Logic for CS

黃瀚萱
Department of Computer Science
National Chengchi University
2020 Spring

Schedule, Part I

Date	Topic
3/6	Introduction to this course
3/13	Thinking as computation
3/20	Propositional Logic
3/27	Logic Inference
4/3	Off
4/10	First Order Logic
4/17	Interpretation of FOL
4/24	Inference in FOL (Online)

Schedule, Part II

Date	Topic
5/1	Prolog Basics & KR (Online)
5/8	Midterm Exam
5/15	**Prolog Programming**
5/22	Logic Programming
5/29	Logic Programming
6/5	Applications of logic in computation
6/12	Final Project Presentation
6/19	Term Exam

Logic Programming with Prolog

The Prolog Language

- **Pro**gramming in **Log**ic: Prolog is a **computer programming language** since 1970
  - Relational databases
  - Mathematical logic
  - Abstract problem solving
  - Natural language processing
  - Design automation
  - Symbolic equation solving
  - Biochemical structure analysis
  - Other AI problems

What is a Programming Language?

- Turing completeness
  - A system of data manipulation rules can simulate a **Turing machine**.
- Turing complete languages
  - General purpose: C/C++/Java/Python/PHP/Javascript ...
  - Functional languages: Lisp/Scheme/Haskell ...
  - Declarative languages: SQL/Prolog ...
- Non-Turing-complete languages
  - Regular expression: `^([a-zA-Z0-9_\-\.]+)@([a-zA-Z0-9_\-\.]+)\.([a-zA-Z]{2,5})$`
  - HTML/CSS/JSON/XML ...

Turing Machine

- A tape consists of a sequence of cells.
  - Each cell is a symbol from a set of alphabet and a terminating symbol.
- A head can access to the tape
  - Read or write a cell
  - Move left or right on the tape
- Register that stores the state of the machine
- A transition table of instructions
  - Given a state q and the current symbol on the tape f, determines what the machine to do
    - Erase or write a symbol
    - Move the head to left or right
    - Next state

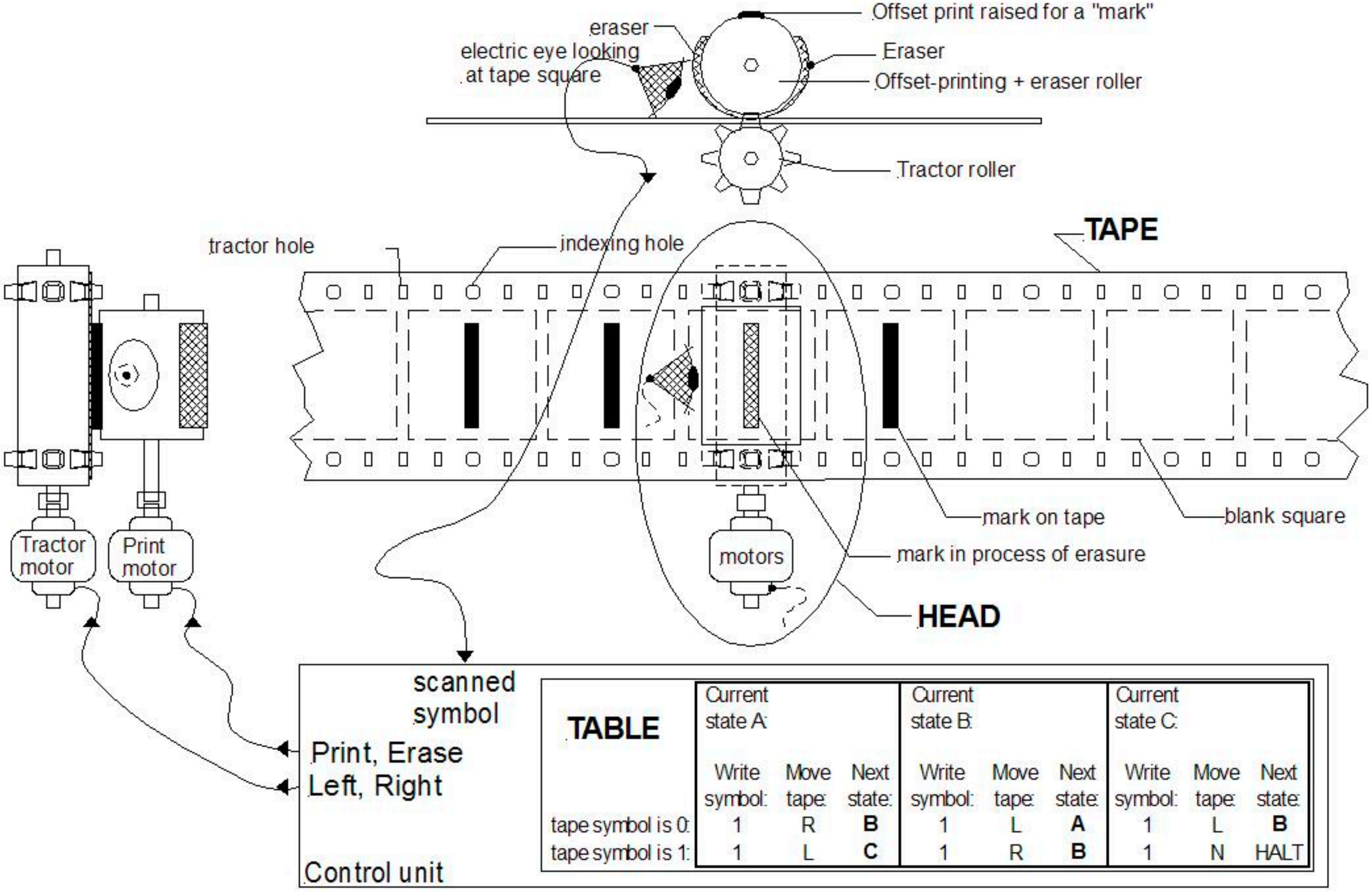


TABLE	Current state A:			Current state B:			Current state C:		
	Write symbol:	Move tape:	Next state:	Write symbol:	Move tape:	Next state:	Write symbol:	Move tape:	Next state:
tape symbol is 0:	1	R	**B**	1	L	**A**	1	L	**B**
tape symbol is 1:	1	L	**C**	1	R	**B**	1	N	HALT

A fanciful mechanical Turing machine's TAPE and HEAD. The TABLE instructions might be on another "read only" tape, or perhaps on punch-cards. Usually a "finite state machine" is the model for the TABLE.

https://en.wikipedia.org/wiki/Turing_machine_gallery

To Test a Language Turing Completeness

- Write a program in the language to simulate a Turing machine
- Essential properties of a Turing complete language
  - Can access to the RAM, registers, or other storage
  - Capable of branch
  - Capable of repetition
    - Looping
    - Recursion

A Way Different from Conventional Programming

- Prolog programmers ask more about which formal relationships and objects occur in the problem
  - And which relationships are true
- A descriptive or declarative programming language

Objects and Relationships

- Prolog is used for solving problems that involve **objects** and **relationships**.
  - Tom owns a cup
  - Own(Tom, cup)
- Not the object in object-oriented programming.
  - In OOP, an object is an instance of a class, which is data structure that can inherit attributes and functions from another class.

Structure of Prolog Programs

- A Prolog program consists of a set of clauses
- 2 types of clauses in Prolog
  - Fact about the given information
    - Atom sentence
  - Rule about how the solution may relate to or be inferred from the given facts.
    - Complex sentence

Elements

- Constants
- Variables
- Atomic sentences
- Conditional sentences
- Queries

Constants

- A Prolog constant must start with a lowercase letter and can be followed by any number of letters, underscores, or digits.
    - tom
    - sister
    - grand_father
- Can also be a single-quoted-string
    - 'how are you?'
- A predicate is a constant.

Variables

- A Prolog variable must start with an uppercase letter and can be followed by any number of letters, underscores, or digits.
  - X
  - MySon
- There are other types of variable such as list.

Atomic Sentences (Atoms)

- An atomic sentence in Prolog has the form as:
  - *predicate*(arg_1, arg_2, ..., arg_k)
- Predicate
  - A Prolog constant
- Argument
  - Either a constant or a variable

```
child(john, X)
delivers_package('UPS', Sender, Receiver)
```

Invalid Atomic Sentence

- Rich(jim)
  - A variable at the position of predicate
- likes(tom, father(john))
  - The second argument is neither a constant nor a variable.

Conditional Sentences

- The conditional sentences of Prolog have the form as:
  - $head$:- $body_1$, $body_2$, ..., $body_n$
- Both head and body are atomic sentences
- Head:
  - The effect of the conditional sentence.
- Body:
  - The causes (premises) of the conditional sentence.

```
mother(Y,X) :- child(X,Y), male(Y)
```

Syntax of a Prolog program

- A Prolog program is a sequence of clauses; a clause is an atomic or conditional sentence terminated by a period.
- For readability, spaces, newlines, and comments can be inserted at the end of a program or just before a constant or variable.
- Comments
  - Starts with a % character and continues until the end of the line.

```
child(john,sue).
child(john,sam).
child(jane,sue).
child(jane,sam).
child(sue,george).
child(sue,gina).

male(john).
male(sam).
male(george).

female(sue).
female(jane).
female(june).

parent(Y,X) :- child(X,Y).
father(Y,X) :- parent(Y,X), male(Y).
mother(Y,X) :- parent(Y,X), female(Y).
opp_sex(X,Y) :- male(X), female(Y).
opp_sex(Y,X) :- male(X), female(Y).
grand_father(X,Z) :- father(X,Y), parent(Y,Z).
```

Prolog Queries

- A Prolog program does nothing by itself.
  - It only acts in response to queries.
- The use of a Prolog program
  - Prepare a file containing the Prolog program.
  - Start the system, and ask it to load the program file.
  - Repeatedly pose a query to the system.

Queries and Outcomes

- A Prolog query is an atomic sentence terminated with a period.
  - ?- `mother(tom, jane).`
- 3 possible outcomes of a query without any variables
  - True: the atomic sentence can be established by back-chaining.
  - False: the atomic sentence cannot be established by back-chaining.
  - No answer: the atomic sentence cannot yet to be established but Prolog is continuing to try alternatives.

Outcomes

```
mother(tom, X).
X=jane
```

- Outcome of a query with variables
  - False: the atomic sentence cannot be established for any values of the variables.
  - No answer: the atomic sentence cannot yet be established for any values of the variables but that it is continuing to try.
  - Values for the variables for which it can establish the query
    - You can ask to find all possible valuations.

Conjunctive Queries

- A sequence of atomic sentences separated by commas and terminated by a period.
- DFS algorithm will try to find valuations that satisfy all conditions at the same time.

```
?- parent(sam,X).
X = john ;
X = jane.
```

```
?- parent(sam,X), female(X).
X = jane.
```

Negation in Queries

- Prolog allows negated queries with the symbol \+

```
?- parent(sam,X), \+female(X).
X = john.
```

```
child(john,sue).
child(john,sam).
child(jane,sue).
child(jane,sam).
child(sue,george).
child(sue,gina).

male(john).
male(sam).
male(george).

female(sue).
female(jane).
female(june).

?- male(gina): false.
?- \+female(gina): true.
```

Trace the Inference

```
?- trace.
true.

[trace]  ?- parent(sam,X), \+male(X).
   Call: (9) parent(sam, _4254) ? creep
   Call: (10) child(_4254, sam) ? creep
   Exit: (10) child(john, sam) ? creep
   Exit: (9) parent(sam, john) ? creep
   Call: (9) male(john) ? creep
   Exit: (9) male(john) ? creep
   Redo: (10) child(_4254, sam) ? creep
   Exit: (10) child(jane, sam) ? creep
   Exit: (9) parent(sam, jane) ? creep
   Call: (9) male(jane) ? creep
   Fail: (9) male(jane) ? creep
X = jane.
```

Another Example

```
criminal(X) :- american(X), weapon(Y), sells(X, Y, Z), hostile(Z).
sells(west, X, nono) :- missle(X), owns(nono, X).
weapon(X) :- missle(X).
hostile(X) :- enemy(X, america).
owns(nono, m1).
missle(m1).
american(west).
enemy(nono, america).
```

Trace the Inference

```
[trace]  ?- criminal(X).
   Call: (8) criminal(_2934) ? creep
   Call: (9) american(_2934) ? creep
   Exit: (9) american(west) ? creep
   Call: (9) weapon(_3148) ? creep
   Call: (10) missle(_3148) ? creep
   Exit: (10) missle(m1) ? creep
   Exit: (9) weapon(m1) ? creep
   Call: (9) sells(west, m1, _3152) ? creep
   Call: (10) missle(m1) ? creep
   Exit: (10) missle(m1) ? creep
   Call: (10) owns(nono, m1) ? creep
   Exit: (10) owns(nono, m1) ? creep
   Exit: (9) sells(west, m1, nono) ? creep
   Call: (9) hostile(nono) ? creep
   Call: (10) enemy(nono, america) ? creep
   Exit: (10) enemy(nono, america) ? creep
   Exit: (9) hostile(nono) ? creep
   Exit: (8) criminal(west) ? creep
X = west.
```

Instantiated Variables and Negation

- Prolog outputs different results given similar inputs.

```
?- parent(X,john), \+female(X).
X = sam.

?- \+female(X), parent(X,john).
false.
```

Search female(X) at the first, and female(X) is true, so \+female(X) is resulted in false.

```
[trace]  ?- parent(X,john), \+female(X).
   Call: (9) parent(_3436, john) ? creep
   Call: (10) child(john, _3436) ? creep
   Exit: (10) child(john, sue) ? creep
   Exit: (9) parent(sue, john) ? creep
   Call: (9) female(sue) ? creep
   Exit: (9) female(sue) ? creep
   Redo: (10) child(john, _3436) ? creep
   Exit: (10) child(john, sam) ? creep
   Exit: (9) parent(sam, john) ? creep
   Call: (9) female(sam) ? creep
   Fail: (9) female(sam) ? creep
X = sam.
```

```
[trace]  ?- \+female(X), parent(X,john).
   Call: (9) female(_3258) ? creep
   Exit: (9) female(sue) ? creep
false.
```

Issue of Uninstantiated Variables and Negation

- The issue of this case is that the variable X is **uninstantiated** when the negated portion of the query is handled.
- Instantiated (實例化)
    - A variable has a tentative value.
- When variables appear in a negated query, make sure that they are already instantiated at an earlier stage of the back-chaining.

```
[trace]  ?- \+female(X), parent(X,john).
   Call: (9) female(_3258) ? creep
   Exit: (9) female(sue) ? creep
false.
```

Equality in Queries

- Prolog allows elements in a query of the form
  - $term_1 = term_2$
- Where the 2 terms are either constants or variables.

```
?- X=sam, X=Y.
X = sam
Y = sam

?- X=Y, sam=X, \+ Y=jack.
X = sam
Y = sam

?- X=Y, sam=X, \+ Y=sam.
false.
```

Inequality

- In fact, it is never necessary to use unnegated equality in a query.

```
?- child(john,X), child(jane,Y), X=Y.

?- child(john,X), child(jane,X).
```

- By contrast, negated equality, inequality, is more useful in queries.

Queries Using Negated Equality

```
?- parent(sam,X), \+X=john.
X = jane.

?- male(X), male(Y), male(Z).
X = Y, Y = Z, Z = john .

?- male(X), male(Y), male(Z), \+ X=Y, \+ X=Z, \+ Y=Z.
X = john,
Y = sam,
Z = george ;
```

Overview of the Prolog Syntax

- A **constant** is either a single-quoted string or a lowercase letter followed by zero or more letters, digits, and underscores.
- A **variable** is an uppercase letter or an underscore followed by zero or more letters, digits, and underscores.
- A **number** is a sequence of digits, optionally preceded by a minus sign and optionally containing a decimal point.
- A **term** is a constant, variable, or number.

Overview of the Prolog Syntax

- A **predicate** is written as a constant.
- An **atomic** sentence (atom) is a predicate optionally followed by terms (arguments).
- An **equality** is two terms connected by =
- A **literal** is an atom or an equality optionally preceded by **\+** for **negation**.

Overview of the Prolog Syntax

- A **query** is a sequence of one or more literals separated by commas and terminated by a period.
- A **clause** is an atom (head) followed by a period or by the :- symbol and then a query (body).
- A **program** is a sequence of one or more clauses.

```
has_two_children(X) :- child(Y, X), child(Z, X), \+ Y = Z
```

Back-chaining in Prolog

- Prolog finds solution with the back-chaining algorithm.

```
likes(john,pizza).
likes(john,sushi).
likes(mary,sushi).
likes(paul,X) :- likes(john,X).
likes(X,ice_cream).
```

Backward Chaining

- These algorithm work backward from the goal (query), chaining through rules to find known facts that support proof.
  - Goal oriented
  - But there are some disadvantages compared with forward chaining.

FOL Backward Chaining

function FOL-BC-ASK(KB, $query$) **returns** a generator of substitutions
  return FOL-BC-OR(KB, $query$, { })

---

generator FOL-BC-OR(KB, $goal$, θ) **yields** a substitution
  for each rule ($lhs \Rightarrow rhs$) in FETCH-RULES-FOR-GOAL(KB, $goal$) **do**
    (lhs, rhs) $\leftarrow$ STANDARDIZE-VARIABLES((lhs, rhs))
    for each θ' **in** FOL-BC-AND(KB, lhs, UNIFY(rhs, $goal$, θ)) **do**
      yield θ'

---

generator FOL-BC-AND(KB, $goals$, θ) **yields** a substitution
  if $\theta = failure$ **then return**
  else if LENGTH($goals$) = 0 **then yield** θ
  else do
    $first$,$rest$ $\leftarrow$ FIRST($goals$), REST($goals$)
    for each θ' **in** FOL-BC-OR(KB, SUBST(θ, $first$), θ) **do**
      for each θ'' **in** FOL-BC-AND(KB, $rest$, θ') **do**
        yield θ''

Backward Chaining Algorithm

- We want to prove if the KB contains a clause of the form *lhs* → *goal*.
  - lhs (left hand side) is a list of conjuncts.
- And/or Search
  - Or part: The goal query can be proved by any rule in the KB
  - And part: All the conjuncts in the *lhs* should be proved.

OR Part

- Fetching all clauses that might unify with the goal
- Standardizing the variables to be new variables
- If the *rhs* does unify with the goal, proving every conjuncts in the *lhs*

generator FOL-BC-OR($KB, goal, \theta$) **yields** a substitution
  for each rule ($lhs \ \Rightarrow \ rhs$) in FETCH-RULES-FOR-GOAL(KB, $goal$) **do**
    (lhs, rhs) $\leftarrow$ STANDARDIZE-VARIABLES((lhs, rhs))
    for each θ' **in** FOL-BC-AND(KB, lhs, UNIFY(rhs, $goal$, θ)) **do**
      yield θ'

Backward-chaining Algorithm

```
function PL-BC-Entails(KB, Q):

    if Q in KB:
       return True
    for each Horn clause c in KB:
       if head[c] == Q:
           T = True
           for each premise P in body[c]:
               if PL-BC-Entails(KB, P) is False:
                   T = False
           if T == True:
               return True
    return False
```

And Part

- And part is aimed at proving each of the conjuncts in turn.
- Checking the first conjunct, and recursively check the rest ones.

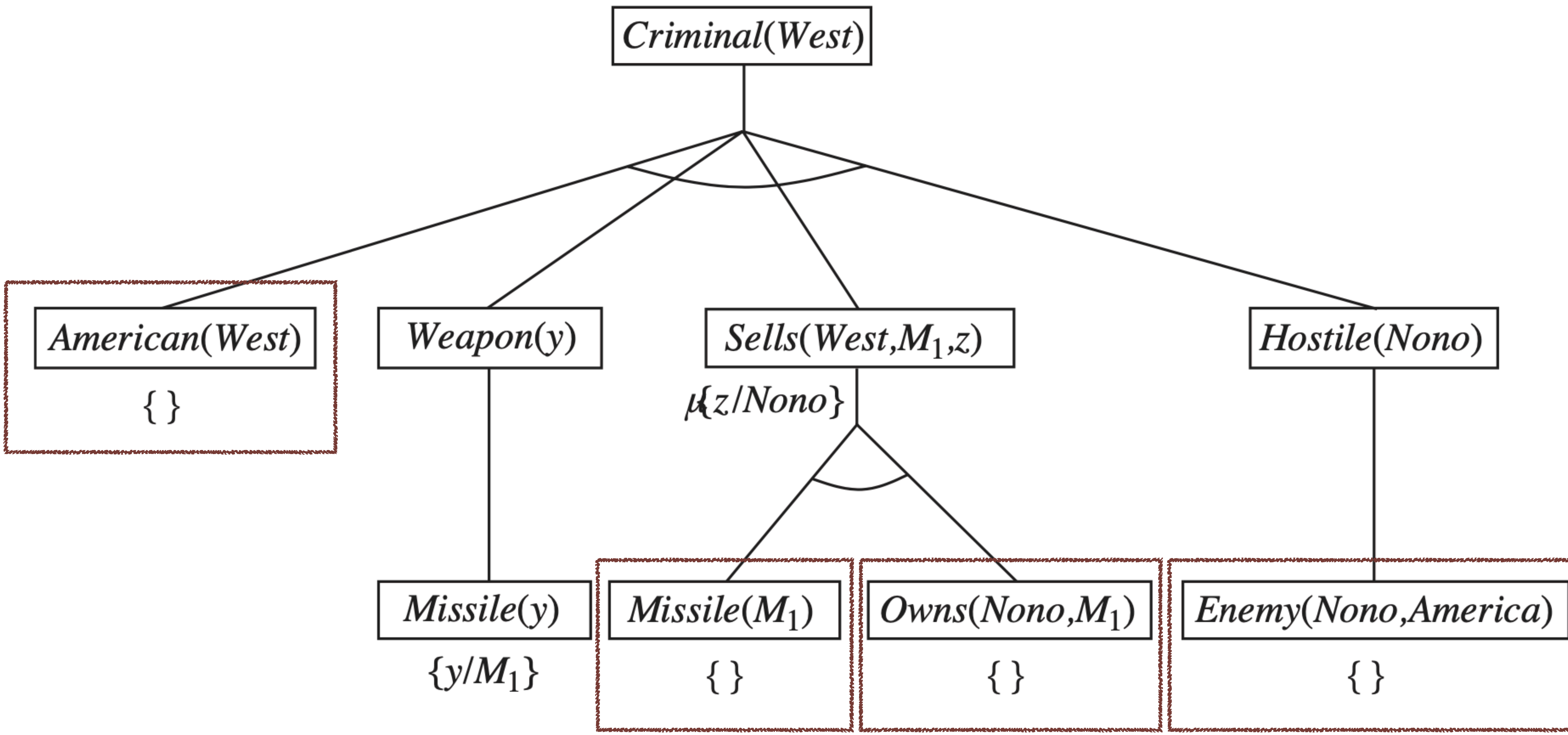
generator FOL-BC-AND(KB, $goals$, θ) **yields** a substitution
  if $\theta = failure$ **then return**
  else if LENGTH($goals$) = 0 **then yield** θ
  else do
    $first$, $rest$ $\leftarrow$ FIRST($goals$), REST($goals$)
    for each θ' **in** FOL-BC-OR(KB, SUBST(θ, $first$), θ) **do**
      for each θ'' **in** FOL-BC-AND(KB, $rest$, θ') **do**
        yield θ''

Backward-chaining Algorithm

```
function PL-BC-Entails(KB, Q):

    if Q in KB:
       return True
    for each Horn clause c in KB:
       if head[c] == Q:
           T = True
           for each premise P in body[c]:
               if PL-BC-Entails(KB, P) is False:
                   T = False
           if T == True:
               return True
    return False
```

Proof Tree of FOL Backward Chaining



Unification

- Lifted inference rules require finding substitutions that makes different logical expressions look identical.

- Unification is to find the substitutions.

    - A key component of all first-order inference algorithms.

- Providing a good heuristic to prune the infeasible solutions in the search space.

Unify Algorithm

- Given two sentences, the unify algorithm returns a unifier for them if one exists.
  - Unify(p, q) = θ where Subst(θ, p) = Subst(θ, q)

To answer *AskVars*(*Knows*(*John*, *x*))

Unify(*Knows*(*John*, *x*), ***Knows*(*John*, *Jane*)**) = {*x*/*Jane*}
Unify(*Knows*(*John*, *x*), ***Knows*(*y*, *Bill***) = {*x*/*Bill*, *y*/*John*}
Unify(*Knows*(*John*, *x*), ***Knows*(*y*, Mother(y))**) = {*y*/*John*, *x*/*Mother*(*John*)}

Knows(*John*, *Mother*(*John*)) = *Knows*(*John*, *Mother*(*John*))

Unify Algorithm

- Given two sentences, the unify algorithm returns a unifier for them if one exists.

  - Unify(p, q) = θ where Subst(θ, p) = Subst(θ, q)

To answer *AskVars*(*Knows*(*John*, *x*))

Unify(*Knows*(*John*, *x*), *Knows*(*John*, *Jane*)) = {*x*/*Jane*}
Unify(*Knows*(*John*, *x*), *Knows*(*y*, *Bill*) = {*x/Bill, y/John*}
Unify(*Knows*(*John*, *x*), *Knows*(*y*, Mother(y))) = {*y/John, x/Mother(John)*}
Unify(*Knows*(*John*, *x*), ***Knows*(*x*, *Elizabeth*)**) = ***fail***

Fails because x cannot take on the values John and Elizabeth at the same time
However, it should not fail because *Knows*(*x*, *Elizabeth*) means that Everyone knows Elizabeth

Standardizing Apart

- Two sentences may use the same variable name like *x*.
- To avoid the situation, standardizing apart one of the sentences being unified.
  - Rename *x* in *Knows*(*x*, *Elizabeth*) to a new one like x_2

To answer *AskVars*(*Knows*(*John*, *x*))

Unify(*Knows*(*John*, *x*), *Knows*(*John*, *Jane*)) = {*x*/*Jane*}
Unify(*Knows*(*John*, *x*), *Knows*(*y*, *Bill*) = {*x/Bill, y/John*}
Unify(*Knows*(*John*, *x*), *Knows*(*y*, Mother(y))) = {*y/John, x/Mother(John)*}
Unify(*Knows*(*John*, *x*), ***Knows*(*x_2*, *Elizabeth*)**) = **{*x*/*Elizabeth*, *x_2*/*John*}**

For More Than One Quantifier

- A complication situation raises more than one substitutions can be made.

To answer *AskVars*(*Knows*(*John*, *x*))

Unify(*Knows(John, x), Knows(y, z)*)

- {*y/John, x/z*}
  - *Knows*(*John*, *z*) = *Knows*(*John*, *z*)
- {*y/John, x/John, z/John*}:
  - *Knows*(*John*, *John*) = *Knows*(*John*, *John*)

Which one is better?

General Unifier

- $\{y/John, x/z\}$
  - $Knows(John, z) = Knows(John, z)$
  - More general because it places fewer restrictions on the variables.
- $\{y/John, x/John, z/John\}$:
  - $Knows(John, John) = Knows(John, John)$
- There is a single most general unifier (MGU) that is unique up to renaming and substitution of variables.

Finding Most General Unifier

- Recursively explore the two expressions simultaneously side by side.
- Building up a unifier along the way
- Failing if two corresponding points in the structures do not match.

Occur Check

- When matching a variable against a complex term, one must check whether the variable itself occurs inside the term.
    - The match fails because no consistent unifier can be constructed for the recursive relation.
        - **S(x)** cannot unify with S(**S(x)**)
    - Occur check is expensive that makes the time complexity is $O(n^4)$ where n is number of expressions being unified.
- Some system does not perform occur check and can make unsound inferences.
- Some system use more complex algorithm with linear time complexity.

function UNIFY(x, y, θ) **returns** a substitution to make x and y identical
inputs: x, a variable, constant, list, or compound expression
y, a variable, constant, list, or compound expression
θ, the substitution built up so far (optional, defaults to empty)

if θ = failure **then return** failure
else if $x = y$ **then return** θ
else if VARIABLE?(x) **then return** UNIFY-VAR(x, y, θ)
else if VARIABLE?(y) **then return** UNIFY-VAR(y, x, θ)
else if COMPOUND?(x) **and** COMPOUND?(y) **then**
  return UNIFY(x.ARGS, y.ARGS, UNIFY(x.OP, y.OP, θ))
else if LIST?(x) **and** LIST?(y) **then**
  return UNIFY(x.REST, y.REST, UNIFY(x.FIRST, y.FIRST, θ))
else return failure

$\underbrace{\text{Knows(}}_{\text{Op}}\ \underbrace{\text{x, y}}_{\text{Arg List}}\text{)}$

---

function UNIFY-VAR(var, x, θ) **returns** a substitution

if $\{var/val\} \in \theta$ **then return** UNIFY(val, x, θ)
else if $\{x/val\} \in \theta$ **then return** UNIFY(var, val, θ)
else if OCCUR-CHECK?(var, x) **then return** failure
else return add $\{var/x\}$ to θ

Unification in Prolog

- Clauses in a program are selected during back-chaining through the matching process **unification**.

- Two atomic sentences whose variables are distinct are said to unify if there is a substitution of values for the variables that makes the atomic sentences identical.

```
likes(john,pizza).
likes(john,sushi).
likes(mary,sushi).
likes(paul,X) :- likes(john,X).
likes(X,ice_cream).

?- likes(john,Y).
```

Unification in Prolog

```
likes(john,pizza).
likes(john,sushi).
likes(mary,sushi).
likes(paul,X) :- likes(john,X).
likes(X,ice_cream).

?- likes(paul,Y).
```

```
likes(john,pizza).
likes(john,sushi).
likes(mary,sushi).
likes(paul,X) :- likes(john,X).
likes(X,ice_cream).

?- likes(jane,Y).
```

Attempt to Unify

```
p(b,X,b) ↔ p(Y,a,b)

p(X,b,X) ↔ p(a,b,Y)

p(b,X,b) ↔ p(Y,Z,b)

p(X,Z,X,Z) ↔ p(Y,W,a,Y)
```

Not to Unify

p(b,X,b) ↔ p(b,Y)

p(b,X,**b**) ↔ p(Y,a,**a**)   ~~a=b~~

p(**X**,b,**X**) ↔ p(a,a,b)   ~~X=b, X=a~~

p(X,b,X,a) ↔ p(Y,Z,Z,Y)   ~~X=Y=Z,Y=a,Z=b~~

Renaming Variables

- While the unification process is not concerned with where the atomic sentences come from query or program during back-chaining.

- Prolog renames the variables in the query before attempting unification to ensure there are no clashes.

```
likes(john,pizza).
likes(john,sushi).
likes(mary,sushi).
likes(paul,X) :- likes(john,X).
likes(X,ice_cream).
?- likes(X,pizza), \+ X=john.
```

```
[trace]  ?- likes(X,pizza), \+ X=john.
   Call: (9) likes(_2956, pizza) ? creep
   Exit: (9) likes(john, pizza) ? creep
   Call: (9) john=john ? creep
   Exit: (9) john=john ? creep
   Redo: (9) likes(_2956, pizza) ? creep
   Call: (10) likes(john, pizza) ? creep
   Exit: (10) likes(john, pizza) ? creep
   Exit: (9) likes(paul, pizza) ? creep
   Call: (9) paul=john ? creep
   Fail: (9) paul=john ? creep
X = paul
```

Rename X in query as _2956

No unification will be made for both likes(paul,X) and likes(X,pizza)

Back-chaining Procedure in Prolog

```
BC(Q: A_1, A_2, ..., A_n):
    if n == 0:
        return True
    for each clause (H, B_1, ..., B_m) in the program:
        if H unifies A_1:
            if all BC(B'_1), ..., BC(B'_m) are True:
                if all BC(A'_2), ..., BC(A'_n) are True:
                    return True
    return False
```

Lifted

Prolog Systems

- Modern systems follows ISO Standard Prolog
  - SWI-Prolog
  - GNU Prolog
  - SICStus Prolog
  - ECLiPSe

Variances in Dialect

- Negation
  - +\ or other symbols such as **not**
- Comment
  - % or the C style **/* ... */**
- Built-in procedures such as loading files, tracing, etc.
- Dynamic predicates
  - If a predicate in the query that does not appear in the program, the query will get false rather than trigger an error.