

# Logic Programming

We thank Prof Basu and UMBC for some of the slides

November 6, 2016

# Introduction to Logic Programming

## Truly Declarative Paradigm

- User declares **what facts** are true.
- User states some queries.
- System determines **how to use the facts** that are true to answer the queries.

Primary difference between imperative programming and logic programming.

- Imperative: explicitly instruct the system how certain computation should be performed.
- Logic: instruct the system what can be used to perform some computation.

# What is Logic Programming

There are many (overlapping) perspectives on logic programming:

- ▶ A Very High Level Programming Language
- ▶ A Procedural Interpretation of *Declarative Specifications*
- ▶ Non-procedural Programming
- ▶ Algorithms minus Control
- ▶ Computations as *Deduction*
- ▶ Theorem Proving

# A Very High Level Language

- ▶ A good programming language should not encumber the programmer with non-essential details.
- ▶ The development of programming languages has been toward freeing the programmer of more and more of the details
  - ▶ ASSEMBLY LANGUAGE: symbolic encoding of data and instructions.
  - ▶ FORTRAN: allocation of variables to memory locations, register saving, etc.
  - ▶ ALGOL: environment manipulations
  - ▶ LISP: memory management
  - ▶ ADA: name conflicts
  - ▶ ML: explicit variable type declarations
  - ▶ JAVA: Platform specifics
- ▶ Logic Programming Languages are a class of languages which attempt to free us from having to worry about many aspects of explicit control.

# A Procedural Interpretation of Declarative Specifications

- ▶ One can take a logical statement like the following: For all  $X$  and  $Y$ ,  $X$  is the father of  $Y$  if  $X$  is a parent of  $Y$  and the gender of  $X$  is male.
- ▶ which would be expressed in Prolog as: `father(X,Y) : parent(X,Y), gender(X,male).`
- ▶ interpret it in two slightly different ways:
  - ▶ declaratively - as a statement of the truth conditions which must be true if a father relationship holds.
  - ▶ procedurally - as a description of what to do to establish that a father relationship holds.

# Non-procedural Programming

- ▶ Logic Programming languages are non - procedural programming languages.
- ▶ A non - procedural language one in which one specifies WHAT needs to be computed but not HOW it is to be done.
- ▶ That is, one specifies:
  - ▶ the set of objects involved in the computation
  - ▶ the relationships which hold between them
  - ▶ the constraints which must hold for the problem to be solved
- ▶ and leaves it up to the language interpreter or compiler to decide HOW to satisfy the constraints.

# Algorithms Minus Control

- ▶ Nikolas Wirth (architect of Pascal) used the following slogan as the title of a book:  $\text{Algorithms} + \text{Data Structures} = \text{Programs}$
- ▶ Bob Kowalski offers a similar one to express the central theme of logic programming:  $\text{Algorithms} = \text{Logic} + \text{Control}$
- ▶ We can view the LOGIC component as: A specification of the essential logical constraints of a particular problem
- ▶ and CONTROL component as: Advice to an evaluation machine (e.g. an interpreter or compiler) on how to go about satisfying the constraints)

# Computation as Deduction

- ▶ Logic programming offers a slightly different paradigm for computation: COMPUTATION IS LOGICAL DEDUCTION
- ▶ It uses the language of logic to express data and programs. For all  $X$  and  $Y$ ,  $X$  is the father of  $Y$  if  $X$  is a parent of  $Y$  and the gender of  $X$  is male.
- ▶ Current logic programming languages use first order logic (FOL) which is often referred to as first order predicate calculus (FOPC).
- ▶ The first order refers to the constraint that we can quantify (i.e. generalize) over objects, but not over functions or relations. We can express "All elephants are mammals but not" for every continuous function  $f$ , if  $n < m$  and  $f(n) < 0$  and  $f(m) > 0$  then there exists an  $x$  such that  $n < x < m$  and  $f(x) = 0$



# Theorem Proving

- ▶ Logic Programming uses the notion of an automatic theorem prover as an interpreter.
- ▶ The theorem prover derives a desired solution from an initial set of axioms.
- ▶ Note that the proof must be a "constructive" one so that more than a true/false answer can be obtained.
- ▶ E.G. The answer to exists  $x$  such that  $x = \text{sqrt}(16)$  should be  $x = 4$  or  $x = -4$  rather than true

# A Short History

- 1965 Efficient theorem provers. Resolution (Alan Robinson)
- 1969 Theorem Proving for problem solving. (Cordell Green)
- 1969 PLANNER, theorem proving as programming (Carl Hewett)
- 1970 Micro - Planner, an implementation (Sussman, Charniak and Winograd)
- 1970 Prolog, an implementation (Alain Colmerauer)
- 1972 Book: Logic for Problem Solving. (Kowalski)
- 1977 DEC - 10 Prolog, an efficient interpreter/compiler (Warren and Pereira)
- 1982 Japan's 5th Generation Computer Project
- 1985 Datalog and deductive databases
- 1995 Prolog interpreter embedded in NT

# PROLOG is the FORTRAN of Logic Programming

- ▶ Prolog is the only widely used logic programming language.
- ▶ As a Logic Programming language, it has a number of advantages: simple, small, fast, easy to write good compilers for it.
- ▶ and disadvantages
  - ▶ It has a fixed control strategy.
  - ▶ It has a strong procedural aspect
  - ▶ limited support parallelism or concurrency or multi -threading.

# Logical Computation

Computation is based on logical facts and deduction rules.

*Computation is related to logical proofs and is not restricted to functional (Church) or imperative (Turing/Von Neumann) computation models.*

# Computing with Logic

- Expert Systems
- Search-based Software Engineering
- Program Analysis
- Automated Theorem Proving
- ...

## Common Theme

Search for the existence of a solution (in a graph) by utilizing known facts (nodes) and relationships (edges)

# To Understand Computing with Logic

We need to understand Logic.

## Logic

- Declarative Statements describing the state of the world.
  - Declarative Statements are either true or false
- Rules of Reasoning use existing declarative statements to conclude new declarative statements.

# Basic Constituents of Logic

- ① Individuals in the world (constants).
- ② Relations over these individuals (properties or predicates): E.g., Edges between nodes.
  - Relations have arity (number of individuals involved in the relation)  
Facts and Rules.  
E.g.,  $n$  is a node,  $n$  has an edge to  $n'$ .
- ③ Quantifiers and variables used to describe all or some individuals

# Another Example

## Declarative Statements: Facts and Rules

- ① Every mother loves her children.
- ② Mary is a mother and Tom is Mary's child.

## Queries

Does Mary love Tom?



## Another Example

- Constants: *mary*, *tom*, ...
- Predicates: *isamother*/1, *childof*/2, *loves*/2

Declarative Statements: Facts and Rules

Facts Mary is a mother *isamother(mary)*

Facts Tom is Mary's child. *childof(tom, mary)*

# Another Example

- Constants: *mary*, *tom*, ...
- Predicates: *isamother*/1, *childof*/2, *loves*/2

Declarative Statements: Facts and Rules

**Facts** Mary is a mother *isamother(mary)*

**Facts** Tom is Mary's child. *childof(tom, mary)*

**Rule** Every mother loves her children.

# Another Example

- Constants: *mary*, *tom*, ...
- Predicates: *isamother*/1, *childof*/2, *loves*/2

Declarative Statements: Facts and Rules

**Facts** Mary is a mother *isamother*(*mary*)

**Facts** Tom is Mary's child. *childof*(*tom*, *mary*)

**Rule** Every mother loves her children.

$\forall X. \forall Y. (\text{loves}(X, Y) \quad (\text{isamother}(X) \wedge \text{childof}(Y, X)))$

# Another Example

- Constants: *mary*, *tom*, ...
- Predicates: *isamother*/1, *childof*/2, *loves*/2

Declarative Statements: Facts and Rules

**Facts** Mary is a mother *isamother*(*mary*)

**Facts** Tom is Mary's child. *childof*(*tom*, *mary*)

**Rule** Every mother loves her children.

$\forall X. \forall Y. (\text{loves}(X, Y) \quad (\text{isamother}(X) \wedge \text{childof}(Y, X)))$

Queries:

Does Mary love Tom?

# Another Example

- Constants: *mary*, *tom*, ...
- Predicates: *isamother*/1, *childof*/2, *loves*/2

Declarative Statements: Facts and Rules

**Facts** Mary is a mother *isamother(mary)*

**Facts** Tom is Mary's child. *childof(tom, mary)*

**Rule** Every mother loves her children.

$\forall X. \forall Y. (\text{loves}(X, Y) \quad (\text{isamother}(X) \wedge \text{childof}(Y, X)))$

Queries:

Does Mary love Tom?

*true* <sup>?</sup> *loves(mary, tom)*

# Another Example

- Constants: *mary*, *tom*, ...
- Predicates: *isamother*/1, *childof*/2, *loves*/2

Declarative Statements: Facts and Rules

**Facts** Mary is a mother *isamother(mary)*

**Facts** Tom is Mary's child. *childof(tom, mary)*

**Rule** Every mother loves her children.

$\forall X. \forall Y. (\text{loves}(X, Y) \quad (\text{isamother}(X) \wedge \text{childof}(Y, X)))$

Queries:

Does Mary love Tom?

*true* <sup>?</sup> *loves(mary, tom)* true

# Horn Clause

Alfred Horn

A Horn Clause:

$$c \quad h_1 \wedge h_2 \wedge \dots \wedge h_n$$

where  $c$  is the consequent and the conjunction of  $h_i$ s is the antecedent.

# Horn Clause

Alfred Horn

A Horn Clause:

$$c \quad h_1 \wedge h_2 \wedge \dots \wedge h_n$$

where  $c$  is the consequent and the conjunction of  $h_i$ s is the antecedent.

*If all  $h_i$ s are true then  $c$  is true*



Horn Clause:  $c \bigwedge_i h_i$

A Horn clause

$$c \leftarrow h_1 \wedge h_2 \wedge \dots \wedge h_n$$

is written in prolog as

`c :- h1, h2, ..., hn`

Horn Clause:  $c \bigwedge_i h_i$

A Horn clause

$$c \leftarrow h_1 \wedge h_2 \wedge \dots \wedge h_n$$

is written in prolog as

$c \text{ :- } h_1, h_2, \dots, h_n$

Horn Clause	Prolog
Consequent $c$	goal
Antecedent $\bigwedge_i h_i$	subgoals
Horn Clause with no Antecedent	Fact
Horn Clause with Antecedent	Rule
Horn Clause with no Consequent	Query

# Prolog

## Facts

```
isamother(mary).          %% Horn Clause with no Antecedent  
childof(tom, mary).      %% Horn Clause with no Antecedent
```

# Prolog

## Facts

```
isamother(mary).          %% Horn Clause with no Antecedent  
childof(tom, mary).      %% Horn Clause with no Antecedent
```

## Rules

```
%% Rule: Horn Clause with antecedent  
loves(mary, tom) :-  
    isamother(mary), childof(tom, mary).
```

# Prolog

## Facts

```
isamother(mary).          %% Horn Clause with no Antecedent
childof(tom, mary).      %% Horn Clause with no Antecedent
```

## Rules

```
%% Rule: Horn Clause with antecedent
loves(mary, tom) :-
    isamother(mary), childof(tom, mary).
```

## Query

```
%% Query: Horn Clause with no consequent
?- loves(mary, tom).
```

# What about Variables?

A Horn clause with variables

$$\forall X_1 \dots X_n, Y_1 \dots Y_m. c(X_1, X_2, \dots, X_n) \leftarrow f(X_1, X_2, \dots, X_n, Y_1, Y_2, \dots, Y_m)$$

$$\forall X_1 \dots X_n. c(X_1, X_2, \dots, X_n) \leftarrow \exists Y_1 \dots Y_m. f(X_1, X_2, \dots, X_n, Y_1, Y_2, \dots, Y_m)$$

where  $f$  is a conjunction of relations

# Prolog

## Facts

```
isamother(mary).      %% Horn Clause with no Antecedent  
childof(tom, mary).   %% Horn Clause with no Antecedent  
childof(jerry, mary). %% Horn Clause with no Antecedent
```

# Prolog

## Facts

```
isamother(mary).      %% Horn Clause with no Antecedent
childof(tom, mary).   %% Horn Clause with no Antecedent
childof(jerry, mary). %% Horn Clause with no Antecedent
```

## Rules

```
%% Rule: Horn Clause with antecedent and with variables
%%      X and Y are universally quantified
loves(X, Y) :-
    isamother(X), childof(Y, X).

%%      X is universally quantified
%%      Y, Z are existentially quantified
hassibling(X) :-
    childof(X, Y), childof(Z, Y).
```



# Prolog

## Facts

```
isamother(mary).          %% Horn Clause with no Antecedent
childof(tom, mary).       %% Horn Clause with no Antecedent
childof(jerry, mary).     %% Horn Clause with no Antecedent
```

## Rules

```
%% Rule: Horn Clause with antecedent and with variables
%%      X and Y are universally quantified
loves(X, Y) :-
    isamother(X), childof(Y, X).

%%      X is universally quantified
%%      Y, Z are existentially quantified
hassibling(X) :-
    childof(X, Y), childof(Z, Y).
```

## Query

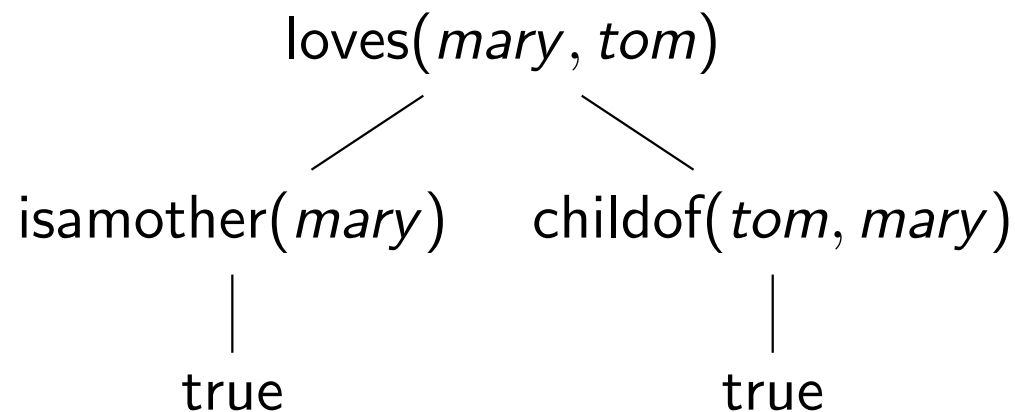
```
%% Query: Horn Clause with no consequent
?- loves(mary, X). %% X is existentially quantified
?- hassibling(jerry).
```

# Programming with Logic: Prolog

## Facts, Rules

```
isamother(mary).
childof(tom, mary).
```

```
loves(X, Y) :-
    isamother(X),
    childof(Y, X).
```



## Queries

```
?- loves(mary, tom).
Yes
```

# Queries with variables

Queries:

`?- loves(mary, X).`

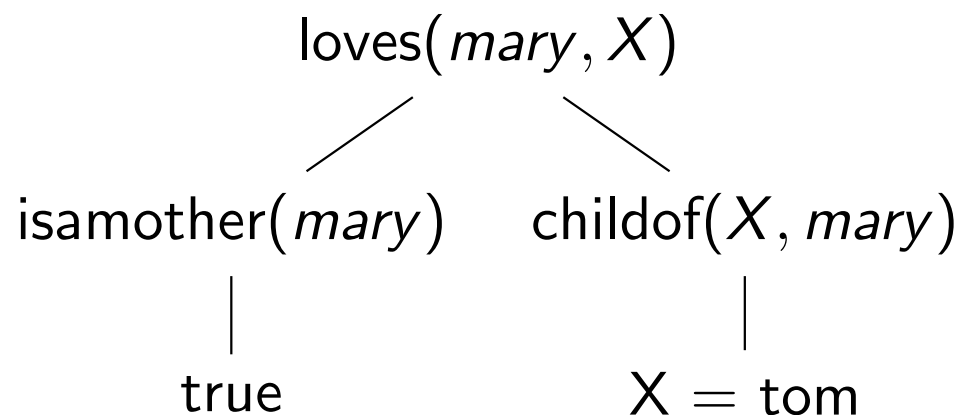
means: does there exists an  $X$  such that `loves(mary,  $X$ )` is true.

# Queries with variables

Queries:

?- loves(mary, X).

means: does there exists an  $X$  such that loves( $mary$ ,  $X$ ) is true.

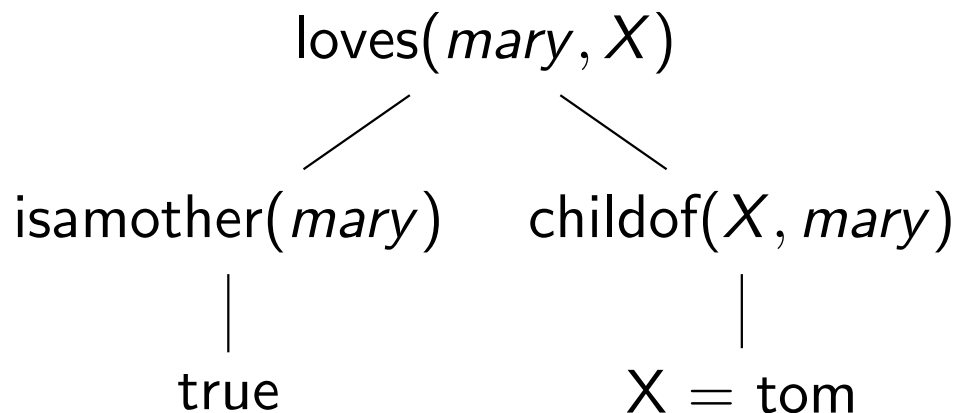


# Queries with variables

Queries:

?- loves(mary, X).

means: does there exists an  $X$  such that loves( $mary$ ,  $X$ ) is true.



Queries with free variables will generate a binding for free variables

## Queries with variables

```
isamother(mary).  
childof(jane, mary).  
childof(tom, mary).
```

```
loves(X, Y) :- isamother(X), childof(Y, X).
```

What is the result of `?- loves(mary, X)`

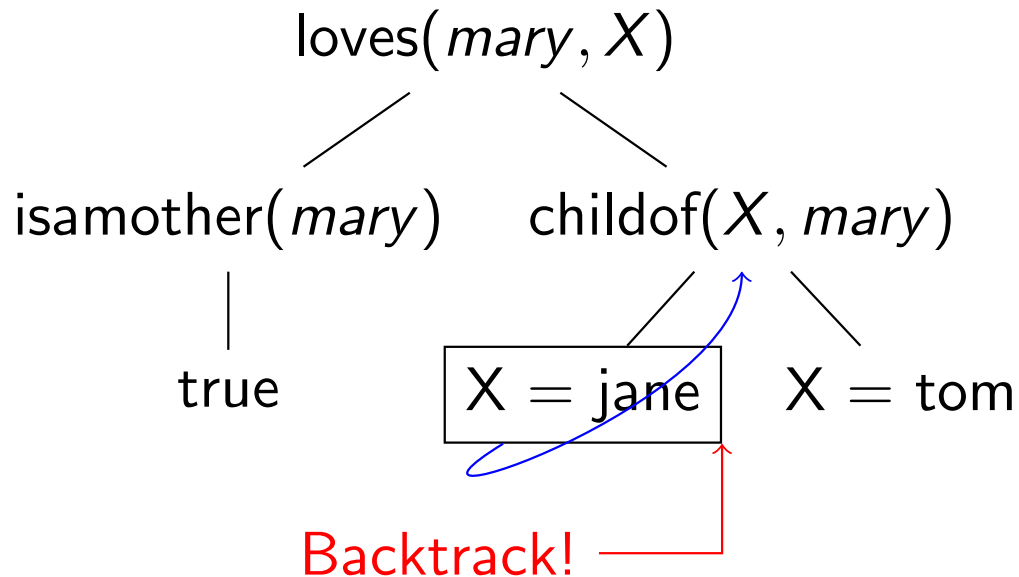
# Queries with variables

```
isamother(mary).
childof(jane, mary).
childof(tom, mary).
```

```
loves(X, Y) :- isamother(X), childof(Y, X).
```

What is the result of `?- loves(mary, X)`

Computes all possible way to satisfy  $\exists X.\text{loves}(\text{mary}, X)$ .



# Syntax of Logic Program

Logic program is a collection of Horn Clauses



# Syntax of Logic Program

Logic program is a collection of Horn Clauses

- How do we write

$$c \leftarrow h_1 \vee h_2$$

as a Horn Clause Statement

# Syntax of Logic Program

Logic program is a collection of Horn Clauses

- How do we write

$$c \leftarrow h_1 \vee h_2$$

as a Horn Clause Statement

$$c \leftarrow h_1$$

$$c \leftarrow h_2$$

# Syntax of Logic Program

Logic program is a collection of Horn Clauses

- How do we write

$$c \leftarrow h_1 \vee h_2$$

as a Horn Clause Statement

$$c \leftarrow h_1$$

$$c \leftarrow h_2$$

`edge(a, b).`

`edge(b, c).`

`edge(c, c).`

`reach(X, Y) :- edge(X, Y).`

`reach(X, Y) :- edge(X, Z), reach(Z, Y).`

# Syntax of Logic Programs

## Terms

constants, Variables, functors (un-interpreted functions with terms as arguments)

## Formulas

predicates with terms as arguments, boolean combination of predicates and universal/existentially quantified variables followed by predicates.

# Syntax of Logic Programs

```

Clause -> Predicate.                <-- fact
        | Predicate :- PredicateSeq. <-- rule
PredicateSeq -> Predicate
               | Predicate, PredicateSeq

```

```

Predicate -> PredName(TermSeq)
TermSeq    -> Term
           | Term, TermSeq

```

```

Term      -> FunctorName(TermSeq) | Constant | Variable

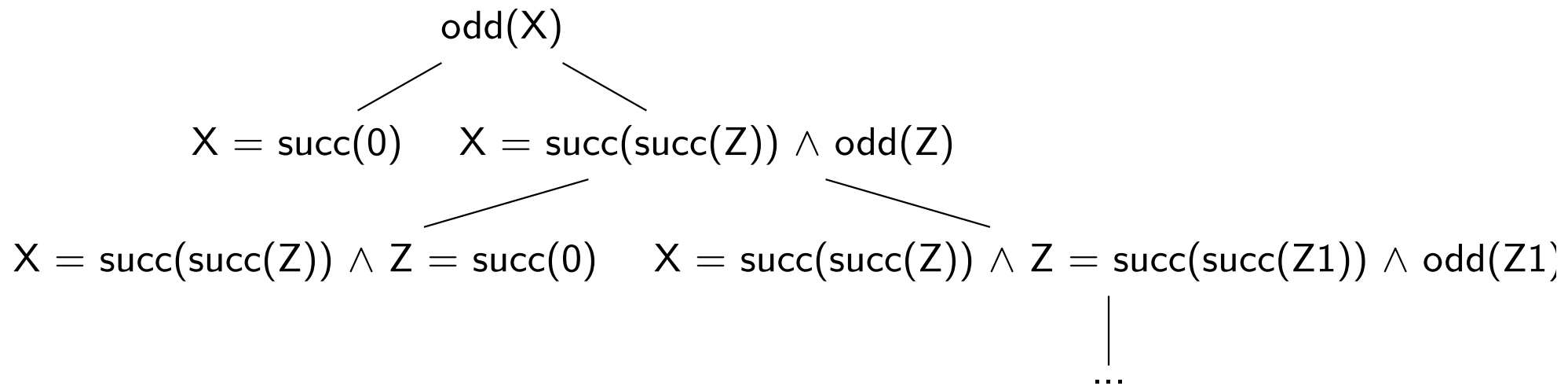
```

# Example

```
odd(succ(0)). %% fact using a Functor
```

```
odd(succ(succ(Z))) :- odd(Z). %% rule using a Functor
```

```
?- odd(X). %% Query
```



```
?- odd(X), odd(succ(X)). %% Query: what is the result?
```

# Logical Implications

- Prolog input: facts and rules (relations)
- Queries: does some inference hold?
- Proof by application logical implication

## Resolution

Horn clauses: a rule-based logical formula

# Executing Logic Programs

Unification/Most General Unifiers

Variable bindings

Backward Chaining/Goal-directed Reasoning

Reducing one proof obligation (goal) into simpler ones (subgoals).

Backtracking

Search for proofs (answers).



# Unification

Given two atomic formula (predicates), they can be unified if and only if they can be made syntactically identical by replacing the variables in them by some terms.

# Unification

Given two atomic formula (predicates), they can be unified if and only if they can be made syntactically identical by replacing the variables in them by some terms.

- Unify `childof(jane, X)` and `childof(jane, mary)`?

# Unification

Given two atomic formula (predicates), they can be unified if and only if they can be made syntactically identical by replacing the variables in them by some terms.

- Unify `childof(jane, X)` and `childof(jane, mary)`?  
yes by replacing `X` by `mary`

# Unification

Given two atomic formula (predicates), they can be unified if and only if they can be made syntactically identical by replacing the variables in them by some terms.

- Unify `childof(jane, X)` and `childof(jane, mary)`?  
yes by replacing `X` by `mary`
- Unify `childof(jane, X)` and `childof(jane, Y)`?

# Unification

Given two atomic formula (predicates), they can be unified if and only if they can be made syntactically identical by replacing the variables in them by some terms.

- Unify `childof(jane, X)` and `childof(jane, mary)`?  
yes by replacing `X` by `mary`
- Unify `childof(jane, X)` and `childof(jane, Y)`?  
yes by replacing `X` and `Y` by the same individual

# Unification

Given two atomic formula (predicates), they can be unified if and only if they can be made syntactically identical by replacing the variables in them by some terms.

- Unify `childof(jane, X)` and `childof(jane, mary)`?  
yes by replacing `X` by `mary`
- Unify `childof(jane, X)` and `childof(jane, Y)`?  
yes by replacing `X` and `Y` by the same individual
- Unify `childof(jane, X)` and `childof(Y, mary)`?

# Unification

Given two atomic formula (predicates), they can be unified if and only if they can be made syntactically identical by replacing the variables in them by some terms.

- Unify `childof(jane, X)` and `childof(jane, mary)`?  
yes by replacing `X` by `mary`
- Unify `childof(jane, X)` and `childof(jane, Y)`?  
yes by replacing `X` and `Y` by the same individual
- Unify `childof(jane, X)` and `childof(Y, mary)`?  
yes by replacing `X` by `mary`, and `Y` by `jane`
- Unify `childof(jane, X)` and `childof(tom, Y)`?

# Unification

Given two atomic formula (predicates), they can be unified if and only if they can be made syntactically identical by replacing the variables in them by some terms.

- Unify `childof(jane, X)` and `childof(jane, mary)`?  
yes by replacing `X` by `mary`
- Unify `childof(jane, X)` and `childof(jane, Y)`?  
yes by replacing `X` and `Y` by the same individual
- Unify `childof(jane, X)` and `childof(Y, mary)`?  
yes by replacing `X` by `mary`, and `Y` by `jane`
- Unify `childof(jane, X)` and `childof(tom, Y)`? No.



# Substitution

Substitution maps variables to terms.

Instantiation is the application of substitution to all variables in a prolog formula, term.

- Unify `childof(jane, X)` and `childof(Y, mary)`?  
yes

# Substitution

Substitution maps variables to terms.

Instantiation is the application of substitution to all variables in a prolog formula, term.

- Unify `childof(jane, X)` and `childof(Y, mary)`?  
yes by  $[X \mapsto \text{mary}, Y \mapsto \text{jane}]$

# Substitution

Substitution maps variables to terms.

Instantiation is the application of substitution to all variables in a prolog formula, term.

- Unify `childof(jane, X)` and `childof(Y, mary)`?  
yes by  $[X \mapsto \text{mary}, Y \mapsto \text{jane}]$
- Unify `p(f(X), X)` and `p(Y, a)`?  
yes

# Substitution

Substitution maps variables to terms.

Instantiation is the application of substitution to all variables in a prolog formula, term.

- Unify `childof(jane, X)` and `childof(Y, mary)`?  
yes by  $[X \mapsto \text{mary}, Y \mapsto \text{jane}]$
- Unify `p(f(X), X)` and `p(Y, a)`?  
yes by  $[X \mapsto a, Y \mapsto f(a)]$

# Most General Unifier

MGU results from a substitution that bounds free variables as little as possible

- Unify  $p(X, f(Y))$  and  $p(g(Z), W)$

# Most General Unifier

MGU results from a substitution that bounds free variables as little as possible

- Unify  $p(X, f(Y))$  and  $p(g(Z), W)$ 
  - $[X \mapsto g(a), Y \mapsto b, W \mapsto f(b)]$
  - $[X \mapsto g(Z), W \mapsto f(Y)]$

# Most General Unifier

MGU results from a substitution that bounds free variables as little as possible

- Unify  $p(X, f(Y))$  and  $p(g(Z), W)$ 
  - $[X \mapsto g(a), Y \mapsto b, W \mapsto f(b)]$
  - $[X \mapsto g(Z), W \mapsto f(Y)]$  **MGU**

# Most General Unifier

MGU results from a substitution that bounds free variables as little as possible

- Unify  $p(X, f(Y))$  and  $p(g(Z), W)$ 
  - $[X \mapsto g(a), Y \mapsto b, W \mapsto f(b)]$
  - $[X \mapsto g(Z), W \mapsto f(Y)]$  **MGU**
- Unify  $f(W, g(Z), Z)$  and  $f(X, Y, h(X))$

MGU?



# Unification and Computing with Logic

Given a query (prove/disprove a predicate holds)

- Search the facts and rules to find whether the query unifies with any consequent
- If the search fails, return false (query result)
- If the search is successful, then
  - if the unification occurs with the consequent of a fact, return the substitution of the variables (if any)
  - if the unification occurs with the consequent of a rule, instantiate the variables (if any) and prove the subgoals

## Example: Recap

### Facts, Rules

```
isamother(mary).  
childof(tom, mary).
```

```
loves(X, Y) :-  
    isamother(X),  
    childof(Y, X).
```

### Queries

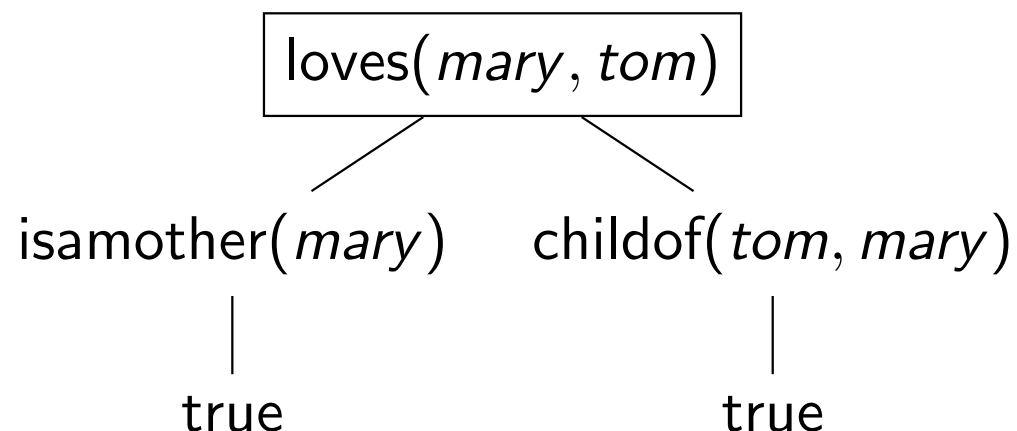
```
?- loves(mary, tom).  
Yes
```

# Example: Recap

## Facts, Rules

```
isamother(mary).
childof(tom, mary).
```

```
loves(X, Y) :-
    isamother(X),
    childof(Y, X).
```



## Queries

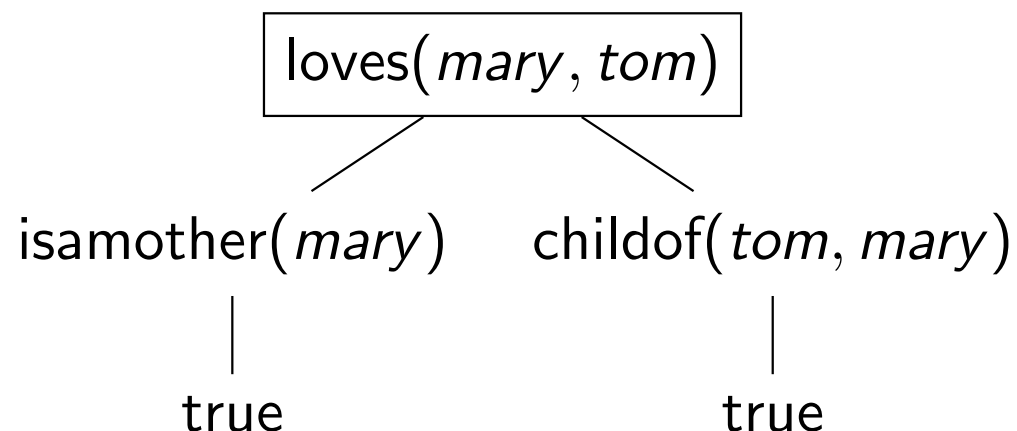
```
?- loves(mary, tom).
Yes
```

# Example: Recap

## Facts, Rules

```
isamother(mary).
childof(tom, mary).
```

```
loves(X, Y) :-
    isamother(X),
    childof(Y, X).
```



## Queries

Query unifies with the rule  
 MGU:  $[X \mapsto \text{mary}, Y \mapsto \text{tom}]$ .

```
?- loves(mary, tom).
Yes
```

# Backtracking

Given a query with free variables (prove/disprove a predicate holds)

- Search the facts and rules to find whether the query unifies with any consequent
- If the search fails, return false (query result)
- If the search is successful, then
  - if the unification occurs with the consequent of a fact, return the substitution of the variables (if any)
    - Re-try to unify with some clause other than the fact, and proceed
  - if the unification occurs with the consequent of a rule, instantiate the variables (if any) and prove the subgoals
    - Re-try to unify with some clause other than the fact, and proceed

# Backtracking

Given a query with free variables (prove/disprove a predicate holds)

- Search the facts and rules to find whether the query unifies with any consequent
- If the search fails, return false (query result)
- If the search is successful, then
  - if the unification occurs with the consequent of a fact, return the substitution of the variables (if any)
    - Re-try to unify with some clause other than the fact, and proceed
  - if the unification occurs with the consequent of a rule, instantiate the variables (if any) and prove the subgoals
    - Re-try to unify with some clause other than the fact, and proceed

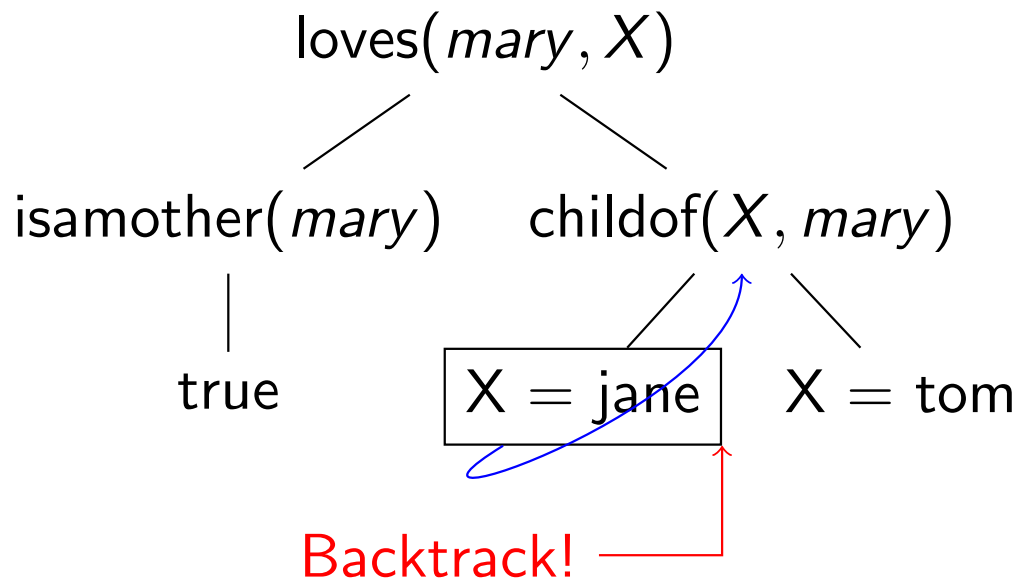
# Backtracking: Example Recap

```
isamother(mary).
childof(jane, mary).
childof(tom, mary).
```

```
loves(X, Y) :- isamother(X), childof(Y, X).
```

What is the result of `?- loves(mary, X)`

Computes all possible way to satisfy  $\exists X.\text{loves}(\text{mary}, X)$ .



# Search for solution

```
edge(a, b).  
edge(b, c).  
edge(c, a).
```

```
reach(X, Y) :-  
    edge(X, Y).  
reach(X, Y) :-  
    edge(X, Z),  
    reach(Z, Y).
```



# Search for solution

```
edge(a, b).  
edge(b, c).  
edge(c, a).
```

```
reach(X, Y) :-  
    edge(X, Y).  
reach(X, Y) :-  
    edge(X, Z),  
    reach(Z, Y).
```

```
?- reach(a, c)
```

# More Language Features

- Lists
- Numbers
- if-then-else

# Lists

A list is ordered sequence of terms enclosed in [...]

- `[a, b, c]`: list containing three elements/atoms a, b and c
- `[]`: empty list
- `[a, [b, c], [[d, e]], []]`: list can contain elements of different types
- `[a|[b, c]]`: same as `[a, b, c]`, a is called the head of the list and `[b, c]` is the tail of the list

?- `[1, 2, 3] = [X|Xs].`

`X = 1`

`Xs = [2, 3]`

?- `[1, 2, 3] = [X|[Y|Rest]].`

`X = 1`

`Y = 2`

`Rest = [3]`

# Unification vs Computation

- $X = 3$ :  $X$  is unified to 3 (assignment w/o computation)
- $X = 3 + 1$ :  $X$  is unified to  $3 + 1$  (not 4)
- $X \text{ is } 3 + 1$ :  $X$  is assigned to 4

# Unification vs Computation

- $X = 3$ :  $X$  is unified to 3 (assignment w/o computation)
- $X = 3 + 1$ :  $X$  is unified to  $3 + 1$  (not 4)
- $X \text{ is } 3 + 1$ :  $X$  is assigned to 4

?-  $X \text{ is } Y + 1$ .

Uninstantiated argument of evaluable function  $+/2$

?-  $X \text{ is } 3, X = 3$ .

$X = 3$

?-  $X = 3, Y \text{ is } X + 1$ .

$X = 3$

$Y = 4$

?-  $X \text{ is } 3, X \text{ is } X + 1$ .

no

# if-then-else

?- Z = 3, (Z == 3 -> X = 1, Y = 2; X = 2, Y = 1).

Z = 3

X = 1

Y = 2

# if-then-else

```
?- Z = 3, (Z == 3 -> X = 1, Y = 2; X = 2, Y = 1).
```

```
Z = 3
```

```
X = 1
```

```
Y = 2
```

```
mypred(Z, X, Y) :-
```

```
    (Z == 3
```

```
        -> X = 1,
```

```
          Y = 2
```

```
    ; X = 2,
```

```
      Y = 1
```

```
).
```

```
mypred(Z, X, Y) :-
```

```
    Z == 3, X = 1, Y = 2.
```

```
mypred(Z, X, Y) :-
```

```
    Z \= 3, X = 2, Y = 1.
```

# Example

Append one list to another

- appending an empty list  $L_1$  to list  $L_2$  results in  $L_2$
- appending a non-empty list  $L_1$  to list  $L_2$  results in  $L$  if the head of  $L_1$  and  $L$  are the same and the tail of  $L$  is obtained by appending the tail of  $L_1$  to list  $L_2$

If  $\mathcal{L}$  represents the set of lists, then signature of append is

$$\mathcal{L} \times \mathcal{L} \times \mathcal{L} \subseteq \text{append}$$

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

`append([X|Xs], L, [X|Ys]) :-  
    append(Xs, L, Ys).`



## Example

Reverse a list (again!)

```
reverse([], []).  
reverse([X|Xs], L) :- reverse(Xs, Ys), append(Ys, [X], L).
```

## Example

Reverse a list (again!)

```
reverse([], []).  
reverse([X|Xs], L) :- reverse(Xs, Ys), append(Ys, [X], L).
```

Length of a list

```
length([], 0).  
length([X|Xs], N) :- length(Xs, M), N is M + 1.
```

## Example

Reverse a list (again!)

```
reverse([], []).
reverse([X|Xs], L) :- reverse(Xs, Ys), append(Ys, [X], L).
```

Length of a list

```
length([], 0).
length([X|Xs], N) :- length(Xs, M), N is M + 1.
```

How about?

```
length([], 0).
length([X|Xs], N) :- M is N - 1, length(Xs, M).
```