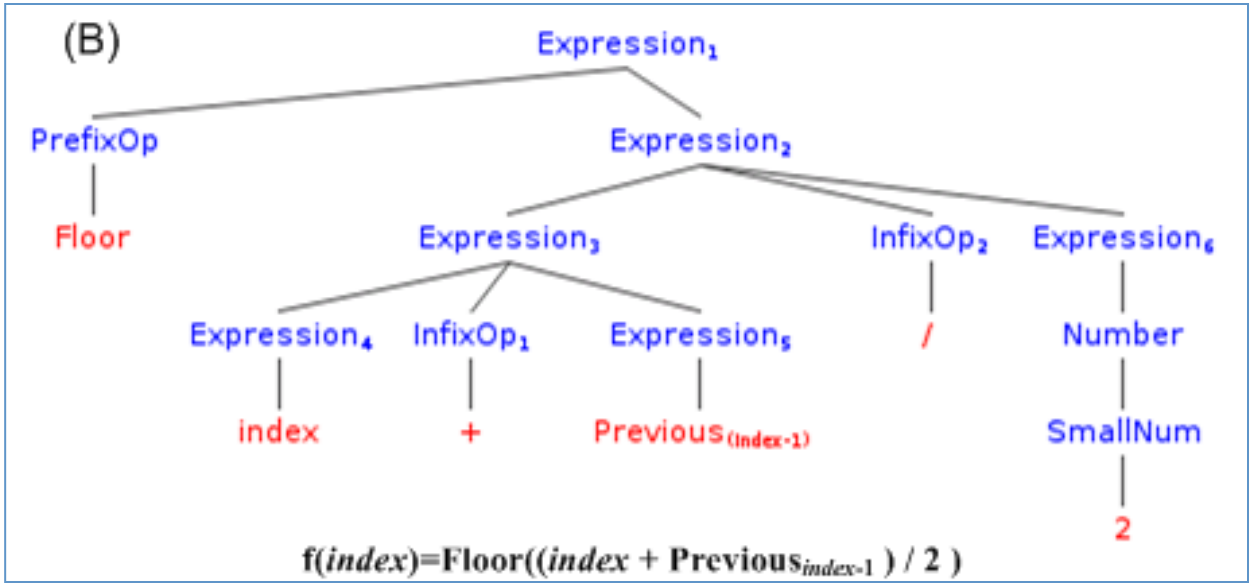
(B) is after training.
88% of humans select this answer.



The sequence [1, 2, 4, 6]



The system predicts (A) is 91% likely. The second most likely solution, in (B), has probability of 1%.



Probabilistic Prolog

- A very interesting tool for modeling cognition
  - Predicting human behavior is also commercially valuable!
- We need a *corpus* of data to train our system.
  - This is how we derive the probabilities.
  - Alternatively, you might know them in advance.
- The probabilities capture the inductive bias of the sample population represented in the corpus.
- There is also inductive bias reflected in the structure of the grammar, but learning a grammar is much more difficult.
  - For math, it seems unnecessary.
- We use a programming language called Prism to implement this.
  - Uses the Viterbi algorithm to derive the most likely proofs.
  - A very different notion than a traditional proof in mathematics.