FP vs. LP

- **Functional Programming**
  - centered around first-class functions
  - strong, parametric polymorphic type systems
  - single-assignment
  - operational semantics based on $\lambda$-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 $f : A \rightarrow B$ as sets of pairs $A \times B$.
  - Relations (as defined above) are also sets of pairs.
  - Function $f$ encodes relation $\{(x, f(x)) \mid x \in f^{-}\}$
  - Unlike functions, relations can map the same domain element to multiple different range elements.
Relational Programming

Three ways to define a function/relation:

- Imperatively:

  \[
  \text{factorial}(x) := \{z := 1; \text{ for } i := 1 \text{ to } x \text{ do } z := z * i; \text{ return } z\}
  \]

- Functionally:

  \[
  \text{factorial}(x) := (\text{if } x \leq 0 \text{ then } 1 \text{ else } x \times \text{factorial}(x - 1))
  \]

- Relationally:

  \[
  \text{factorial}(0, 1).
  \]
  \[
  \text{factorial}(x, y) \text{ if } \text{factorial}(x - 1, y/x).
  \]

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 optimize 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:
  - \( p(t_1, \ldots, t_n). \)
  - \( p(t_1, \ldots, t_n) :- p_1(t_1, \ldots, t_i), p_2(t_1, \ldots, t_j), \ldots, p_m(t_1, \ldots, t_k). \)
- 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 (captive identifiers)
    - Examples: \( X, \) Foo
  - structures (tree-shaped data structures)
    - Examples: \( \text{foo}(3,12), \text{foo}(\text{foo}(13),\text{foo}(16,12)) \)
    - 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).
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).
```
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).
Q1: How might we decide parent relations?

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

Q2: Grandparent?

gp(X, 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).
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?

ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z), ancestor(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) :-
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) :-
Q1: How might we decide parent relations?
    \[ \text{parent}(X,Y) :- \text{father}(X,Y). \]
    \[ \text{parent}(X,Y) :- \text{mother}(X,Y). \]

Q2: Grandparent?
    \[ \text{gp}(X,Y) :- \text{parent}(X,Z), \text{parent}(Z,Y). \]

Q3: Great-grandparent?
    \[ \text{ggp}(X,Y) :- \text{gp}(X,Z), \text{parent}(Z,Y). \]

Q4: Ancestor?
    \[ \text{ancestor}(X,Y) :- \text{parent}(X,Y). \]
    \[ \text{ancestor}(X,Y) :- \text{parent}(X,Z), \text{ancestor}(Z,Y). \]
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
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
ancestor(tony, phil)

parent(tony, phil)
FAILS

father(tony, phil)
FAILS

mother(tony, phil)
FAILS

ancestor(X1, Y1)
:- parent(X1, Z1), ancestor(Z1, Y1).

X1 = tony, Y1 = phil

mary jill
|     |     
mother |      
it is a mother relationship

John
|     |
father |      
it is a father relationship

Lisa
|     |
father |      
it is a father relationship

Sarah
|     |
father |      
it is a father relationship

Bill
|     |
father |      
it is a father relationship

Rob
|     |
father |      
it is a father relationship

Phil
|     |
father |      
it is a father relationship

Search Example

ancestor(tony, phil)

ancestor(X₁, Y₁) :- parent(X₁, Y₁).

ancestor(X₁, Y₁) :- parent(X₁, Z₁), ancestor(Z₁, Y₁).

ancestor(tony, phil)

parent(tony, phil)

FAILS

parent(X₂, Y₂) :- father(X₂, Y₂).

X₁ = X₂ = tony, Y₁ = Y₂ = phil

father(tony, phil)

FAILS

parent(X₂, Y₂) :- mother(X₂, Y₂).

X₁ = X₂ = tony, Y₁ = Y₂ = phil

mother(tony, phil)

FAILS
Search Example

ancestor(tony, phil)

ancestor(X₁, Y₁) :- parent(X₁, Y₁).
X₁ = tony, Y₁ = phil
parent(tony, phil)

ancestor(X₁, Y₁) :- parent(X₁, Z₁),
ancestor(Z₁, Y₁).

ancestor(X₁, Y₁) :- parent(X₁, Y₁).

ancestor(tony, phil)
Search Example

\[
\text{ancestor}(X_1, Y_1) :- \text{parent}(X_1, Y_1).
\]

\[
X_1 = \text{tony}, Y_1 = \text{phil}
\]

\[
\text{parent}(\text{tony}, \text{phil})
\]

\[
\text{parent}(X_2, Y_2) :- \text{father}(X_2, Y_2).
\]

\[
\text{parent}(X_2, Y_2) :- \text{mother}(X_2, Y_2).
\]

\[
\text{ancestor}(X_1, Y_1) :- \text{parent}(X_1, Z_1), \text{ancestor}(Z_1, Y_1).
\]
Search Example

ancestor(tONY,phil)

ancestor(X₁,Y₁) :- parent(X₁,Y₁).
  X₁ = tONY, Y₁ = phil

parent(tONY,phil)

parent(X₂,Y₂) :- father(X₂,Y₂).
  X₁ = X₂ = tONY, Y₁ = Y₂ = phil

father(tONY,phil)

ancestor(X₁,Y₁) :- parent(X₁,Z₁),
  ancestor(Z₁,Y₁).

ancestor(X₁,Y₁) :- parent(X₁,Z₁),
  ancestor(Z₁,Y₁).

failS

parent(X₂,Y₂) :- mother(X₂,Y₂).

parent(X₂,Y₂) :- mother(X₂,Y₂).

failS

Search Example

ancestor(tony,phil)

ancestor(X₁,Y₁) :- parent(X₁,Y₁).
  X₁ = tony, Y₁ = phil

parent(tony,phil)

parent(X₂,Y₂) :- father(X₂,Y₂).
  X₁ = X₂ = tony, Y₁ = Y₂ = phil

father(tony,phil)

ancestor(X₁,Y₁) :- parent(X₁,Z₁),
  ancestor(Z₁,Y₁).

ancestor(X₁,Y₁) :- parent(X₁,Z₁),
  ancestor(Z₁,Y₁).

failS

parent(X₂,Y₂) :- mother(X₂,Y₂).

parent(X₂,Y₂) :- mother(X₂,Y₂).

failS

ancestor(tony,phil)
Search Example

ancestor(tony, phil)

ancestor(X₁, Y₁) :- parent(X₁, Y₁).
  X₁ = tony, Y₁ = phil
parent(tony, phil)

parent(X₂, Y₂) :- father(X₂, Y₂).
X₁ = X₂ = tony, Y₁ = Y₂ = phil
father(tony, phil)

FAILS

ancestor(X₁, Y₁) :- parent(X₁, Z₁), ancestor(Z₁, Y₁).

ancestor(X₁, Y₁) :- parent(X₁, Z₁), ancestor(Z₁, Y₁).

parent(X₂, Y₂) :- mother(X₂, Y₂).
parent(X₂, Y₂) :- mother(X₂, Y₂).

parent(X₂, Y₂) :- mother(X₂, Y₂).

ancestor(tony, phil)
Search Example

ancestor(tony, phil)

ancestor(X₁, Y₁) :- parent(X₁, Y₁).
X₁ = tony, Y₁ = phil
parent(tony, phil)

parent(X₂, Y₂) :- father(X₂, Y₂).
X₁ = X₂ = tony, Y₁ = Y₂ = phil
father(tony, phil)

FAILS

ancestor(X₁, Y₁) :- parent(X₁, Z₁),
ancestor(Z₁, Y₁).

ancestor(X₁, Y₁) :- parent(X₁, Z₁),
ancestor(Z₁, Y₁).

ancestor(X₁, Y₁) :- parent(X₁, Z₁),
ancestor(Z₁, Y₁).

ancestor(X₁, Y₁) :- parent(X₁, Z₁),
ancestor(Z₁, Y₁).

father(tony, phil)

mother(tony, phil)
Search Example

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

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

father(tony, phil) :- mother(tony, phil).
   \[ X_1 = X_2 = tony, Y_1 = Y_2 = phil \]

parent(\(X_1, Y_1\)) :- parent(\(X_1, Y_1\)).
   \[ X_1 = tony, Y_1 = phil \]

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

FAILS

FAILS
Search Example

\[\text{ancestor}(\text{tony}, \text{phil})\]

\[\begin{align*}
\text{ancestor}(X_1, Y_1) & \leftarrow \text{parent}(X_1, Z_1), \text{ancestor}(Z_1, Y_1). \\
\end{align*}\]
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

```
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).

parent(X_2, Y_2) :~ mother(X_2, Y_2).
```
Search Example

ancestor(tony, phil)

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

\(X_1 = \text{tony}, Y_1 = \text{phil}\)

parent(tony, \(Z_1\)), ancestor(\(Z_1, \text{phil}\))

parent(\(X_2, Y_2\)) :- father(\(X_2, Y_2\)).

\(X_1 = X_2 = \text{tony}, Y_1 = \text{phil}, Y_2 = Z_1\)

father(tony, \(Z_1\)), ancestor(\(Z_1, \text{phil}\))

parent(\(X_2, Y_2\)) :- mother(\(X_2, Y_2\)).
Search Example

```
ancestor(tony,phil)

ancestor(X1,Y1) :- parent(X1,Z1), ancestor(Z1,Y1).
X1 = tony, Y1 = phil
parent(tony,Z1), ancestor(Z1,phil)

parent(X2,Y2) :- father(X2,Y2).
X1 = X2 = tony, Y1 = phil, Y2 = Z1
father(tony,Z1), ancestor(Z1,phil)

father(tony,abe).
father(tony,sarah).
```

```
EVENTUALLY FAILS
father(tony,sarah).
X1 = X2 = tony, Y1 = phil, Y2 = Z1 = sarah
ancestor(sarah,phil)
```

```
parent(X2,Y2) :- mother(X2,Y2).
lisa tony
nancy abe sarah bill
mary john susan rob
jill
phil
```
ancestor(tony, phil)
**Search Example**

```
ancestor(tony, phil)

\[\vdash ancestor(X_1, Y_1) \iff parent(X_1, Z_1), ancestor(Z_1, Y_1)\]

\[X_1 = tony, Y_1 = phil\]

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

\[\vdash parent(X_2, Y_2) \iff father(X_2, Y_2)\]

\[X_1 = X_2 = tony, Y_1 = phil, Y_2 = Z_1\]

father(tony, Z_1), ancestor(Z_1, phil)

father(tony, abe).

\[X_1 = X_2 = tony, Y_1 = phil, Y_2 = Z_1 = abe\]

ancestor(abe, phil)

\[\vdash\]

\[\text{EVENTUALLY FAILS}\]
```
Search Example

ancestor(tony, phil)

parent(tony, Z1), ancestor(Z1, phil)

parent(X1, Y1) :- parent(X1, Z1), ancestor(Z1, Y1).
X1 = tony, Y1 = phil

parent(tony, Z1), ancestor(Z1, phil)

father(tony, Z1), ancestor(Z1, phil)

father(tony, abe).
X1 = X2 = tony, Y1 = phil, Y2 = Z1 = abe
ancestor(abe, phil)

EVENTUALLY FAILS
father(tony,sarah).

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

ancestor(sarah,phil)

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

father(susan,phil).

mother(susan,phil).

SUCCEEDS

FAILS
father(tony,sarah).

\[ X_1 = X_2 = \text{tony}, \ Y_1 = \text{phil}, \ Y_2 = Z_1 = \text{sarah} \]

ancestor(sarah,phil)

\[ \vdots \]

ancestor(susan,phil)

fail

mother(susan,phil).

succeeds

ancestor(\(X_3\), \(Y_3\)) :- parent(\(X_3\), \(Z_3\)), ancestor(\(Z_3\), \(Y_3\)).
Search Example

father(tony,sarah).

\[ X_1 = X_2 = \text{tony}, \quad Y_1 = \text{phil}, \quad Y_2 = Z_1 = \text{sarah} \]

ancestor(sarah,phil)

\[ \vdots \]

ancestor(susan,phil)

\[
\text{ancestor}(X_3, Y_3) :- \text{parent}(X_3, Y_3).
\]

\[
\text{ancestor}(X_3, Y_3) :- \text{parent}(X_3, Z_3), \quad \text{ancestor}(Z_3, Y_3).
\]
father(tony,sarah).

\[ X_1 = X_2 = \text{tony}, \ Y_1 = \text{phil}, \ Y_2 = Z_1 = \text{sarah} \]

ancestor(sarah,phil)

\[ \vdots \]

ancestor(susan,phil)

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

\[ X_3 = \text{susan}, \ Y_3 = \text{phil} \]

parent(susan,phil)

ancestor(\(X_3, Y_3\)) :- parent(\(X_3, Z_3\)), ancestor(\(Z_3, Y_3\)).
father(tony,sarah).

\[ X_1 = X_2 = \text{tony}, \quad Y_1 = \text{phil}, \quad Y_2 = Z_1 = \text{sarah} \]

ancestor(sarah,phil)

\[ \vdots \]

ancestor(susan,phil)

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

\[ X_3 = \text{susan}, \quad Y_3 = \text{phil} \]

parent(susan,phil)

father(susan,phil).

mother(susan,phil).
father(tony,sarah).

\[ X_1 = X_2 = \text{tony}, \quad Y_1 = \text{phil}, \quad Y_2 = Z_1 = \text{sarah} \]

ancestor(sarah,phil)

\[ \vdots \]

ancestor(susan,phil)

\[ \text{ancestor}(X_3,Y_3) :\text{- parent}(X_3,Y_3). \]

\[ X_3 = \text{susan}, \quad Y_3 = \text{phil} \]

parent(susan,phil)

father(susan,phil).

FAILS

mother(susan,phil).

ancestor(X_3,Y_3) :\text{- parent}(X_3,Z_3),

ancestor(Z_3,Y_3).

\[ j\text{ill} \]

\[ m\text{ary} \quad j\text{ohn} \quad s\text{usan} \quad b\text{ill} \]

\[ l\text{isa} \quad t\text{ony} \quad s\text{arah} \quad b\text{ill} \]

\[ m\text{ary} \quad j\text{ohn} \quad s\text{usan} \quad r\text{ob} \]

\[ j\text{ill} \quad p\text{hil} \]
father(tony,sarah).

\[ X_1 = X_2 = tony, \ Y_1 = phil, \ Y_2 = Z_1 = sarah \]

ancestor(sarah,phil)

\[ \vdots \]

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

\[ \text{ancestor}(\text{susan}, \text{phil}) \]
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
Our definition of ancestor:

\[
\text{ancestor}(X,Y) :\ - \ \text{parent}(X,Y).
\]
\[
\text{ancestor}(X,Y) :\ - \ \text{parent}(X,Z), \ \text{ancestor}(Z,Y).
\]

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

\[
\text{ancestor}(X,Y) :\ - \ \text{parent}(X,Z), \ \text{ancestor}(Z,Y).
\]
\[
\text{ancestor}(X,Y) :\ - \ \text{parent}(X,Y).
\]

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

\[
\text{ancestor}(X,Y) :\ - \ \text{parent}(X,Y).
\]
\[
\text{ancestor}(X,Y) :\ - \ \text{ancestor}(Z,Y), \ \text{parent}(X,Z).
\]

**Q3:** What if we did both?

\[
\text{ancestor}(X,Y) :\ - \ \text{ancestor}(Z,Y), \ \text{parent}(X,Z).
\]
\[
\text{ancestor}(X,Y) :\ - \ \text{parent}(X,Y).
\]
Equality Predicates

- "\(=\)" means "unifiable"
  - attempts a unification (possibly adding new variable bindings)
  - Example #1: \(f(X,a)=f(b,Y)\). (succeeds with \(X = b, Y = a\))
  - Example #2: \(X=a, X=b\). (fails)
  - Example #3: \(X=a, a=X\). (succeeds with \(X = a\))

- "\(==\)" 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)

- "\(\neq\)\(\) is negation of "\(==\)"
  - sibling(X,Y) :- parent(Z,X), parent(Z,Y), X \(\neq\) Y.
Inequalities

- **Numerical inequalities**
  - $X < Y$, $X > Y$, $X \leq Y$, $X \geq Y$
  - 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**
  - $X \preceq Y$, $X \succeq Y$, $X \preceq Y$, $X \succeq Y$
  - compare arbitrary atoms according to a “standard” ordering
  - Example: `bar \preceq foo` (*succeeds*)
  - $X$ and $Y$ must be bound
Choice Operators

- **Semicolon is disjunction**
  - Example: parent(X,Y) :- (father(X,Y); mother(X,Y)), X \(\neq\) Y.
  - Always replacable with multiple rules, so never necessary
  - But it can sometimes be very convenient.

- **Ternary operator:** \(P_1 \rightarrow P_2 ; P_3\)
  - 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
Advanced Programming Languages

Features and Use Cases

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)
**Syntax**
- `[]` is the empty list
- `[H|T]` is a list with head `H` and tail `T`
  - Recall: list tail is a 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`. 

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

\[
\text{sum([],0).}
\]
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.

```prolog
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.

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).
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],L,L3) :- append(T1,L,L3).
```
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 `append(L1,L2,L3)` that succeeds with `L3` equal to list `L1` appended by list `L2`.

```prolog
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`.

```prolog
pick(X,[X|T],T).
pick(X,[Y|T1],[Y|T2]) :- X == Y, pick(X,T1,T2).
```
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).
```
Exercise: Implement a predicate \texttt{append(L1,L2,L3)} that succeeds with \texttt{L3} equal to list \texttt{L1} appended by list \texttt{L2}.

\begin{verbatim}
append([],L,L).
append([H1|T1],L2,[H1|T3]) :- append(T1,L2,T3).
\end{verbatim}

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

\begin{verbatim}
pick(X,[X|T],T).
pick(X,[Y|T1],[Y|T2]) :- X \leftarrow Y, pick(X,T1,T2).
\end{verbatim}
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.
  ```prolog
  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`.
  ```prolog
  lplus(0,Y,Y).
  lplus(s(X),Y,s(Z)) :- lplus(X,Y,Z).
  ```
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.
  
  
  ```prolog
  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`.

  ```prolog
  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.

  ```prolog
  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`.

  ```prolog
  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`.

```prolog
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`.

```prolog
ltimes(0,Y,0).
ltimes(s(X),Y,Z) :- ltimes(X,Y,XY), lplus(XY,Y,Z).
```
Exercise: Implement a predicate \texttt{lminus(X,Y,Z)} that succeeds with \( Z \) equal to the logical numeral that encodes the difference between logical numerals \( X \) and \( Y \).

\[
\text{lminus}(X,Y,Z) :- \text{lplus}(Y,Z,X).
\]
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`. 

\[
ltimes(0,Y,0).
\]

\[
ltimes(s(X),Y,Z) :- ltimes(X,Y,XY), lplus(XY,Y,Z).
\]
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`.

\[ lminus(X,Y,Z) :\text{-} lplus(Y,Z,X). \]

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`.

\[ ltimes(0,Y,0). \]
**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 \( Y \).

\[
\begin{align*}
ltimes(0,Y,0). \\
ltimes(s(X),Y,Z) & \ :- \ ltimes(X,Y,XY), \ lplus(XY,Y,Z).
\end{align*}
\]
Cryptarithmetic Puzzles

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

\[
\begin{array}{c}
A M \\
+ P M \\
\hline
D A Y
\end{array}
\]

\[\text{solve([A,M,P,D,Y]) :-}\]

A M

+ P M

\[\text{D A Y}\]
Cryptarithmetic Solution

\[
\begin{array}{c}
A \\ M \\
+ \\
\downarrow \\
D \\ A \\ Y
\end{array}
\]

\[\text{solve([A,M,P,D,Y])} \ :- \\
\text{pick(M,[0,1,2,3,4,5,6,7,8,9],L1),} \\
\text{pick(Y,L1,L2),} \\
\text{pick(A,L2,L3), A} \neq 0, \\
\text{pick(P,L3,L4), P} \neq 0, \\
\text{A} \neq (A+P+C1) \mod 10, \\
\text{D} \neq (A+P+C1) \div 10, \\
\text{solve([A,M,P,D,Y])} \
\]

Cryptarithmetic Solution

\[
\begin{array}{c}
A M \\
+ P M \\
\hline
D A Y
\end{array}
\]

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

Cryptarithmetic Solution

\[
\begin{array}{c}
A M \\
+ P M \\
\hline
D A Y
\end{array}
\]

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

\[
\begin{array}{c}
A M \\
+ \\
P M \\
\hline
D A Y
\end{array}
\]

```prolog
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,_).
```
Cryptarithmetic Solution

\[
\begin{array}{c}
A M \\
+ P M \\
\hline
D A Y
\end{array}
\]

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) \div 10,
pick(Y,L1,L2),
pick(A,L2,L3), A \leftarrow 0,
pick(P,L3,L4), P \leftarrow 0,
Cryptarithmetic Solution

\[
\begin{array}{c}
A M \\
+ P M \\
\hline
D A Y
\end{array}
\]

\[
solve([A,M,P,D,Y]) :-
\]
\[
pick(M,[0,1,2,3,4,5,6,7,8,9],L1),
\]
\[
Y \text{ is } (M+M) \mod 10,
\]
\[
C1 \text{ is } (M+M) \div 10,
\]
\[
pick(Y,L1,L2),
\]
\[
pick(A,L2,L3), A \\leq 0,
\]
\[
pick(P,L3,L4), P \\leq 0,
\]
\[
A \text{ is } (A+P+C1) \mod 10,
\]
Cryptarithmetic Solution

\[
\begin{array}{c}
A M \\
+ P M \\
\hline
D A Y
\end{array}
\]

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

Cryptarithmetic Solution

\[
\begin{array}{c}
A M \\
+ P M \\
\hline
D A Y
\end{array}
\]

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

  \[
  \text{mem}(X, [X \mid \_]) :- !. \\
  \text{mem}(X, [\_ \mid T]) :- \text{mem}(X, T).
  \]

- How does this differ from \text{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.
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.