Lecture 9. Logic Programming

November 19, 2018

Revisit the previous example

Reverse a list

Imperative Programming

```
void reverse(struct node** head_ref)
{
   struct node* prev   = NULL;
   struct node* current = *head_ref;
   struct node* next;
   while (current != NULL)
   {
        next  = current->next;
        current->next = prev;
        prev = current;
        current = next;
   }
   *head_ref = prev;
}
```

Functional Programming

```
(define (rev lst)
  (if (null? lst)
    lst
    (append (rev (cdr lst))
            (list (car lst)))))
```

input output fact

Logic Programming

```
rev([], []).
rev([H|T, L) :-
    rev(T, T1),
    append(T1, [H], L).
```

rule

[H|T], L

① rev(T, T1)

② Append(T1, [H], L)

Query: rev([1,2,3], L).

L = [3, 2, 1]

Logic Programming : propositional logic

predicate / first order logic

Introduction to Logic Programming

Truly Declarative Paradigm

- User declares **what facts** are true.
- User states some queries.
- System determinines **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.

NLP processing

Compiler

C + prolog

prolog + Python

Interpreter of Prolog

- "Execute a program": make inference from a database of facts and rules.
- Presenting knowledge: propositional logic → fact & logical connectives
  - fact: proposition that's unconditional true
  - rule: proposition that's conditional true; dependent on other propositions
  - a fact or a rule is a statement or clause in Prolog.

first order logic

predicate : variables

quantifiers, relations

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: Forall 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.

test conditions

Non-procedural Programming

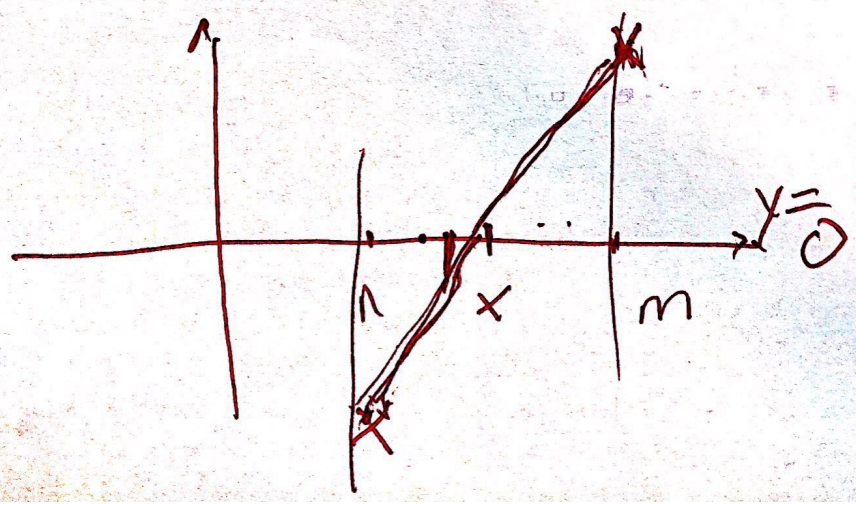
- Logic Programming languages are non - procedural programming languages.
- A non - procedural language one in which one specifies WHAT needs to be computed but not HOW it is to be done.
- That is, one specifies:
  - the set of objects involved in the computation
  - the relationships which hold between them
  - the constraints which must hold for the problem to be solved
- and leaves it up the 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: Algorithms + Data Structures = Programs
- Bob Kowalski offers a similar one to express the central theme of logic programming: Algorithms = Logic + 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. Forall 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. W e 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 = sqrt(16) should be x = 4 or x = -4 rather than true

Provide an example

demonstrate

proof

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.

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)

Logical Computation

Computation is based on logical facts and deduction rules.

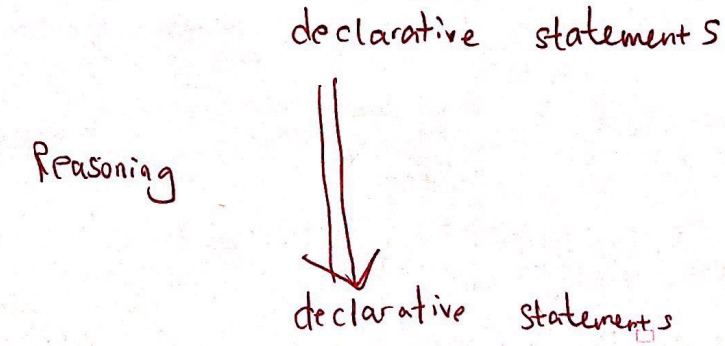
Computation is related to logical proofs and is not restricted to functional (Church) or imperative (Turing/Von Neumann) computation models.

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*)

Rule Every mother loves her children.

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

predicate

Quantifier

Queries:

Does Mary love Tom?

true $\stackrel{?}{}$ loves(*mary*, *tom*) true

Horn Clause

Logic

Alfred Horn

A Horn Clause:

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

where c is the consequent and the conjunction of h_is is the antecedent.

If all h_is are true then c is true

Horn Clause: $c \leftarrow \bigwedge_i h_i$

A Horn clause

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

is written in prolog as

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

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

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).
```

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).
```

Syntax of Logic Program

Logic program is a collection of Horn Clauses

- How do we write

$$c \leftarrow h_1 \vee h_2$$

$h_1 \wedge h_2 \wedge \cdots h_n$

as a Horn Clause Statement

$C :- h_1, h_2 \cdots h_n$

$$\left.\begin{array}{l} c \leftarrow h_1 \\ c \leftarrow h_2 \end{array}\right\}$$

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

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

Programming with Logic: Prolog

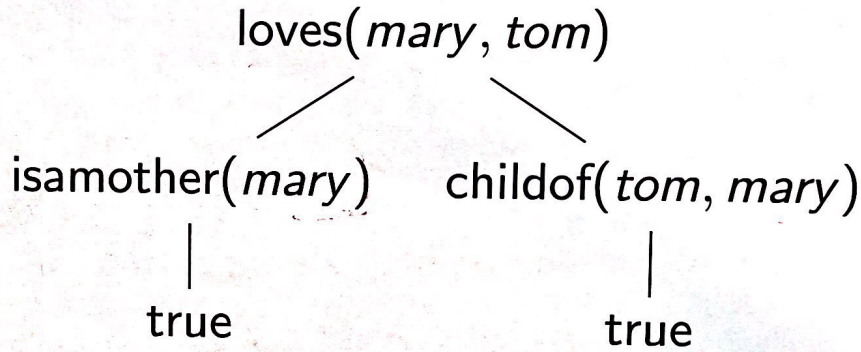
Facts, Rules

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

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

Queries

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

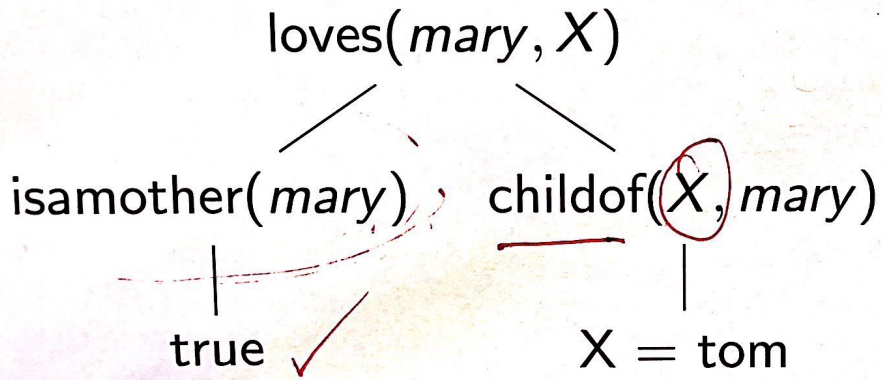


Queries with variables

Queries:

```
?- loves(mary, X).
```

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



Queries with free variables will generate a binding for free variables

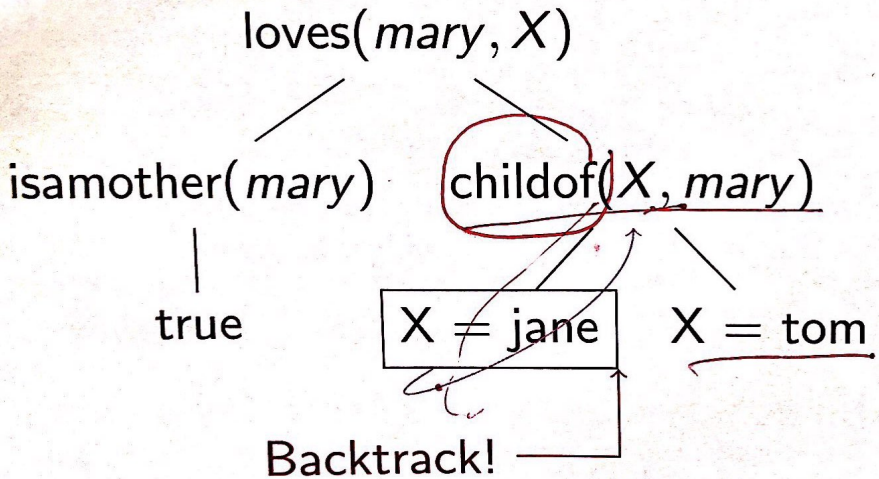
Queries with variables

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

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

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

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



Prolog : Enviroment

Prolog : Syntax

Atoms , Integer, flo

Arithmetrics , Predicates

Lists (member function).

If-then-else

Predicate: returns true / false

returns value by providing X.

Example: Recap

Facts, Rules

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

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

Consequent

Antecedent

horn clause

Queries

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

Syntax of Logic Programs

Very little syntax

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

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

Term      -> FunctorName(TermSeq) | Constant | Variable | Lists
```

?- predNam(TermSeq)

atom, number

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]
```

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).
```

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

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

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

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

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

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

?- append([1], X, [3,4]).
false

Logical Implications

Append
Member

- 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.

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

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

MGU?

$W \mapsto X$

$Z \mapsto h(X)$

$Y \mapsto g(Z)$

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) Symbolic
- `X is 3 + 1`: X is assigned to 4

```
?- X is Y + 1.
Uninstantiated argument of evaluable function +/2

?- X is 3, X = 3.
X = 3

?- X = 3, Y is X + 1.
X = 3
Y = 4

?- X is 3, X is X + 1.
no
```

Unification

- Unification is a pattern matching operation between two terms, both of which can contain variables.
- A *substitution* is an assignment of variables to values.
- Two terms unify if there is a substitution that makes the terms identical.

  Unifying f(X,2) and f(3,Y) produces X=3, Y=2
- A most general unifier (mgu) is a substitution that unifies the terms w/o 'over-assigning' any variables.
- The result of applying a most general unifier to a set of terms results in a most general instance (mgi).
- E.g., unify f(X,Y) and f(1,A)
  - A substitution: X=1, Y=2, A=2
  - The mgu: X=1, Y=A
  - The mgi: f(1,Y)

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

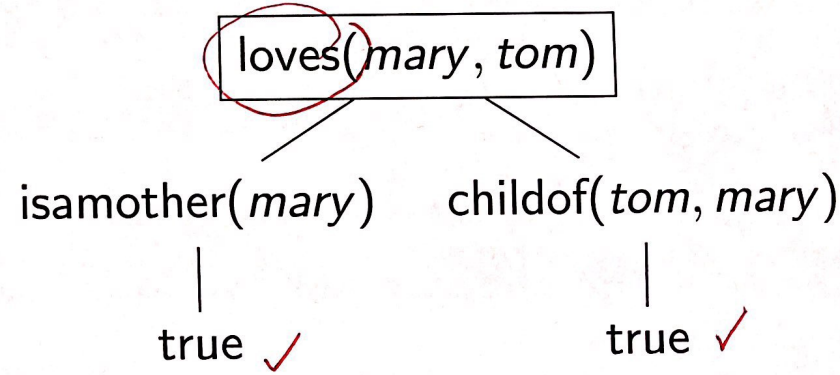
Facts, Rules

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

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

Queries

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

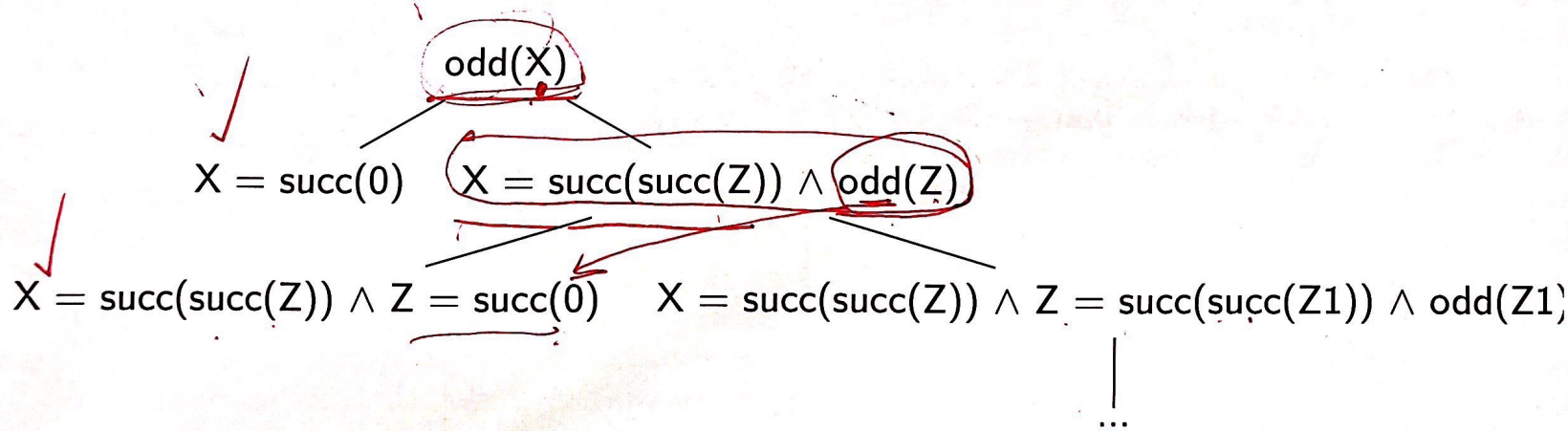


Query unifies with the rule
MGU: $[X \mapsto \texttt{mary}, Y \mapsto \texttt{tom}]$.

Example

```
odd(succ(0)).  %% fact using a Functor
odd(succ(succ(Z))) :- odd(Z). %% rule using a Functor

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



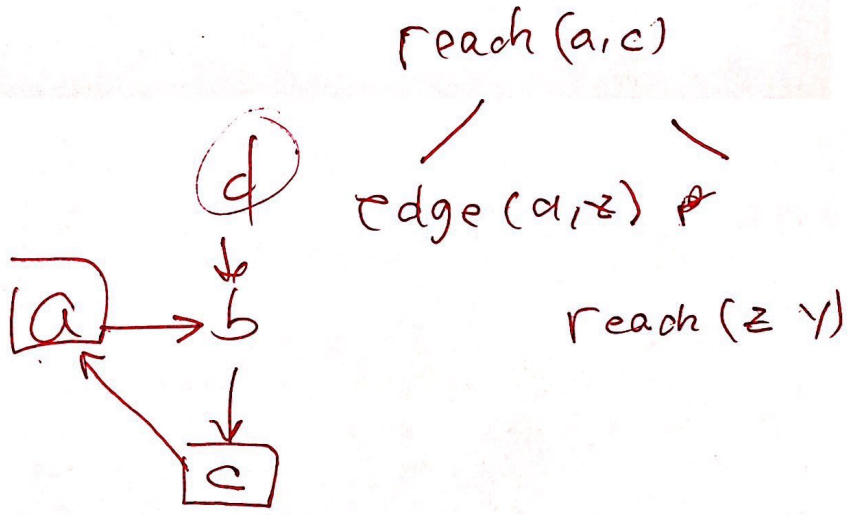

```
?- odd(X), odd(succ(X)). %% Query: what is the result?
```

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~~ rule, and proceed

Search for solution



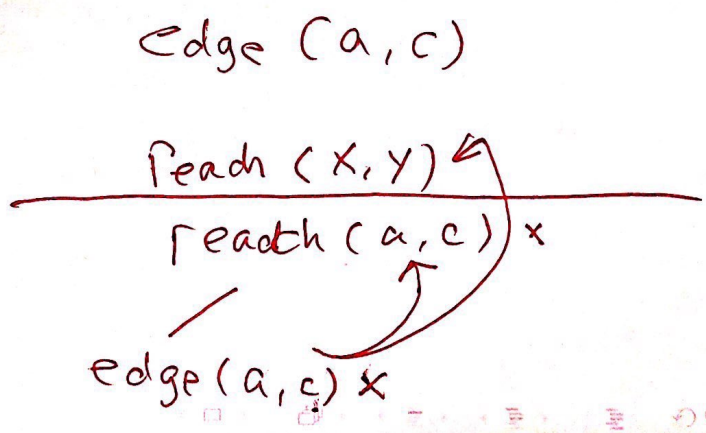

```
edge(a, b).
edge(b, c).
edge(c, a).

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

?- reach(a, c)
```

X ↦ a

Y ↦ c



Reach (a, d)

① edge (a, d) — false

② edge (a, x) — x ↦ b ✗

②-② reach (z, d) — edge (X, d) ✗

edge (Z, d)

reach (d, y) — ✗

Review:

— What is Logic programming?

- high level programming
- declarative
- 1970s prolog
- Expert systems, Natural Language Program analysis
- ~~Horn~~ Horn Clause, Horn Logic

theoretical foundations < Predicate calculs, Propositional calculs

A logical puzzle:

C:\Wei\git\weile\Teaching-Courses\cs342\cs342-2018fall\11.logicprogramming\houselogicpuzzleQuestion.txt Thursday, November 15, 2

Houses logical puzzle: who owns the zebra and who drinks water?

1) Five colored houses in a row, each with an owner, a pet, cigarettes, and a drink.
2) The English lives in the red house.
3) The Spanish has a dog.
4) They drink coffee in the green house.
5) The Ukrainian drinks tea.
6) The green house is next to the white house.
7) The Winston smoker has a serpent.
8) In the yellow house they smoke Kool.
9) In the middle house they drink milk.
10) The Norwegian lives in the first house from the left.
11) The Chesterfield smoker lives near the man with the fox.
12) In the house near the house with the horse they smoke Kool.
13) The Lucky Strike smoker drinks juice.
14) The Japanese smokes Kent.
15) The Norwegian lives near the blue house.

solution to the logical puzzle:

```
zebra_owner(Owner) :-
    houses(Hs),
    member(h(Owner,zebra,_,_,_), Hs).

water_drinker(Drinker) :-
    houses(Hs),
    member(h(Drinker,_,_,water,_), Hs).


houses(Hs) :-
    % each house in the list Hs of houses is represented as:
    %      h(Nationality, Pet, Cigarette, Drink, Color)
    length(Hs, 5),                                          %  1
    member(h(english,_,_,_,red), Hs),                       %  2
    member(h(spanish,dog,_,_,_), Hs),                       %  3
    member(h(_,_,_,coffee,green), Hs),                      %  4
    member(h(ukrainian,_,_,tea,_), Hs),                     %  5
    next(h(_,_,_,_,green), h(_,_,_,_,white), Hs),           %  6
    member(h(_,snake,winston,_,_), Hs),                     %  7
    member(h(_,_,kool,_,yellow), Hs),                       %  8
    Hs = [_,_,h(_,_,_,milk,_),_,_],                         %  9
    Hs = [h(norwegian,_,_,_,_)|_],                          % 10
    next(h(_,fox,_,_,_), h(_,_,chesterfield,_,_), Hs),      % 11
    next(h(_,_,kool,_,_), h(_,horse,_,_,_), Hs),            % 12
    member(h(_,_,lucky,juice,_), Hs),                       % 13
    member(h(japonese,_,kent,_,_), Hs),                     % 14
    next(h(norwegian,_,_,_,_), h(_,_,_,_,blue), Hs),        % 15
    member(h(_,_,_,water,_), Hs),       % one of them drinks water
    member(h(_,zebra,_,_,_), Hs).       % one of them owns a zebra
```

In-class example program for max and sum

```
max([X], X).
max([H|T],M):-max(T,N), (H>N->M is H;M is N).


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