Logic Programming

CS 4301/6371: Advanced Programming Languages

Kevin W. Hamlen

March 26 – April 2, 2024

FP vs. LP

- Functional Programming
  - centered around first-class functions
  - strong, parametric polymorphic type systems
  - single-assignment
  - operational semantics based on λ-calculus
- Logic Programming
  - centered around *relations*
  - no type system
  - no explicit assignment operation(!)
  - operational semantics based on unification and depth-first search

Relations

- Relation
  - **Definition (relation):** A *relation* is a cartesian product $A \times B$ of two sets A and B.
  - Example: $\leq$ relation over $\mathbb{N} \times \mathbb{N}$: $\{(0,0),(0,1),(1,1),(0,2),(1,2),(2,2),\ldots\}$
- Relations generalize functions.
  - Recall: We write (partial) functions $f : A \rightharpoonup B$ as sets of pairs $A \times B$.
  - Relations (as defined above) are also sets of pairs.
  - Function f encodes relation $\{(x, f(x)) \mid x \in f^{\leftarrow}\}$
  - Unlike functions, relations can map the same domain element to multiple different range elements.

Relational Programming

- Three ways to define a function/relation:
  - Imperatively:

$$factorial(x) := \{z := 1;\ \textbf{for}\ i := 1\ \textbf{to}\ x\ \textbf{do}\ z := z * i;\ \textbf{return}\ z\}$$

  - Functionally:

$$factorial(x) := (\textbf{if}\ x \leq 0\ \textbf{then}\ 1\ \textbf{else}\ x * factorial(x-1))$$

  - Relationally:

$$factorial(0, 1).$$
$$factorial(x, y)\ \textbf{if}\ factorial(x-1, y/x).$$

- Note the differences in approach:
  - Imperative style is an operational recipe.
    - You are essentially doing the compiler's job.
    - Compiler must reverse-engineer your code to optimze it!
  - Functional is a mathematical recipe.
    - better, but still somewhat operational
  - Relational defines necessary and sufficient conditions.
    - Compiler creates a search algorithm for the solution
    - Implementation details abstracted away from programmer
    - Search algorithm can be highly optimized by language implementation

Prolog Programming

- Prolog programs consist of:
  - facts (unconditional truths)
  - rules (conditional truths)
  - queries (cause the program to "run" by initiating a search for a solution to a question)
- Example: factorial program

```
factorial(0,1).
factorial(X,Y) :- X2 is X-1, factorial(X2,Y2), Y is X*Y2.
```

```
?- factorial(5,X).
X = 120
```

LP Applications

- Originally invented by Robert Kowalski (for theorem-proving) and Alain Colmeraur (for NLP) [1973]
- Now used primarily for:
  - artificial intelligence
  - scheduling problems
  - databases (Datalog)
  - model-checking
  - compilers
  - software engineering (verification, etc.)
  - network protocol analysis
  - many other applications...

Running Prolog

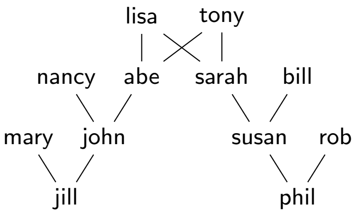
- One Prolog programming assignment (see eLearning)
- Two installation options:
  - Install SWI Prolog on your machine (see link on course web page)
  - Use CS Dept linux machines to do the assignment
- Programming
  - Create a text file name "lastname.pl".
  - Text file contains facts and rules (no queries)
- Running your program
  - Type "pl" at the Unix prompt.
  - Type "consult(*lastname*)." at the Prolog prompt.
  - Enter queries at the Prolog prompt.
  - To reload after changing programs, just type "make."
  - Exit by hitting Control-C then pressing "e".

Prolog Syntax

- Each program line has one of two forms:
  - $\mathrm{p}(t_1,\ldots,t_n)$.
  - $\mathrm{p}(t_1,\ldots,t_n)$:- $\mathrm{p}_1(t_1,\ldots,t_i)$, $\mathrm{p}_2(t_1,\ldots,t_j)$, ..., $\mathrm{p}_m(t_1,\ldots,t_k)$.
  - Don't forget the period ending each line!
  - p is a *predicate* consisting of lower-case letters (e.g., "factorial").
  - $t_1, \ldots, t_n$ are *terms* (defined below)
- Terms can be:
  - integer constants (1, -13, etc.)
  - atoms (non-numerical constants)
    - consist of lower-case letters or surrounded by single-quotes
    - Examples: x, abc, 'Foo'
  - variables (captialized identifiers)
    - Examples: X, Foo
  - structures (tree-shaped data structures)
    - Examples: foo(3,12), foo(foo(13),foo(16,12))
    - Warning: Syntax resembles predicates but means something completely different!
    - No type system, so be careful!

Example: Family Tree Relational Data Structure

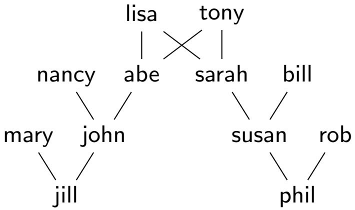
```
father(tony,abe).
father(tony,sarah).
father(abe,john).
father(bill,susan).
father(john,jill).
father(rob,phil).
mother(lisa,abe).
mother(lisa,sarah).
mother(nancy,john).
mother(sarah,susan).
mother(mary,jill).
mother(susan,phil).
```



Reasoning About Family Trees

Q1: How might we decide parent relations?

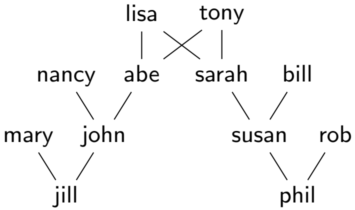
```
parent(X,Y) :-
```



Reasoning About Family Trees

Q1: How might we decide parent relations?

```
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
```



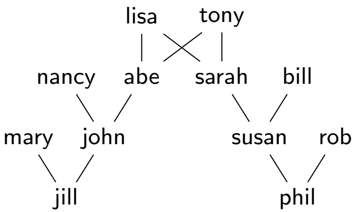
Reasoning About Family Trees

Q1: How might we decide parent relations?

```
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
```

Q2: Grandparent?

```
gp(X,Y) :-
```



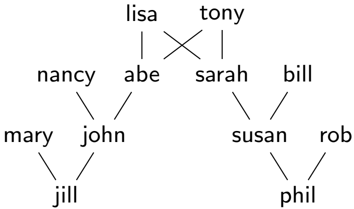
Reasoning About Family Trees

Q1: How might we decide parent relations?

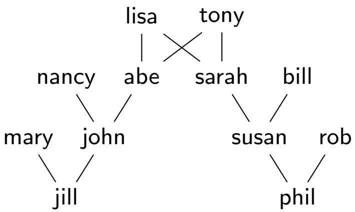
```
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
```

Q2: Grandparent?

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



Reasoning About Family Trees



Q1: How might we decide parent relations?

```
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
```

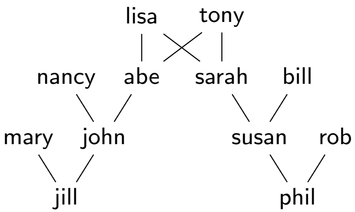
Q2: Grandparent?

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

Q3: Great-grandparent?

```
ggp(X,Y) :-
```

Reasoning About Family Trees



Q1: How might we decide parent relations?

```
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
```

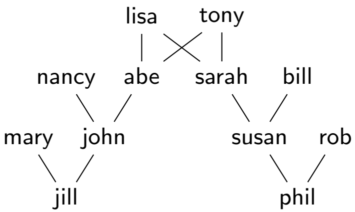
Q2: Grandparent?

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

Q3: Great-grandparent?

```
ggp(X,Y) :- gp(X,Z), parent(Z,Y).
```

Reasoning About Family Trees



Q1: How might we decide parent relations?

```
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
```

Q2: Grandparent?

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

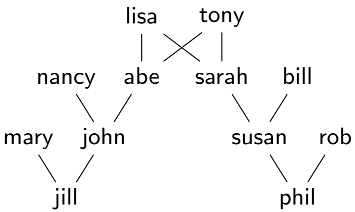
Q3: Great-grandparent?

```
ggp(X,Y) :- gp(X,Z), parent(Z,Y).
```

Q4: Ancestor?

```
ancestor(X,Y) :-
```

Reasoning About Family Trees



Q1: How might we decide parent relations?

```
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
```

Q2: Grandparent?

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

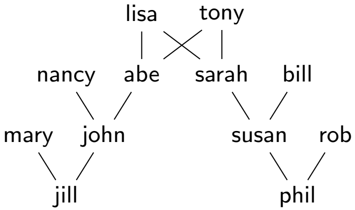
Q3: Great-grandparent?

```
ggp(X,Y) :- gp(X,Z), parent(Z,Y).
```

Q4: Ancestor?

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

Query Examples




```
?- father(abe,john).
true.

?- father(tony,X).
X = abe ;     (user presses semicolon)
X = sarah.

?- parent(X,susan).
X = bill ;     (user presses semicolon)
X = sarah ;     (user presses semicolon)
false.

?-
```

Queries

- typed at Prolog prompt (not in external files)
- consist of a predicate possibly containing variables
  - if no variables, result is either `true` or `false`
  - otherwise, result is an instantiation of variables or `false`
- no solutions, one solution, or many solutions
  - no solution: `false`
  - after printing one solution, Prolog waits for user input
  - hit ⟨RETURN⟩ to stop search; Prolog says `true`
  - hit ; to find more solutions; Prolog either finds another and waits for more input or says `false`
- convergence not guaranteed!
  - queries can diverge (i.e., loop infinitely)
  - hit ⟨CTRL-C⟩ to interrupt, then “a” to abort

Search Procedure

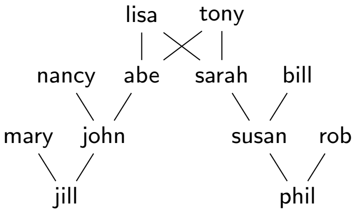
- How does Prolog search for query solutions?
- Three internal data structures:
  - search tree in which each node has ...
  - a list of goals (predicates), and
  - a set of variable bindings (instantiations)
- Two important concepts:
  - **unification:** find instantiation of vars to make equal terms (if such instantiation exists)
  - **back-tracking:** revisiting past decisions after a failed goal is reached

Search Procedure

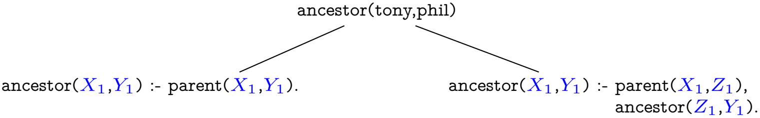
- Initially...
  - search tree has just a root node
  - goal list consists only of the query
  - set of variable bindings is empty
- **Step 1:** Scan file from **top to bottom** for a fact or rule whose lhs potentially matches the current goal.
  - for each such fact/rule, add a child node to the search tree
  - descend to the leftmost child
- **Step 2:** Unify the top goal with this rule's lhs, yielding more variable instantiations
- **Step 3:** Add all rhs predicates to goal list, **left to right**
- Return to Step 1.
- Steps 1 or 2 may fail
  - no matching rule or failed unification
  - if so, backtrack to parent node and try next child
  - if root node fails, stop and return `false`

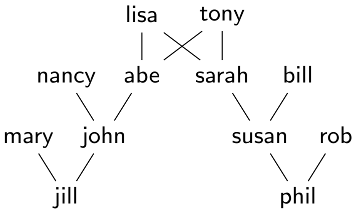
Search Example

ancestor(tony,phil)



Search Example



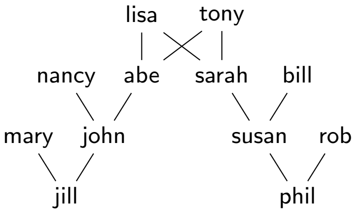


Search Example

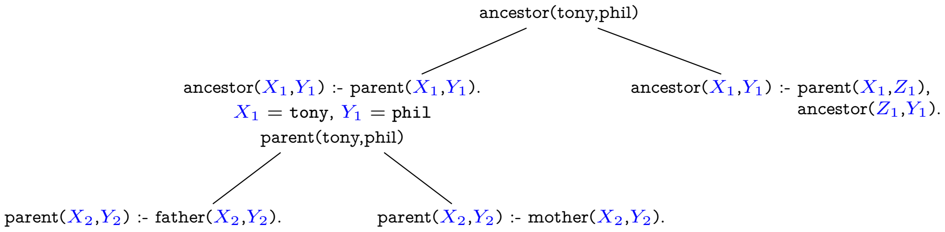
ancestor(tony,phil)

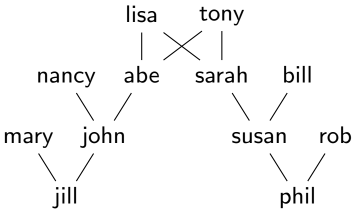
ancestor(X_1,Y_1) :- parent(X_1,Y_1).
X_1 = tony, Y_1 = phil
parent(tony,phil)

ancestor(X_1,Y_1) :- parent(X_1,Z_1),
ancestor(Z_1,Y_1).

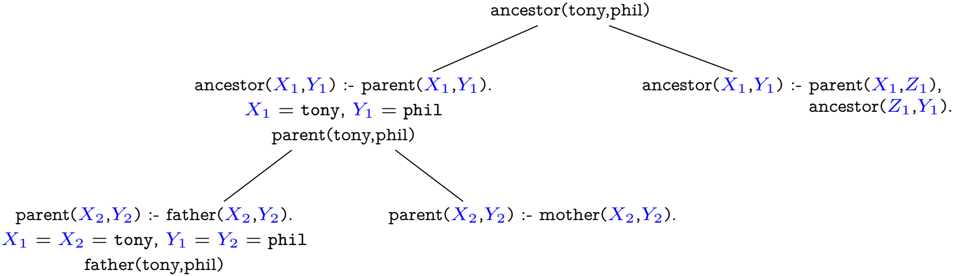


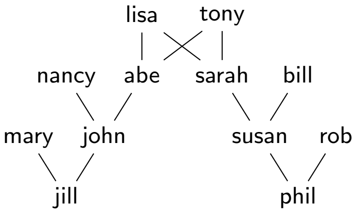
Search Example



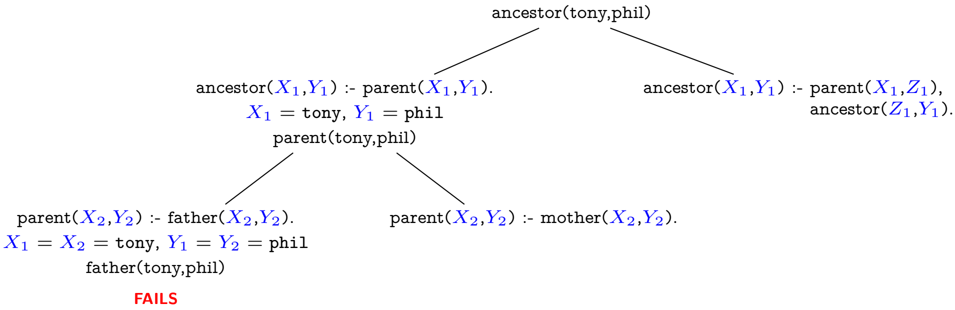


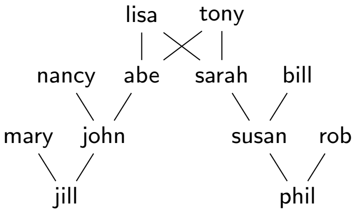
Search Example



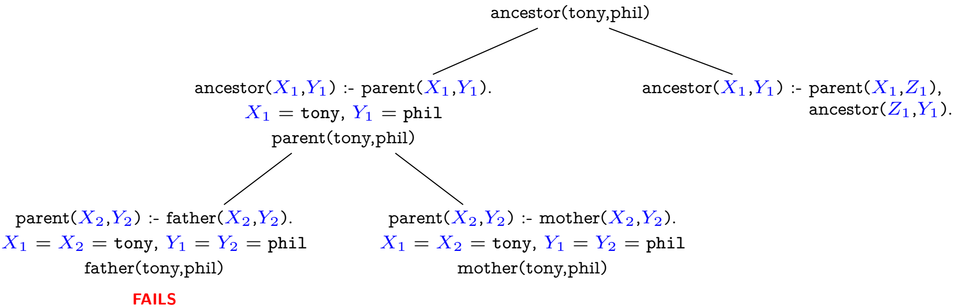


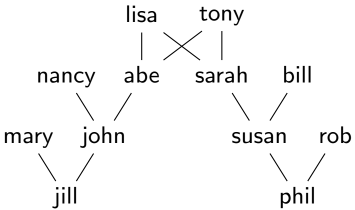
Search Example





Search Example





Search Example

ancestor(tony,phil)

ancestor(X_1,Y_1) :- parent(X_1,Y_1).
X_1 = tony, Y_1 = phil
parent(tony,phil)

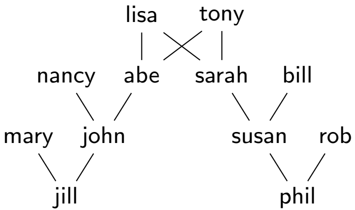
parent(X_2,Y_2) :- father(X_2,Y_2).
X_1 = X_2 = tony, Y_1 = Y_2 = phil
father(tony,phil)

FAILS

parent(X_2,Y_2) :- mother(X_2,Y_2).
X_1 = X_2 = tony, Y_1 = Y_2 = phil
mother(tony,phil)

FAILS

ancestor(X_1,Y_1) :- parent(X_1,Z_1),
ancestor(Z_1,Y_1).

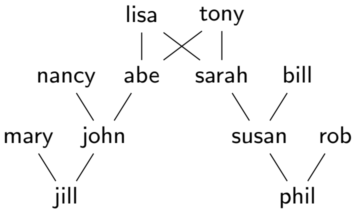


Search Example

ancestor(tony,phil)

⋮

ancestor(X_1,Y_1) :- parent(X_1,Z_1), ancestor(Z_1,Y_1).



Search Example

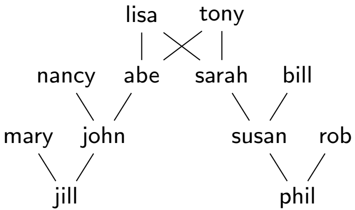
ancestor(tony,phil)

⋮

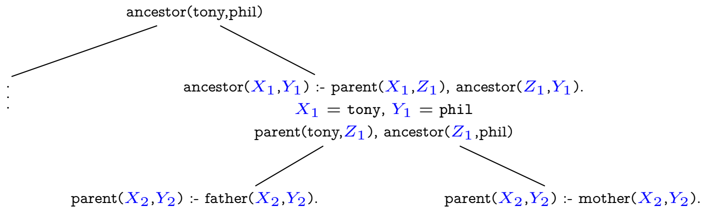
ancestor(X_1,Y_1) :- parent(X_1,Z_1), ancestor(Z_1,Y_1).

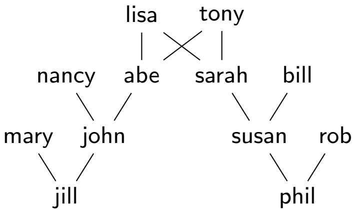
X_1 = tony, Y_1 = phil

parent(tony,Z_1), ancestor(Z_1,phil)



Search Example





Search Example

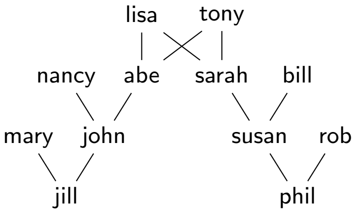
ancestor(tony,phil)

⋮

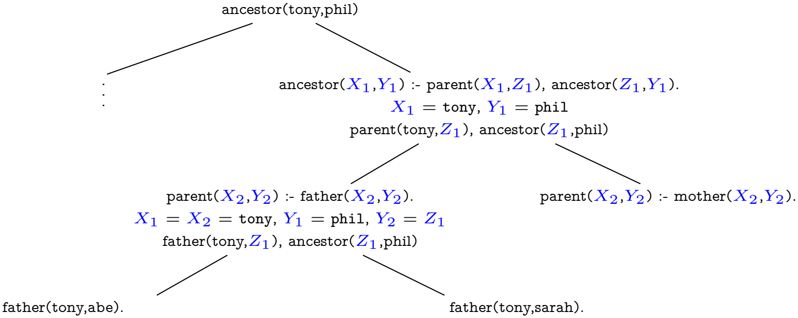
ancestor(X_1,Y_1) :- parent(X_1,Z_1), ancestor(Z_1,Y_1).
X_1 = tony, Y_1 = phil
parent(tony,Z_1), ancestor(Z_1,phil)

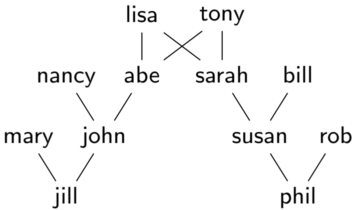
parent(X_2,Y_2) :- father(X_2,Y_2).
$X_1 = X_2$ = tony, Y_1 = phil, $Y_2 = Z_1$
father(tony,Z_1), ancestor(Z_1,phil)

parent(X_2,Y_2) :- mother(X_2,Y_2).

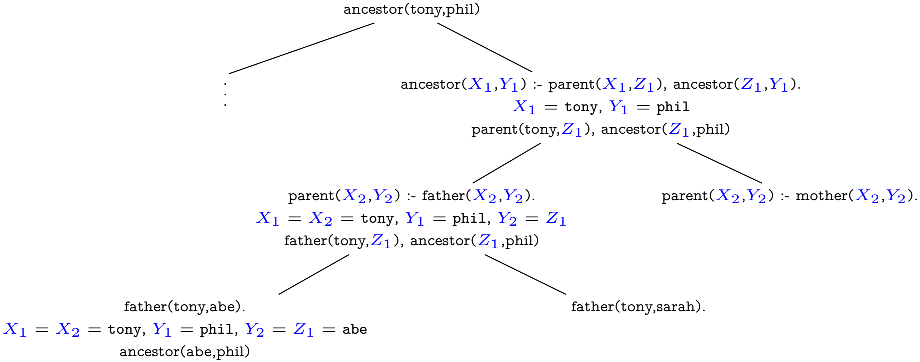


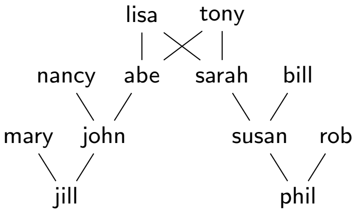
Search Example





Search Example





Search Example

ancestor(tony,phil)

⋮

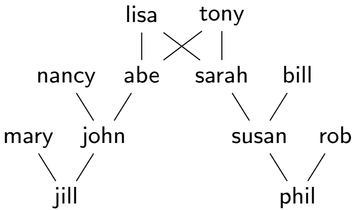
ancestor(X_1,Y_1) :- parent(X_1,Z_1), ancestor(Z_1,Y_1).
X_1 = tony, Y_1 = phil
parent(tony,Z_1), ancestor(Z_1,phil)

parent(X_2,Y_2) :- father(X_2,Y_2).
X_1 = X_2 = tony, Y_1 = phil, Y_2 = Z_1
father(tony,Z_1), ancestor(Z_1,phil)

parent(X_2,Y_2) :- mother(X_2,Y_2).

father(tony,abe).
X_1 = X_2 = tony, Y_1 = phil, Y_2 = Z_1 = abe
ancestor(abe,phil)
. . .
EVENTUALLY FAILS

father(tony,sarah).



Search Example

ancestor(tony,phil)

⋮

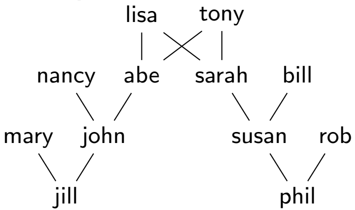
ancestor(X_1,Y_1) :- parent(X_1,Z_1), ancestor(Z_1,Y_1).
X_1 = tony, Y_1 = phil
parent(tony,Z_1), ancestor(Z_1,phil)

parent(X_2,Y_2) :- father(X_2,Y_2).
X_1 = X_2 = tony, Y_1 = phil, Y_2 = Z_1
father(tony,Z_1), ancestor(Z_1,phil)

parent(X_2,Y_2) :- mother(X_2,Y_2).

father(tony,abe).
X_1 = X_2 = tony, Y_1 = phil, Y_2 = Z_1 = abe
ancestor(abe,phil)
...
EVENTUALLY FAILS

father(tony,sarah).
X_1 = X_2 = tony, Y_1 = phil, Y_2 = Z_1 = sarah
ancestor(sarah,phil)

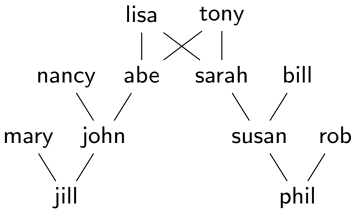


Search Example

father(tony,sarah).

$X_1 = X_2 =$ tony, $Y_1 =$ phil, $Y_2 = Z_1 =$ sarah

ancestor(sarah,phil)



Search Example

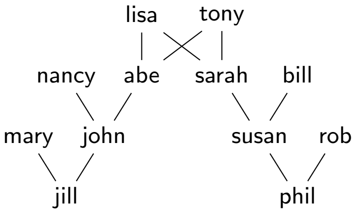
father(tony,sarah).

$X_1 = X_2 = \texttt{tony},\ Y_1 = \texttt{phil},\ Y_2 = Z_1 = \texttt{sarah}$

ancestor(sarah,phil)

⋮

ancestor(susan,phil)



Search Example

father(tony,sarah).

$X_1 = X_2 =$ tony, $Y_1 =$ phil, $Y_2 = Z_1 =$ sarah

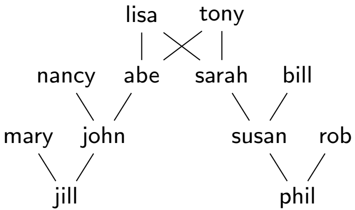
ancestor(sarah,phil)

⋮

ancestor(susan,phil)

ancestor(X_3,Y_3) :- parent(X_3,Y_3).

ancestor(X_3,Y_3) :- parent(X_3,Z_3),
ancestor(Z_3,Y_3).



Search Example

father(tony,sarah).

$X_1 = X_2 =$ `tony`, $Y_1 =$ `phil`, $Y_2 = Z_1 =$ `sarah`

ancestor(sarah,phil)

⋮

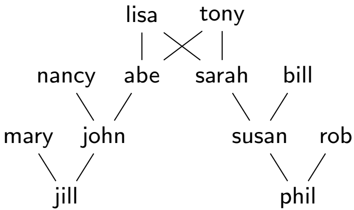
ancestor(susan,phil)

ancestor(X_3,Y_3) :- parent(X_3,Y_3).

$X_3 =$ `susan`, $Y_3 =$ `phil`

parent(susan,phil)

ancestor(X_3,Y_3) :- parent(X_3,Z_3),
ancestor(Z_3,Y_3).



Search Example

father(tony,sarah).

$X_1 = X_2 =$ tony, $Y_1 =$ phil, $Y_2 = Z_1 =$ sarah

ancestor(sarah,phil)

⋮

ancestor(susan,phil)

ancestor(X_3,Y_3) :- parent(X_3,Y_3).

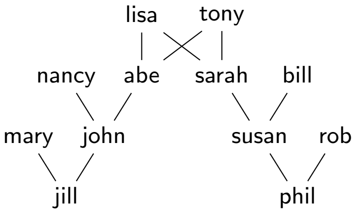
$X_3 =$ susan, $Y_3 =$ phil

parent(susan,phil)

father(susan,phil).

mother(susan,phil).

ancestor(X_3,Y_3) :- parent(X_3,Z_3),
ancestor(Z_3,Y_3).



Search Example

father(tony,sarah).

$X_1 = X_2 =$ tony, $Y_1 =$ phil, $Y_2 = Z_1 =$ sarah

ancestor(sarah,phil)

⋮

ancestor(susan,phil)

ancestor(X_3,Y_3) :- parent(X_3,Y_3).

$X_3 =$ susan, $Y_3 =$ phil

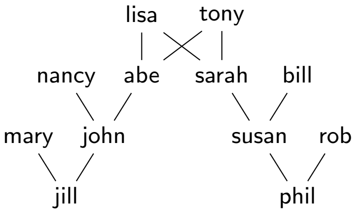
parent(susan,phil)

father(susan,phil).

FAILS

mother(susan,phil).

ancestor(X_3,Y_3) :- parent(X_3,Z_3),
ancestor(Z_3,Y_3).



Search Example

father(tony,sarah).

$X_1 = X_2 =$ tony, $Y_1 =$ phil, $Y_2 = Z_1 =$ sarah

ancestor(sarah,phil)

⋮

ancestor(susan,phil)

ancestor(X_3,Y_3) :- parent(X_3,Y_3).

$X_3 =$ susan, $Y_3 =$ phil

parent(susan,phil)

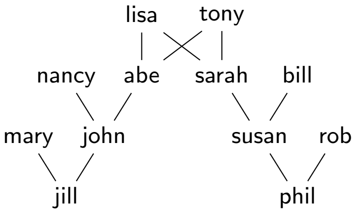
father(susan,phil).

FAILS

mother(susan,phil).

SUCCEEDS

ancestor(X_3,Y_3) :- parent(X_3,Z_3),
ancestor(Z_3,Y_3).



Important Points

- Order matters!
  - order of facts/rules in file
  - order of predicates on rhs of each rule
  - order *only affects termination* (as long as you stick to a certain language subset...), but does not change answers
- Tips for good ordering:
  - put facts before rules (base cases first)
  - put "easy" predicates before "harder" ones

Impact of Reordering

Our definition of ancestor:

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

Q1: What would happen if we reversed the rule order?

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

Q2: What if we reversed the conjunct order within the last rule?

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

Q3: What if we did both?

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

Equality Predicates

- "=" means "unifiable"
  - attempts a unification (possibly adding new variable bindings)
  - Example #1: `f(X,a)=f(b,Y).` *(succeeds with $X = \mathtt{b}$, $Y = \mathtt{a}$)*
  - Example #2: `X=a, X=b.` *(fails)*
  - Example #3: `X=a, a=X.` *(succeeds with $X = \mathtt{a}$)*
- "==" means "physically equal"
  - tests existing bindings (no new unification!)
  - Example #1: `a==b` *(fails)*
  - Example #2: `X==Z` *(fails)*
  - Example #3: `X=Z, X==Z` *(succeeds)*
  - Example #4: `X==a` *(fails)*
  - Example #5: `X=a, X==a` *(succeeds)*
- "\== is negation of "=="
  - `sibling(X,Y) :- parent(Z,X), parent(Z,Y), X \== Y.`

Inequalities

- Numerical inequalities
  - `X < Y`, `X > Y`, `X =< Y`, `X >= Y`
  - succeed only when both `X` and `Y` are *already bound to integers*
  - no unification occurs
  - no arithmetic expressions permitted!
    - Example: `X+3 < X+4` *(syntax error)*
- Non-numerical comparisons
  - `X @< Y`, `X @> Y`, `X @=< Y`, `X @>= Y`
  - compare arbitrary atoms according to a "standard" ordering
  - Example: `bar @< foo` *(succeeds)*
  - `X` and `Y` must be bound

Choice Operators

- Semicolon is disjunction
  - Example: `parent(X,Y) :- (father(X,Y); mother(X,Y)), X \== Y.`
  - Always replacable with multiple rules, so never necessary
  - But it can sometimes be very convenient.
- Ternary operator: P_1 -> P_2 ; P_3
  - If P_1 succeeds, do P_2 (and discard P_3); otherwise do P_3 (and discard P_2)
  - Not quite the same as logical implication (think of it as "if P_1 is provable..." rather than "if P_1 is true...")
  - Diverges when P_1 diverges
  - Always replacable with multiple rules (like disjunction)
- Underscore is a wildcard

```
isparent(X) :- parent(X,_).
```

  - If you write a variable on a rule's lhs that's never used on its rhs, you'll get a warning. Use underscore instead.
  - Warnings help programmer identify typos (e.g., mistyped variable names).

Negation

- "\+ P" succeeds when predicate P terminates with failure
  - NOT the same as logical negation!
  - think of it more like "P is disprovable"
  - loops when P loops
  - can exacerbate order-sensitivity issues
  - avoid spurious uses, but sometimes needed

Arithmetic

- "is" keyword
  - Syntax: X is 3+5
  - single variable on left
  - arithmetic *expression* on right
  - no unbound variables permitted on right!
- Examples:
  - X=5, X is 4+2 *(fails)*
  - X is Y+3 *(aborts with error if Y unbound)*
  - X=5, Y is X+3 *(succeeds with $Y = 8$)*
- Equality *does not* solve arithmetic
  - X = 3+5 *(binds X to the literal* **structure** *"3+5")*
- The "is" keyword *is not* an assignment operation
  - X is X+1 *(always fails)*
  - X=X+1 *(always fails)*

Lists

- Syntax
  - [] is the empty list
  - [H|T] is a list with head H and tail T
    - Recall: list tail is *list* of all elements except head
    - tail can be empty
  - [X,Y|Z] is a list with first two elements X and Y, and remaining elements Z
- **Exercise:** Implement a predicate sum(L,S) that succeeds with S equal to the sum of numbers in list L.

Lists

- Syntax
  - [] is the empty list
  - [H|T] is a list with head H and tail T
    - Recall: list tail is *list* of all elements except head
    - tail can be empty
  - [X,Y|Z] is a list with first two elements X and Y, and remaining elements Z
- **Exercise:** Implement a predicate sum(L,S) that succeeds with S equal to the sum of numbers in list L.

```
sum([],0).
```

Lists

- Syntax
  - [] is the empty list
  - [H|T] is a list with head H and tail T
    - Recall: list tail is *list* of all elements except head
    - tail can be empty
  - [X,Y|Z] is a list with first two elements X and Y, and remaining elements Z
- **Exercise:** Implement a predicate sum(L,S) that succeeds with S equal to the sum of numbers in list L.

```
sum([],0).
sum([H|T],S) :- sum(T,S1), S is H+S1.
```

More List Examples

Exercise: Implement a predicate `append(L1,L2,L3)` that succeeds with `L3` equal to list `L1` appended by list `L2`.

More List Examples

Exercise: Implement a predicate `append(L1,L2,L3)` that succeeds with `L3` equal to list `L1` appended by list `L2`.

```
append([],L,L).
```

More List Examples

Exercise: Implement a predicate `append(L1,L2,L3)` that succeeds with `L3` equal to list `L1` appended by list `L2`.

```
append([],L,L).
append([H1|T1],L2,[H1|T3]) :- append(T1,L2,T3).
```

More List Examples

Exercise: Implement a predicate `append(L1,L2,L3)` that succeeds with `L3` equal to list `L1` appended by list `L2`.

```
append([],L,L).
append([H1|T1],L2,[H1|T3]) :- append(T1,L2,T3).
```

Exercise: Implement a predicate `pick(X,L1,L2)` that succeeds when `X` is a member of list `L1`, and `L2` is list `L1` without the first `X`.

More List Examples

Exercise: Implement a predicate append(L1,L2,L3) that succeeds with L3 equal to list L1 appended by list L2.

```
append([],L,L).
append([H1|T1],L2,[H1|T3]) :- append(T1,L2,T3).
```

Exercise: Implement a predicate pick(X,L1,L2) that succeeds when X is a member of list L1, and L2 is list L1 without the first X.

```
pick(X,[X|T],T).
```

More List Examples

Exercise: Implement a predicate append(L1,L2,L3) that succeeds with L3 equal to list L1 appended by list L2.

```
append([],L,L).
append([H1|T1],L2,[H1|T3]) :- append(T1,L2,T3).
```

Exercise: Implement a predicate pick(X,L1,L2) that succeeds when X is a member of list L1, and L2 is list L1 without the first X.

```
pick(X,[X|T],T).
pick(X,[Y|T1],[Y|T2]) :- X \== Y, pick(X,T1,T2).
```

Logical Arithmetic

- Encode natural numbers as structures:
  - zero is 0
  - one is `s(0)`
  - two is `s(s(0))`
- **Exercise:** Implement a predicate `num(N)` that succeeds when `N` is a valid logical arithmetic encoding.

```
num(0).
num(s(N)) :- num(N).
```

- **Exercise:** Implement a predicate `lplus(X,Y,Z)` that succeeds with `Z` equal to the logical numeral that encodes the sum of logical numerals `X` and `Y`.

Logical Arithmetic

- Encode natural numbers as structures:
  - zero is 0
  - one is s(0)
  - two is s(s(0))
- **Exercise:** Implement a predicate num(N) that succeeds when N is a valid logical arithmetic encoding.

```
num(0).
num(s(N)) :- num(N).
```

- **Exercise:** Implement a predicate lplus(X,Y,Z) that succeeds with Z equal to the logical numeral that encodes the sum of logical numerals X and Y.

```
lplus(0,Y,Y).
```

Logical Arithmetic

- Encode natural numbers as structures:
  - zero is 0
  - one is s(0)
  - two is s(s(0))
- **Exercise:** Implement a predicate num(N) that succeeds when N is a valid logical arithmetic encoding.

```
num(0).
num(s(N)) :- num(N).
```

- **Exercise:** Implement a predicate lplus(X,Y,Z) that succeeds with Z equal to the logical numeral that encodes the sum of logical numerals X and Y.

```
lplus(0,Y,Y).
lplus(s(X),Y,s(Z)) :- lplus(X,Y,Z).
```

Logical Arithmetic

Exercise: Implement a predicate `lminus(X,Y,Z)` that succeeds with `Z` equal to the logical numeral that encodes the difference between logical numerals `X` and `Y`.

Logical Arithmetic

Exercise: Implement a predicate lminus(X,Y,Z) that succeeds with Z equal to the logical numeral that encodes the difference between logical numerals X and Y.

```
lminus(X,Y,Z) :- lplus(Y,Z,X).
```

Logical Arithmetic

Exercise: Implement a predicate `lminus(X,Y,Z)` that succeeds with `Z` equal to the logical numeral that encodes the difference between logical numerals `X` and `Y`.

```
lminus(X,Y,Z) :- lplus(Y,Z,X).
```

Exercise: Implement a predicate `ltimes(X,Y,Z)` that succeeds with `Z` equal to the logical numeral that encodes the product of logical numerals `X` and `Y`.

Logical Arithmetic

Exercise: Implement a predicate lminus(X,Y,Z) that succeeds with Z equal to the logical numeral that encodes the difference between logical numerals X and Y.

```
lminus(X,Y,Z) :- lplus(Y,Z,X).
```

Exercise: Implement a predicate ltimes(X,Y,Z) that succeeds with Z equal to the logical numeral that encodes the product of logical numerals X and Y.

```
ltimes(0,Y,0).
```

Logical Arithmetic

Exercise: Implement a predicate lminus(X,Y,Z) that succeeds with Z equal to the logical numeral that encodes the difference between logical numerals X and Y.

```
lminus(X,Y,Z) :- lplus(Y,Z,X).
```

Exercise: Implement a predicate ltimes(X,Y,Z) that succeeds with Z equal to the logical numeral that encodes the product of logical numerals X and Y.

```
ltimes(0,Y,0).
ltimes(s(X),Y,Z) :- ltimes(X,Y,XY), lplus(XY,Y,Z).
```

Cryptarithmetic Puzzles

$$\begin{array}{rr} & A\,M \\ + & P\,M \\ \hline & D\,A\,Y \end{array}$$

Exercise: Use Prolog to find a mapping from letters to digits such that:

- no leftmost digit is a zero
- no two letters are assigned the same digit

Specifically, `solve([A,M,P,D,Y])` should succeed with a list of digits for the corresponding letters satisfying all above constraints.

Cryptarithmetic Solution

$$\begin{array}{rr} & A\ M \\ + & P\ M \\ \hline & D\ A\ Y \end{array}$$

```
solve([A,M,P,D,Y]) :-
```

Cryptarithmetic Solution

$$\begin{array}{r} A\ M \\ +\quad P\ M \\ \hline D\ A\ Y \end{array}$$

```
solve([A,M,P,D,Y]) :-
    pick(M,[0,1,2,3,4,5,6,7,8,9],L1),
```

Cryptarithmetic Solution

$$\begin{array}{r} A\ M \\ +\quad P\ M \\ \hline D\ A\ Y \end{array}$$

```
solve([A,M,P,D,Y]) :-
   pick(M,[0,1,2,3,4,5,6,7,8,9],L1),
   Y is (M+M) mod 10,
   C1 is (M+M) // 10,
```

Cryptarithmetic Solution

$$\begin{array}{rr} & A\ M \\ + & P\ M \\ \hline & D\ A\ Y \end{array}$$

```
solve([A,M,P,D,Y]) :-
   pick(M,[0,1,2,3,4,5,6,7,8,9],L1),
   Y is (M+M) mod 10,
   C1 is (M+M) // 10,
   pick(Y,L1,L2),
```

Cryptarithmetic Solution

$$\begin{array}{r} A\ M \\ +\quad P\ M \\ \hline D\ A\ Y \end{array}$$

```
solve([A,M,P,D,Y]) :-
    pick(M,[0,1,2,3,4,5,6,7,8,9],L1),
    Y is (M+M) mod 10,
    C1 is (M+M) // 10,
    pick(Y,L1,L2),
    pick(A,L2,L3), A \== 0,
```

Cryptarithmetic Solution

$$\begin{array}{r} A\ M \\ +\quad P\ M \\ \hline D\ A\ Y \end{array}$$

```
solve([A,M,P,D,Y]) :-
  pick(M,[0,1,2,3,4,5,6,7,8,9],L1),
  Y is (M+M) mod 10,
  C1 is (M+M) // 10,
  pick(Y,L1,L2),
  pick(A,L2,L3), A \== 0,
  pick(P,L3,L4), P \== 0,
```

Cryptarithmetic Solution

$$\begin{array}{rccc} & & A & M \\ + & & P & M \\ \hline & D & A & Y \end{array}$$

```
solve([A,M,P,D,Y]) :-
    pick(M,[0,1,2,3,4,5,6,7,8,9],L1),
    Y is (M+M) mod 10,
    C1 is (M+M) // 10,
    pick(Y,L1,L2),
    pick(A,L2,L3), A \== 0,
    pick(P,L3,L4), P \== 0,
    A is (A+P+C1) mod 10,
```

Cryptarithmetic Solution

$$\begin{array}{rr} & A\ M \\ + & P\ M \\ \hline & D\ A\ Y \end{array}$$

```
solve([A,M,P,D,Y]) :-
   pick(M,[0,1,2,3,4,5,6,7,8,9],L1),
   Y is (M+M) mod 10,
   C1 is (M+M) // 10,
   pick(Y,L1,L2),
   pick(A,L2,L3), A \== 0,
   pick(P,L3,L4), P \== 0,
   A is (A+P+C1) mod 10,
   D is (A+P+C1) // 10, D \== 0,
```

Cryptarithmetic Solution

$$\begin{array}{rr} & A\ M \\ + & P\ M \\ \hline & D\ A\ Y \end{array}$$

```
solve([A,M,P,D,Y]) :-
   pick(M,[0,1,2,3,4,5,6,7,8,9],L1),
   Y is (M+M) mod 10,
   C1 is (M+M) // 10,
   pick(Y,L1,L2),
   pick(A,L2,L3), A \== 0,
   pick(P,L3,L4), P \== 0,
   A is (A+P+C1) mod 10,
   D is (A+P+C1) // 10, D \== 0,
   pick(D,L4,_).
```

Cut Operator

- Predicate "!" always succeeds and cannot be backtracked over.
  - prunes the search tree when it appears
  - can make code significantly more difficult to understand and debug
- Example: List membership

```
mem(X,[X|_]) :- !.
mem(X,[_|T]) :- mem(X,T).
```

  - How does this differ from pick(X,L)?
  - What happens if we delete the cut (and the whole first rule)?
- Green vs. red cuts
  - **Green cut:** a cut that doesn't change any success/failure if removed (only improves efficiency)
  - **Red cut:** a non-green cut
  - Many logic programmers consider red cuts to be poor programming, and consider green cuts to be at best a necessary evil.

Strategic Cuts

In this class:

- I won't require you to know anything about cuts (all problems solvable without them).
- You should avoid using cuts until you are a proficient logic programmer (comfortable with most other aspects of the language).
- If you use cuts, stick to green cuts only. (If you aren't sure, you shouldn't be using cuts!)
- Read more about them online (cuts surround much theory, history, and opinion of logic programming!).

Final Remarks

- Prolog has no function calls!
  - `f(...)` as an argument to a predicate is a structure (not evaluated).
  - `f(...)` as a predicate sometimes feels like a function, but it's not. It's a search.
  - Easy to get confused if you're an imperative or functional programmer.
- Inputs vs. outputs
  - There are no functions, so there are no return values.
  - Many (most?) predicates are intended to work with certain arguments being "inputs" and others being "outputs" (but they can be in any order).
  - If this is desired, I will try to be clear about it: `mypredicate(X,Y,Out)`.
  - Really great solutions work correctly with any/all combinations of arguments being bound and unbound!
- Ordering
  - Success does not stop the program (e.g., user may press semicolon, caller may backtrack, etc.)!
  - Correct code must never later succeed on wrong answers.
- Grading and partial credit
  - Don't write me a Java program. I'm evaluating whether you can think like a logic programmer.
  - If you rely upon predicates we've defined in class or on homework, you must define them again (because their exact definitions often affect whether your code works).
  - Good logic programs are usually short (relative to imperative and even functional code), elegant, and clear.