Intelligent Systems Principles and Programming

Xiaoqing Zheng

zhengxq@fudan.edu.cn

Foundation of knowledge representation

- **Logics**
- **Ontology**
- **Theory of Computation**

Propositional logic

Propositional calculus symbols

- Propositional symbols

  P, *Q*, *R*, *S*, ...
- Truth symbols

  true, *false*
- Connectives

  $\wedge$ conjunction

  $\vee$ disjunction

  $\neg$ not

  $\rightarrow$ implication

  $\equiv$ equivalence

Propositional logic

Propositional calculus sentences

- Every propositional symbol and truth symbol is sentences.

  $true$, P, Q

- The ***negation*** of a sentence is a sentence.

  $\neg false$, $\neg P$

- The ***conjunction***, or ***and***, of two sentences is a sentence.

  $P \wedge \neg P$

- The ***disjunction***, or ***or***, of two sentences is a sentence.

  $P \vee \neg P$

Propositional logic

Propositional calculus sentences

- The implication of one sentence from another is a sentence.

  $P \rightarrow Q$

- The ***equivalence*** of two sentences is a sentence.

  $P \vee Q \equiv R$

Legal sentences are also called ***well-formed formulas*** or ***WFFs***.

Propositional logic

Propositional calculus semantics

The ***interpretation*** of a set of propositions is the assignment of a truth value, either *true* or *false*, to each propositional symbol.

- The truth assignment of ***negation***, $\neg P$, where P is any propositional symbol, is *false* if the assignment to P is *true*, a *true* if the assignment to P is *false*.
- The truth assignment of ***conjunction***, $\wedge$ is *true* only when both conjuncts have *true* value; otherwise it is *false*.
- The truth assignment of ***disjunction***, $\vee$ is *false* only when both disjuncts have *false* value; otherwise it is *true*.

Propositional logic

Propositional calculus semantics

The ***interpretation*** of a set of propositions is the assignment of a truth value, either truth or false, to each propositional symbol.

- The truth assignment of ***implication***, →, is *false* only when the premise or symbol before the implication is *true* and the truth value of the consequent or symbol after the implication is *false*; otherwise it is *true*.
- The truth assignment of ***equivalence***, ≡, is *true* only when both expressions have the same truth assignment for all possible interpretations; otherwise it is *false*.

Presumption

It has rained $\longrightarrow$ Ground is wet

Truth table

It has rained	*Ground is wet*	*Formula*
false	*false*	*true*
false	*true*	*true*
true	*false*	*false*
true	*true*	*true*

Propositional logic

Equivalences

- $\neg(\neg P) \equiv P$
- $(P \vee Q) \equiv (\neg P \rightarrow Q)$
- $(P \rightarrow Q) \equiv (\neg Q \rightarrow \neg P)$ (The ***contrapositive*** law)
- $\neg(P \vee Q) \equiv (\neg P \wedge \neg Q)$ (The ***Morgan's*** law)
  $\neg(P \wedge Q) \equiv (\neg P \vee \neg Q)$
- $(P \vee Q) \equiv (Q \vee P)$ (The ***commutative*** laws)
  $(P \wedge Q) \equiv (Q \wedge P)$
- $((P \vee Q) \vee R)) \equiv (P \vee (Q \vee R))$ (The ***associative*** laws)
  $((P \wedge Q) \wedge R)) \equiv (P \wedge (Q \wedge R))$
- $P \vee (Q \wedge R) \equiv (P \vee Q) \wedge (P \vee R)$ (The ***distributive*** laws)
  $P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge R)$

Proof

$(\neg P \vee Q) \equiv (P \rightarrow Q)$

Truth table

P	Q	$\neg P$	$\neg P \vee Q$	$P \rightarrow Q$	$(\neg P \vee Q) \equiv (P \rightarrow Q)$
true	*true*	*false*	*true*	*true*	*true*
true	*false*	*false*	*false*	*false*	*true*
false	*true*	*true*	*true*	*true*	*true*
false	*false*	*true*	*true*	*true*	*true*

Syllogism

All men are mortal

he is a man $\longrightarrow$ he will die

Socrates is man

Socrates is a man

Will Socrates die?

First order predicate calculus

$$\left.\begin{array}{l}\forall x(man(x) \rightarrow mortal(x)) \\ man(Socrates)\end{array}\right\} \Longrightarrow mortal(Socrates)$$

First-order logic

Symbols and terms

- ***Truth symbols*** *true* and *false*.
- ***Constant symbols*** are symbol expressions having the first character ***lowercase***.
- ***Variable symbols*** are symbol expressions beginning with an ***lowercase*** character.
- ***Function symbols*** are symbol expressions having the first character ***lowercase***. Functions have an attached ***arity*** indicating the number of elements of the domain mapped onto each element of the range.

First-order logic

Predicates and atomic sentences

- ***Predicate symbols*** are symbols beginning with a ***lowercase*** letter. Predicates have an associated non-negative integer referred to as the ***arity*** or ***argument number*** for the predicates.
- An ***atomic sentence*** is a predicate constant of arity n, followed by n terms, t_1, t_2, ..., t_n, enclosed in parentheses and separated by commas.
- The truth values, *true* and *false*, are atomic sentences.

First-order logic

Logical connectives

$\wedge$	conjunction
$\vee$	disjunction
$\neg$	not
$\rightarrow$	implication
$\equiv$	equivalence
$\exists$	existential quantifier
$\forall$	universal quantifier

First-order logic

Predicate calculus sentences

- Every ***atomic*** sentence is a sentence.
- If s is a sentence, then so is its ***negation***, $\neg s$.
- If s_1 and s_2 is a sentence, then so is their ***conjunction***, $s_1 \wedge s_2$.
- If s_1 and s_2 is a sentence, then so is their ***disjunction***, $s_1 \vee s_2$.
- If s_1 and s_2 is a sentence, then so is their ***implication***, $s_1 \rightarrow s_2$.
- If s_1 and s_2 is a sentence, then so is their ***equivalence***, $s_1 \equiv s_2$.
- If x is a variable and s is a sentence, then $\forall x\ s$, is a sentence.
- If x is a variable and s is a sentence, then $\exists x\ s$, is a sentence.

Family relationships

$mother(eve, abel)$

$mother(eve, cain)$

$father(adam, abel)$

$father(adam, cain)$

$$\forall x \forall y ((father(x,y) \vee mother(x,y) \rightarrow parent(x,y))$$
$$\forall x \forall y \forall z ((parent(x,y) \wedge parent(x,z) \rightarrow sibling(y,z))$$

Intuitively, it is clear that these implications can be used to infer facts such as

$sibling(cain, abel)$

First-order logic

Interpretation

Let the domain **D** be a nonempty set. An ***interpretation*** over **D** is an assignment of the entities of **D** to each of the constant, variable, predicate, and function symbols of a predicate calculus expression, such that:

- Each ***constant*** is assigned an ***element*** of **D**.
- Each ***variable*** is assigned to a ***nonempty subset*** of **D**; these are the allowable substitutions for that variable.
- Each ***function*** f of arity m is defined on m arguments of **D** and defines a mapping from $\mathbf{D}^m$ into **D**.
- Each ***predicate*** p of arity n is defined on n arguments of **D** and defines a mapping from $\mathbf{D}^n$ into $\{true, false\}$.

First-order logic

Truth value of predicate calculus expressions

Assume an expression *E* and an interpretation **I** for *E* over a nonempty domain **D**. The truth value for *E* is determined by:

- The value of a ***constant*** is the element of **D** it is assigned to by **I**.
- The value of a ***variable*** is the set of elements of **D** it is assigned to by **I**.
- The value of a ***function*** expression is that element of **D** obtained by evaluating the function for the parameter values assigned by the interpretation **I**.
- The value of an ***atomic sentence*** is either *true* or *false*, as determined by the interpretation **I**.

First-order logic

Truth value of predicate calculus expressions

Assume an expression *E* and an interpretation **I** for *E* over a nonempty domain **D**. The truth value for *E* is determined by:

- The value of expressions using $\neg$, $\wedge$, $\vee$, $\rightarrow$, and $\equiv$ is determined from the value of their operands.
- The value of $\forall x\ s$ is *true* if *s* is *true* for all assignments to *x* under **I**, and it is *false* otherwise.
- The value of $\exists x\ s$ is *true* if there is an assignment to *x* in the interpretation **I** under which *s* is *true*; otherwise it is *false*.

First-order logic

First-order predicate calculus

- First-order predicate calculus allows ***quantified variables*** to refer to objects in the domain of discourse and not to ***predicates*** or ***functions***.

First-order logic

First-order predicate calculus

- First-order predicate calculus allows ***quantified variables*** to refer to objects in the domain of discourse and not to ***predicates*** or ***functions***.

$$\forall(likes)likes(george, kate)$$

Higher-order predicate calculi

Examples

- If it does not rain on Monday, Tom will go to the mountains.

  $\neg weather(rain, monday) \rightarrow go(tom, mountains)$
- Emma is a Doberman pinscher and a good dog.

  $gooddog(emma) \wedge isa(emma, doberman)$
- All basketball player are tall.

  $\forall x(basketball\_player(x) \rightarrow tall(x))$
- Some people like anchovies.

  $\exists x(person(x) \wedge likes(x, anchovies))$
- Nobody likes taxes.

  $\neg \exists x(likes(x, taxes))$

Satisfiable, Model, and Inference

1. $\forall x(A(x) \rightarrow B(x))$

2. $\forall x(B(x) \rightarrow C(x))$

Interpretation 1: $A(a)$, $A(b)$, $B(a)$, $\neg B(b)$, $\neg C(a)$,$\neg C(b)$

Interpretation 2: $A(a)$, $A(b)$, $B(a)$, $\neg C(b)$,

Interpretation 3: $A(a)$, $B(a)$, $C(a)$

Inference

- The ability to ***infer*** new correct expressions from a set of true assertions is an important feature of the predicate calculus.
- These new expressions are correct in that they are ***consistent with*** all previous interpretations of the original set of expressions.
- An expression *x* ***logically follows*** from a set of predicate calculus expressions ***S*** if every interpretation that ***satisfies*** *S* also ***satisfies*** *x*.
- Inference rules produce new sentences based on the ***syntactic form*** of given logical assertions.

What is the meaning of satisfies?

First-order logic

Satisfy, model, valid, inconsistent

For a predicate calculus expression *x* and an interpretation **I**:

- If *x* has a ***value*** of *true* under **I** and a particular variable assignment, then **I** is said to ***satisfy*** *x*.
- If **I** satisfies *x* for all variable assignments, then **I** is a ***model*** of *x*.
- *x* is ***satisfiable*** if and only if there exist an interpretation and variable assignment that satisfy it; otherwise, it is ***unsatisfiable***.

First-order logic

Satisfy, model, valid, inconsistent

For a predicate calculus expression *x* and an interpretation **I**:

- A set of expressions is ***satisfiable*** if and only if there exist an interpretation and variable assignment that satisfy every element.
- If a set of expressions is not satisfiable, it is said to be ***inconsistent***.
- If *x* has a value *true* for all possible interpretations, *x* is said to be ***valid***.

First-order logic

Satisfy, model, valid, inconsistent

For a predicate calculus expression x and an interpretation **I**:

- A set of expressions is ***satisfiable*** if and only if there exist an interpretation and variable assignment that satisfy every element.
- If a set of expressions is not satisfiable, it is said to be ***inconsistent***.
- If x has a value *true* for all possible interpretations, x is said to be ***valid***.

Inconsistent: $\exists x\ (P(x) \land \neg P(x))$

Valid: $(P(x) \lor \neg P(x))$

Proof procedure

A ***proof procedure*** is a combination of an inference rule and an algorithm for applying that rule to a set of logical expressions to generate new sentences.

Logical follows, sound, and complete

- A predicate calculus expression x ***logically follows*** from a set ***S*** of predicate calculus expression if every interpretation and variable assignment that satisfies **S** also satisfies x.
- An inference rule is ***sound*** if every predicate calculus expression produced by the rule from a set of ***S*** of predicate calculus expressions also logically follows from ***S***.
- An inference rule is ***complete*** if, given a set ***S*** of predicate calculus expressions, the rule can infer every expression that logically follows from ***S***.

Rules

Modus ponens, modus tollens, elimination

- If the sentences P and $P \rightarrow Q$ are known to be *true*, then ***modus ponens*** let us infer Q.
- Under the inference rule ***modus tollens***, if $P \rightarrow Q$ are known to be *true* and Q is known to be *false*, we can infer $\neg P$.
- ***Elimination*** allows us to infer the truth of either of the conjuncts from the truth of a conjunctive sentence. For instance, $P \wedge Q$ let us conclude P and Q are *true*.

Rules

Introduction, universal instantiation

- ***Introduction*** lets us infer the truth of its conjuncts. For instance, if P and Q are *true*, then a conjunction from the truth of its conjuncts. $P \wedge Q$.
- ***Universal instantiation全称实例化*** states that if any universally quantified variable in a *true* sentence is replaced by any appropriate term from the domain, the result is a *true* sentence. Thus, if a is from the domain of x, $\forall x\, P(x)$ lets us infer $P(a)$.

Example

$$\forall x(man(x) \rightarrow mortal(x))$$

$$man(Socrates)$$

Example

$\forall x(man(x) \rightarrow mortal(x))$

$man(Socrates)$

Unification { Socrates/x }

$man(x) \rightarrow mortal(x)$

$man(Socrates) \rightarrow mortal(Socrates)$ **(Universal instantiation)**

Example

$\forall x(man(x) \rightarrow mortal(x))$

$man(Socrates)$

***Unification* 合一{ *Socrates/x* }**

$man(x) \rightarrow mortal(x)$

$man(Socrates) \rightarrow mortal(Socrates)$ **(Universal instantiation)**

$man(Socrates)$

$mortal(Socrates)$ **(Modus ponens)**

Unification

Most general unifier (mgu)(更一般合一性)

If s is any unifier of expressions E and g is the ***most general unifier*** of that set of expressions, then for s applied to E there exists another unifier s' such that $Es = Egs'$, where Es and Egs' are the ***composition*** of unifiers applied to the expression E.

- Any constant is considered a ***ground instance*** and may not be replaced. Neither can two different ground instances be substituted for one variable.
- A variable cannot be unified with a term containing that variable.

Unification

Most general unifier (最一般合一式(mgu)

If s is any unifier of expressions E and g is the ***most general unifier*** of that set of expressions, then for s applied to E there exists another unifier s' such that $Es = Egs'$, where Es and Egs' are the ***composition*** of unifiers applied to the expression E.

- It is important that any unifying substitution be made ***consistently*** across all occurrences of the variable in both expressions being matched.
- The unifier must be as ***general*** as possible: that the most general unifier be found for the two expressions.

Unification example

$$\mathit{parents}(\underbrace{x}_{\uparrow}, \mathit{father}(x), \mathit{mother}(\mathit{bill})) \qquad \mathit{parents}(\underbrace{\mathit{bill}}_{\uparrow}, \mathit{father}(\mathit{bill}), y)$$

$$\Big\Downarrow \ \{\ \textbf{\textit{bill/x}}\ \}$$

$$\mathit{parents}(\mathit{bill}, \mathit{father}(\mathit{bill}), \underbrace{\mathit{mother}(\mathit{bill})}_{\uparrow})$$

$$\mathit{parents}(\mathit{bill}, \mathit{father}(\mathit{bill}), \underbrace{y}_{\uparrow})$$

$$\{\ \textbf{\textit{mother(bill)/y}}\ \} \ \Big\Downarrow$$

$$\mathit{parents}(\mathit{bill}, \mathit{father}(\mathit{bill}), \mathit{mother}(\mathit{bill}))$$

mgu: {*bill/x*, *mother*(*bill*)/*y* }

A logic-based financial advisor

The function of the advisor is to help a user decider whether to invest in a savings account or the stock market.

- Individuals with an ***inadequate savings account*** should always make increasing the amount saved their first priority.
- Individuals with an ***adequate savings account*** and an ***adequate income*** should consider a riskier but potentially more profitable investment in the stock market.
- Individuals with a ***lower income*** who already have an ***adequate savings account*** may want to consider splitting their surplus income between savings and stocks.

A logic-based financial advisor

[1] $savings\_account(inadequate) \rightarrow investment(savings)$

[2] $savings\_account(adequate) \wedge income(adequate)$
$\rightarrow investment(stocks)$

[3] $savings\_account(adequate) \wedge income(inadequate)$
$\rightarrow investment(combinaton)$

A logic-based financial advisor

[4] $\forall x(amount\_saved(x) \wedge \exists y(dependents(y) \wedge greater(x, minsavings(y))) \rightarrow savings\_account(adequate))$

[5] $\forall x(amount\_saved(x) \wedge \exists y(dependents(y) \wedge \neg greater(x, minsavings(y))) \rightarrow savings\_account(inadequate))$

[6] $\forall x(earnings(x, steady) \wedge \exists y(dependents(y) \wedge greater(x, minincome(y))) \rightarrow income(adequate))$

[7] $\forall x(earnings(x, steady) \wedge \exists y(dependents(y) \wedge \neg greater(x, minincome(y))) \rightarrow income(inadequate))$

[8] $\forall x(earnings(x, unsteady) \rightarrow income(inadequate))$

A logic-based financial advisor

[9] $amount\_saved(22000)$

[10] $earnings(25000, steady)$

[11] $dependents(3)$

Functions:

$$minsavings(x) = 5000 \times x$$

$$minincome(x) = 15000 + (4000 \times x)$$

A logic-based financial advisor

[10] $earnings(25000, steady)$

[11] $dependents(3)$

[7] $\forall x(earnings(x, steady) \wedge \exists y(dependents(y) \wedge \neg greater(x, minincome(y))) \rightarrow income(inadequate))$

{ 25000/*x*, 3/*y* }

A logic-based financial advisor

[10] $earnings(25000, steady)$

[11] $dependents(3)$

[7] $\forall x(earnings(x, steady) \wedge \exists y(dependents(y) \wedge$
$\neg greater(x, minincome(y))) \rightarrow income(inadequate))$

{ 25000/*x*, 3/*y* }

[12] $earnings(25000, steady) \wedge dependents(3) \wedge$
$\neg greater(25000, minincome(3)) \rightarrow income(inadequate)$

A logic-based financial advisor

[10] $earnings(25000, steady)$

[11] $dependents(3)$

[7] $\forall x(earnings(x, steady) \wedge \exists y(dependents(y) \wedge \neg greater(x, minincome(y))) \rightarrow income(inadequate))$

{ 25000/*x*, 3/*y* }

[12] $earnings(25000, steady) \wedge dependents(3) \wedge \neg greater(25000, minincome(3)) \rightarrow income(inadequate)$

[13] $income(inadequate)$

A logic-based financial advisor

[9] $amount\_saved(22000)$

[11] $dependents(3)$

[4] $\forall x(amount\_saved(x) \land \exists y(dependents(y) \land greater(x, minsavings(y))) \rightarrow savings\_account(adequate))$

{ 22000/*x*, 3/*y* }

A logic-based financial advisor

[9] $amount\_saved(22000)$

[11] $dependents(3)$

[4] $\forall x(amount\_saved(x) \wedge \exists y(dependents(y) \wedge greater(x, minsavings(y))) \rightarrow savings\_account(adequate))$

{ 22000/*x*, 3/*y* }

[14] $amount\_saved(22000) \wedge dependents(3) \wedge greater(22000, minsavings(3)) \rightarrow savings\_account(adequate)$

A logic-based financial advisor

[9] $amount\_saved(22000)$

[11] $dependents(3)$

[4] $\forall x(amount\_saved(x) \wedge \exists y(dependents(y) \wedge greater(x, minsavings(y))) \rightarrow savings\_account(adequate))$

{ 22000/*x*, 3/*y* }

[14] $amount\_saved(22000) \wedge dependents(3) \wedge greater(22000, minsavings(3)) \rightarrow savings\_account(adequate)$

[15] $savings\_account(adequate)$

A logic-based financial advisor

[13] $income(inadequate)$

[15] $savings\_account(adequate)$

[3] $savings\_account(adequate) \wedge income(inadequate)$
$\rightarrow investment(combinaton)$

$$\Downarrow$$

[16] $investment(combination)$

Answer:

Individuals with a lower income who already have an adequate savings account may want to consider splitting their surplus income between savings and stocks (***combination***).

Resolution refutation proofs

合取范式（conjunctive normal form），是命题公式的一种标准形。如（p∨q）∧(¬ r∨p)

- Put the premises or axioms into ***clause form***.
- Add the ***negation*** of what is to be proved, in clause form, to the set of axioms.
- ***Resolve*** these clauses together, producing new clauses that logically follow from them.
- Produce a ***contradiction*** by generating the ***empty clause***.
- The ***substitutions*** used to produce the empty clause are those under which the opposite of the negated goal is true.

证明：A->B<-->~AVB<-->~(A∩~B) <-->证明A∩~B矛盾，如果矛盾，结论成立

1、将待证明问题A->B转化为其否命题A∩~B

2、将合取范式，得到子句集（构成合取范式的有限个简单析取式的集合就是子句集)

3、对子句集进行归结，得到空子句

Example

- All dogs are animals

  $\forall x(dog(x) \rightarrow animal(x))$

- All animals will die

  $\forall y(animal(y) \rightarrow die(y))$

- Fido is a dog

  $dog(fido)$

Example

- All dogs are animals

  $\forall x(dog(x) \rightarrow animal(x))$

- All animals will die

  $\forall y(animal(y) \rightarrow die(y))$

- Fido is a dog

  $dog(fido)$

***Question:* Will Fido die?**

$die(fido)$

Example

	Clause Form
• All dogs are animals	
$\forall x(dog(x) \rightarrow animal(x))$	$\neg dog(x) \vee animal(x)$
• All animals will die	
$\forall y(animal(y) \rightarrow die(y))$	$\neg animal(y) \vee die(y)$
• Fido is a dog	
$dog(fido)$	$dog(fido)$

***Question:* Will Fido die?**

$die(fido)$	$\neg die(fido)$

Example

$\neg dog(x) \vee animal(x)$ $\neg animal(y) \vee die(y)$

Example

$$\begin{tikzcd}
\neg dog(x) \lor animal(x) \arrow[dr, dash, "\{\, y/x \,\}"'] & & \neg animal(y) \lor die(y) \arrow[dl, dash] \\
& \neg dog(y) \lor die(y) &
\end{tikzcd}$$

Example

$$\begin{tikzcd}
& \neg dog(x) \vee animal(x) \arrow[dr, dash, "\{ y/x \}"'] & & \neg animal(y) \vee die(y) \arrow[dl, dash] \\
dog(fido) \arrow[dr, dash, "\{ fido/y \}"'] & & \neg dog(y) \vee die(y) \arrow[dl, dash] & \\
& die(fido) & &
\end{tikzcd}$$

Example

$$\begin{tikzcd}
 & \neg dog(x) \vee animal(x) \arrow[drr, dash, "\{\, y/x \,\}"'] & & & \neg animal(y) \vee die(y) \arrow[dl, dash] \\
dog(fido) \arrow[dr, dash, "\{\, fido/y \,\}"'] & & & \neg dog(y) \vee die(y) \arrow[dll, dash] & \\
 & die(fido) \arrow[dr, dash, "\{\,\}"'] & & \neg die(fido) \arrow[dl, dash] & \\
 & & \square & &
\end{tikzcd}$$

Producing the clause form

$$\forall x([a(x) \wedge b(x)] \rightarrow [c(x,i) \wedge \exists y(\exists z[c(y,z)] \rightarrow d(x,y))]) \vee \forall x(e(x))$$

1. We eliminate the $\rightarrow$ by using the equivalent form.

$a \rightarrow b \equiv \neg a \vee b$

$$\forall x(\neg[a(x) \wedge b(x)] \vee [c(x,i) \wedge \exists y(\exists z[\neg c(y,z)] \vee d(x,y))]) \vee \forall x(e(x))$$

Producing the clause form

$$\forall x(\neg[a(x)\wedge b(x)]\vee[c(x,i)\wedge\exists y(\exists z[\neg c(y,z)]\vee d(x,y))])\vee\forall x(e(x))$$

2. We reduce the scope of negation. This may be accomplished using a number of the transformations.

$$\neg(\neg a)\equiv a$$

$$\neg\exists x(a(x))\equiv\forall x(\neg a(x))$$

$$\neg\forall x(b(x))\equiv\exists x(\neg b(x))$$

$$\neg(a\wedge b)\equiv\neg a\vee\neg b$$

$$\neg(a\vee b)\equiv\neg a\wedge\neg b$$

$$\forall x([\neg a(x)\vee\neg b(x)]\vee[c(x,i)\wedge\exists y(\exists z[\neg c(y,z)]\vee d(x,y))])\vee\forall x(e(x))$$

Producing the clause form

$$\forall x([\neg a(x) \vee \neg b(x)] \vee [c(x,i) \wedge \exists y(\exists z[\neg c(y,z)] \vee d(x,y))]) \vee \forall x(e(x))$$

3. We standardize by renaming all variables so that variables bound by different quantifiers have unique names.

$$\forall x(a(x)) \vee \forall x(b(x)) \equiv \forall x(a(x)) \vee \forall y(b(y))$$

$$\forall x([\neg a(x) \vee \neg b(x)] \vee [c(x,i) \wedge \exists y(\exists z[\neg c(y,z)] \vee d(x,y))]) \vee \forall w(e(w))$$

Producing the clause form

$$\forall x([\neg a(x) \vee \neg b(x)] \vee [c(x,i) \wedge \exists y(\exists z[\neg c(y,z)] \vee d(x,y))]) \vee \forall w(e(w))$$

4. Move all quantifiers to the left without changing their order.

$$\forall x \exists y \exists z \forall w([\neg a(x) \vee \neg b(x)] \vee [c(x,i) \wedge (\neg c(y,z) \vee d(x,y))]) \vee e(w)$$

Producing the clause form

$$\forall x \exists y \exists z \forall w([\neg a(x) \vee \neg b(x)] \vee [c(x,i) \wedge (\neg c(y,z) \vee d(x,y))]) \vee e(w)$$

5. All existential quantifiers are eliminated by a process called ***skolemization***.

$$\forall x \exists y(mother(x,y)) \qquad \forall x(mother(x,m(x)))$$

$$\forall x \forall y \exists z \forall w(foo(x,y,z,w)) \qquad \forall x \forall y \forall w(foo(x,y,f(x,y),w))$$

$$\forall x \forall w([\neg a(x) \vee \neg b(x)] \vee [c(x,i) \wedge (\neg c(f(x),g(x)) \vee d(x,f(x)))]) \vee e(w)$$

Producing the clause form

$$\forall x \forall w([\neg a(x) \vee \neg b(x)] \vee [c(x,i) \wedge (\neg c(f(x), g(x)) \vee d(x, f(x)))]) \vee e(w)$$

6. Drop all universal quantification. By this point only universally quantified variables exit with no variable conflicts.

$$[\neg a(x) \vee \neg b(x)] \vee [c(x,i) \wedge (\neg c(f(x), g(x)) \vee d(x, f(x)))] \vee e(w)$$

Producing the clause form

$$[\neg a(x) \vee \neg b(x)] \vee [c(x,i) \wedge (\neg c(f(x), g(x)) \vee d(x, f(x)))] \vee e(w)$$

7. We convert the expression to the conjunct of disjuncts form. This requires using the associative and distributive properties of $\wedge$ and $\vee$.

$$a \vee (b \vee c) \equiv (a \vee b) \vee c$$

$$a \wedge (b \wedge c) \equiv (a \wedge b) \wedge c$$

$$a \vee (b \wedge c) \equiv (a \vee b) \wedge (a \vee c)$$

$$[\neg a(x) \vee \neg b(x) \vee c(x,i) \vee e(w)] \wedge$$
$$[\neg a(x) \vee \neg b(x) \vee \neg c(f(x), g(x)) \vee d(x, f(x)) \vee e(w)]$$

Producing the clause form

$$[\neg a(x) \vee \neg b(x) \vee c(x,i) \vee e(w)] \wedge$$
$$[\neg a(x) \vee \neg b(x) \vee \neg c(f(x), g(x)) \vee d(x, f(x)) \vee e(w)]$$

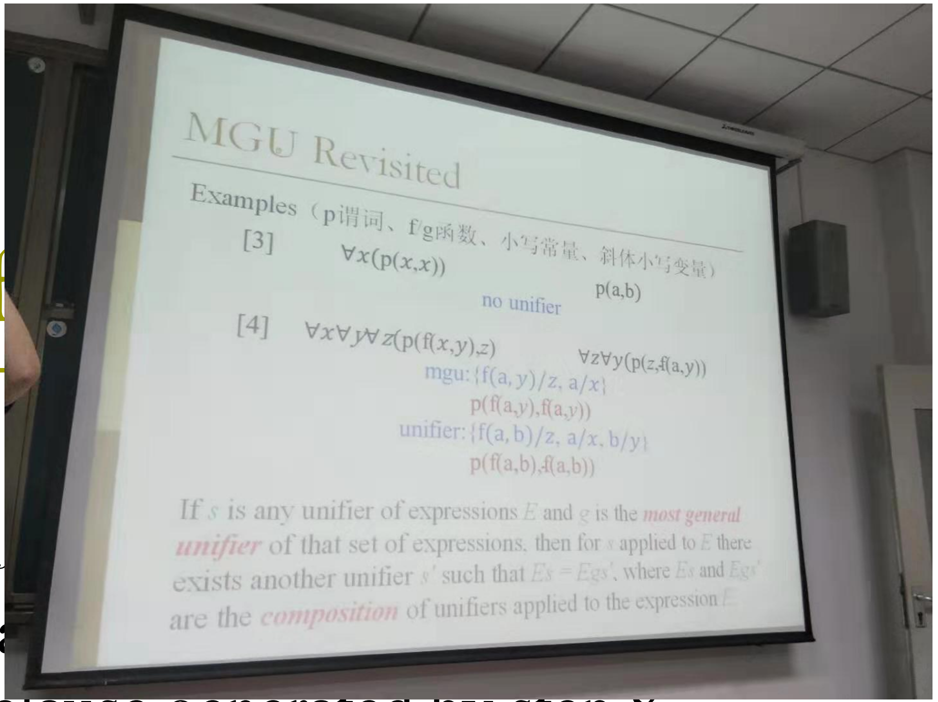
8. Now call each conjunct a separate clause

$$[\neg a(x) \vee \neg b(x) \vee c(x,i) \vee e(w)]$$
$$[\neg a(x) \vee \neg b(x) \vee \neg c(f(x), g(x)) \vee d(x, f(x)) \vee e(w)]$$

Producing the clause f

$[\neg a(x) \vee \neg b(x) \vee c(x,i) \vee e(w)]$

$[\neg a(x) \vee \neg b(x) \vee \neg c(f(x), g(x)) \vee d(x,$



MGU Revisited

Examples（p谓词、f/g函数、小写常量、斜体小写变量）

[3] $\forall x(p(x,x))$ p(a,b)

no unifier

[4] $\forall x \forall y \forall z(p(f(x,y),z)$ $\forall z \forall y(p(z,f(a,y))$

mgu: {f(a, y)/z, a/x}

p(f(a,y),f(a,y))

unifier: {f(a, b)/z, a/x, b/y}

p(f(a,b),f(a,b))

If *s* is any unifier of expressions *E* and *g* is the ***most general unifier*** of that set of expressions, then for *s* applied to *E* there exists another unifier *s'* such that *Es* = *Egs'*, where *Es* and *Egs'* are the ***composition*** of unifiers applied to the expression *E*

9. The final step is to standardize the va
requires giving the variable in each clause generated by step 8
different name.

$[\neg a(x) \vee \neg b(x) \vee c(x,i) \vee e(w)]$

$[\neg a(u) \vee \neg b(u) \vee \neg c(f(u), g(u)) \vee d(u, f(u)) \vee e(v)]$

Story of happy student

Anyone passing his history exams and winning the lottery is happy. But anyone who studies or is lucky can pass all his exams. John did not study but he is lucky. Anyone who is lucky wins the lottery. Is John happy.

Story of happy student

- Anyone passing history exams and winning the lottery is happy.

  $\forall x(pass(x, history) \wedge win(x, lottery) \rightarrow happy(x))$
- Anyone who studies or is lucky can pass all his exams.

  $\forall x \forall y(study(x) \vee lucky(x) \rightarrow pass(x, y))$
- John did not study but he is lucky.

  $\neg study(john) \wedge lucky(john)$
- Anyone who is lucky wins the lottery

  $\forall x(lucky(x) \rightarrow win(x, lottery))$

***Question:* Is John happy?**

$\neg happy(john)$

Story of happy student

- Anyone passing history exams and winning the lottery is happy.

  $\neg pass(x, history) \vee \neg win(x, lottery) \vee happy(x)$

- Anyone who studies or is lucky can pass all his exams.

  $\neg study(y) \vee pass(y, z)$ $\neg lucky(w) \vee pass(w, v)$

- John did not study but he is lucky.

  $\neg study(john)$ $lucky(john)$

- Anyone who is lucky wins the lottery

  $\neg lucky(u) \vee win(u, lottery)$

Question: *Is John happy?*

$\neg happy(john)$ 假 设 条 件 ?

Story of happy student

$$\begin{tikzcd}[column sep=small]
 & \lnot pass(x, history) \lor \lnot win(x, lottery) \lor happy(x) \arrow[d, "\{\, u/x \,\}"'] & \\
 & & \lnot lucky(u) \lor win(u, lottery) \arrow[dl] \\
\lnot happy(john) \arrow[d, "\{\, john/u \,\}"'] & \lnot pass(u, history) \lor happy(u) \lor \lnot lucky(u) \arrow[dl] & \\
\lnot pass(john, history) \lor \lnot lucky(john) \arrow[dr, "\{\ \}"'] & & lucky(john) \arrow[dl] \\
 & \lnot pass(john, history) &
\end{tikzcd}$$

Story of happy student

$$\begin{tikzcd}
\neg pass(john, history) \arrow[drr, dash, "\{ john/w, history/v \}"'] & & & \neg lucky(w) \vee pass(w, v) \arrow[dl, dash] \\
lucky(john) \arrow[dr, dash, "\{\ \}"'] & & \neg lucky(john) \arrow[dl, dash] & \\
& \square & &
\end{tikzcd}$$

Story of exciting life

All people who are not poor and are smart are happy. Those people who read are not stupid. John can read and is wealthy. Happy people have exciting life. Can anyone be found with an exciting life?

We assume

$$\forall x(smart(x) \equiv \neg stupid(x))$$

$$\forall y(wealthy(y) \equiv \neg poor(y))$$

Story of exciting life

- All people who are not poor and are smart are happy.

  $\forall x(\neg poor(x) \wedge smart(x) \rightarrow happy(x))$

- Those people who read are not stupid.

  $\forall y(read(y) \rightarrow smart(y))$

- John can read and is wealthy.

  $\neg poor(john) \wedge read(john)$

- Happy people have exciting life

  $\forall z(happy(z) \rightarrow exciting(z))$

Question:* *Can anyone be found with an exciting life?

$\neg \exists w(exciting(w))$

Story of exciting life

- All people who are not poor and are smart are happy.
  $poor(x) \vee \neg smart(x) \vee happy(x)$
- Those people who read are not stupid.
  $\neg read(y) \vee smart(y)$
- John can read and is wealthy.
  $\neg poor(john) \qquad read(john)$
- Happy people have exciting life
  $\neg happy(z) \vee exciting(z)$

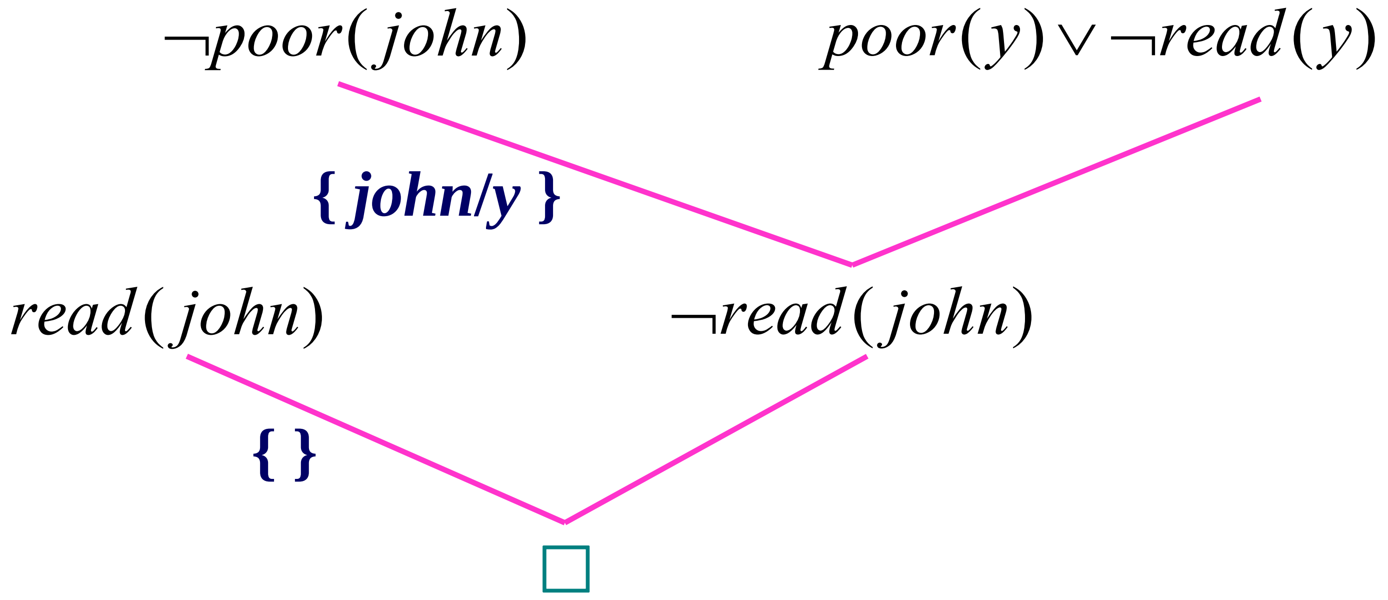
Question:* *Can anyone be found with an exciting life?

$\neg exciting(w)$

Story of exciting life

$$\begin{tikzcd}
 & \neg exciting(w) \arrow[dr, no head, "\{z/w\}"'] & & \neg happy(z) \lor exciting(z) \arrow[dl, no head] \\
 & poor(x) \lor \neg smart(x) \lor happy(x) \arrow[dr, no head, "\{x/z\}"'] & \neg happy(z) \arrow[d, no head] & \\
\neg read(y) \lor smart(y) \arrow[dr, no head, "\{y/x\}"'] & & poor(x) \lor \neg smart(x) \arrow[dl, no head] & \\
 & poor(y) \lor \neg read(y) & &
\end{tikzcd}$$

Story of exciting life



Another resolution refutation

$$\begin{tikzcd}
poor(x) \vee \neg smart(x) \vee happy(x) \arrow[rrd, dash, "\{\,z/x\,\}"'] & & \neg happy(z) \vee exciting(z) \arrow[d, dash] \\
\neg read(y) \vee smart(y) \arrow[d, dash, "\{\,y/z\,\}"'] & & exciting(z) \vee poor(x) \vee \neg smart(z) \arrow[lld, dash] \\
\neg read(y) \vee exciting(y) \vee poor(y) \arrow[rd, dash, "\{\,john/y\,\}"'] & & \neg poor(john) \arrow[ld, dash] \\
& \neg read(john) \vee exciting(john) &
\end{tikzcd}$$

Another resolution refutation

$$\begin{tikzcd}
\neg read(john) \vee exciting(john) \arrow[dr, dash, "\{\,\}"'] & & read(john) \arrow[dl, dash] & \\
& exciting(john) \arrow[dr, dash, "\{\,john/w\,\}"'] & & \neg exciting(w) \arrow[dl, dash] \\
& & \square &
\end{tikzcd}$$

Answer extraction from refutation

$$\begin{tikzcd}
\textbf{Question} & exciting(w) \arrow[d, swap, "\{ z/w \}"] \\
 & exciting(z) \arrow[d, swap, "\{ x/z \}"] \\
 & exciting(x) \arrow[d, swap, "\{ y/x \}"] \\
 & exciting(y) \arrow[d, swap, "\{ john/y \}"] \\
\textbf{Answer} & exciting(john)
\end{tikzcd}$$

Story of grandparent

Everyone has a parent. The parent of a parent is a grandparent. Given the person John, prove that John has a grandparent.

- Everyone has a parent

  $\forall x \exists y\, parent(x, y)$

- A parent of a parent is a grandparent

  $\forall x \forall y \forall z (parent(x, y) \wedge parent(y, z) \rightarrow grandparent(x, z))$

Question: ***Has John a grandparent*?**

$\neg \exists w (grandparent(john, w))$

Story of grandparent

Everyone has a parent. The parent of a parent is a grandparent. Given the person John, prove that John has a grandparent.

- Everyone has a parent

  $parent(x, pa(x))$

- A parent of a parent is a grandparent

  $\neg parent(w, y) \vee \neg parent(y, z) \vee grandparent(w, z)$

Question: ***Has John a grandparent?***

$\neg grandparent(john, v)$

Story of grandparent

$$\neg parent(w,y) \vee \neg parent(y,z) \vee grandparent(w,z)$$

$$\neg grandparent(john,v)$$

{ *john/w, v/z* }

$$parent(x, pa(x)) \qquad \neg parent(john,y) \vee \neg parent(y,v)$$

{ *john/x, pa(x)/y* }

$$parent(x, pa(x)) \qquad \neg parent(pa(john),v)$$

{ *pa(john)/x, pa(x)/v* }

$$\Box$$

Answer extraction from refutation

$$\begin{tikzcd}
\textbf{Question} & grandparent(john, v) \arrow[d, "\{ john/w, v/z\}"'] \\
& grandparent(john, v) \arrow[d, "\{ john/x, pa(x)/y \}"'] \\
& grandparent(john, v) \arrow[d, "\{ pa(john)/x, pa(x)/v \}"'] \\
\textbf{Answer} & grandparent(john, pa(pa(john)))
\end{tikzcd}$$

Strategies for resolution

- The ***breadth-first*** strategy.
- The ***set of support*** strategy.
- The ***unit preference*** strategy.
- The ***linear input form*** strategy

Linear input form

The linear input form is not a ***complete*** strategy, as can be seen by applying it to the following set of four clauses.

$$\neg a \vee \neg b$$

$$a \vee \neg b$$

$$\neg a \vee b$$

$$a \vee b$$

Herbrand structure

Herbrand Domain

$H = \bigcup_{i=0}^{\infty} H_i$ for a set of first-order clauses ***S***

$$H_0 = \begin{cases} \{\, a \,\} & \text{these is no constant in } \boldsymbol{S}. \\ \{\, c \mid c \text{ appear in } \boldsymbol{S} \,\} & \text{otherwise} \end{cases}$$

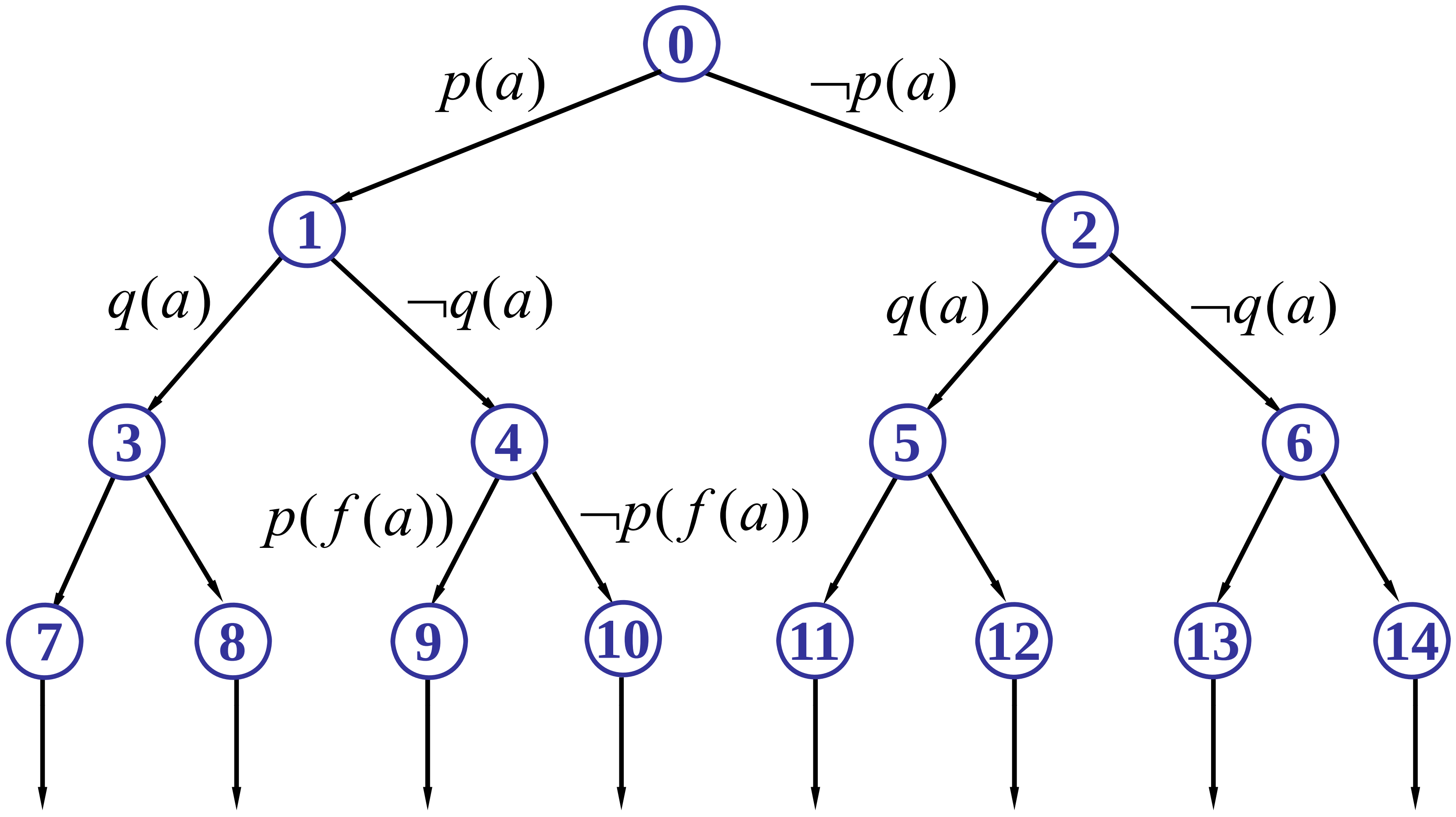
$$H_1 = H_0 \bigcup \{ f^{(n)} t_1 \cdots t_n \mid t_1, \cdots, t_n \in H_0 \}$$

$$H_{i+1} = H_i \bigcup \{ f^{(n)} t_1 \cdots t_n \mid t_1, \cdots, t_n \in H_i \}$$

where $f^{(n)}$ are functions appeared in ***S***.

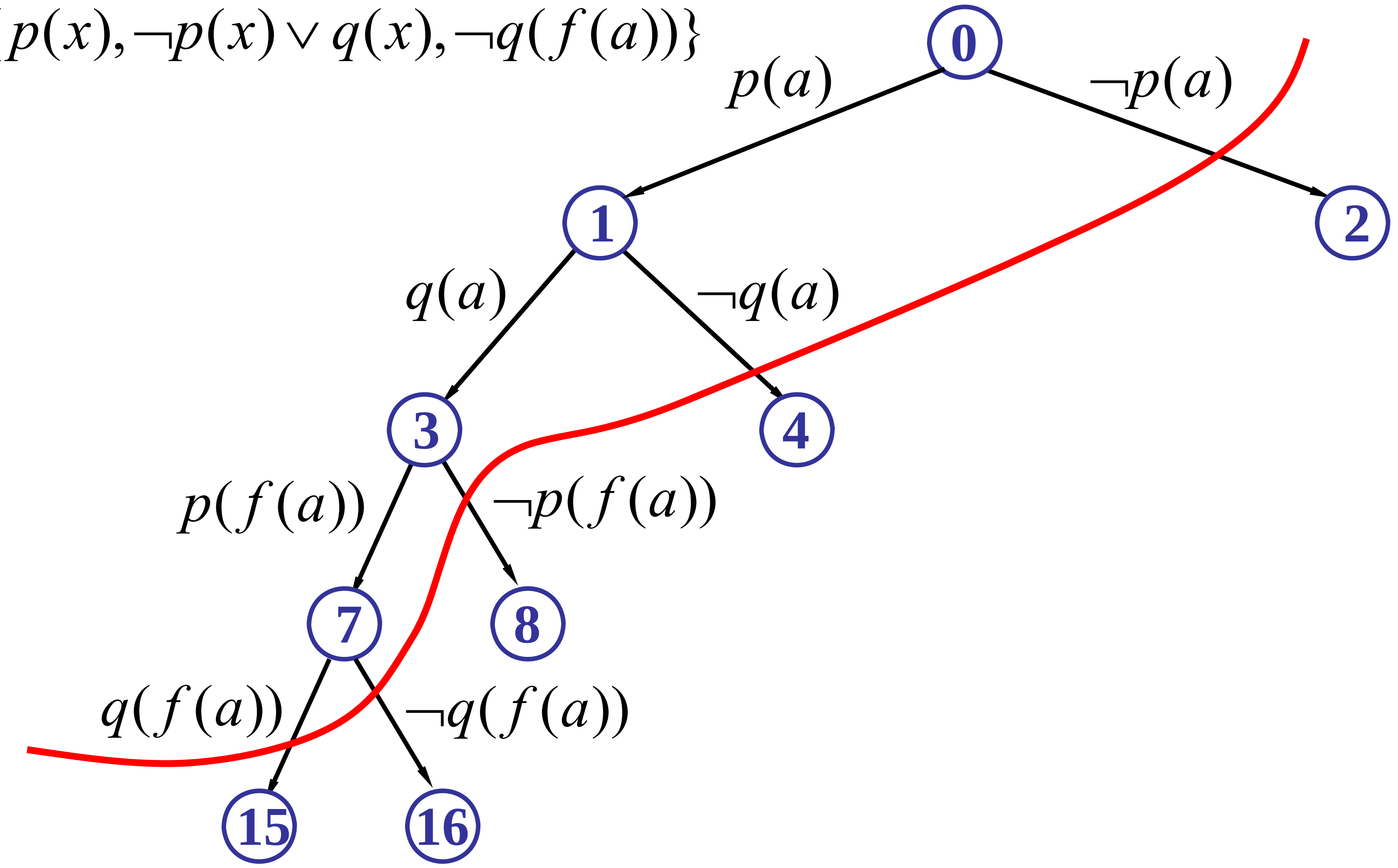
Semantic tree

$\{p(x), \neg p(x) \vee q(x), \neg q(f(a))\}$



Semantic tree

$\{p(x), \neg p(x) \vee q(x), \neg q(f(a))\}$



Categories of logic systems

- Very expressive but ***undecidable*** logics, typically variants of first- or higher-order logics;
- Quantifier-free formalisms of low computational complexity (typically ***P-*** or ***NP-complete***), such as (fragments of) classical propositional logic and its non-monotonic variants.
- ***Decidable*** logics with restricted quantification located between propositional and first-order logics, typical examples are modal, description and propositional temporal logics.

Default reasoning

- **Default Logic**

$$\frac{p : \neg r_1 \wedge \cdots \wedge \neg r_n}{q}$$

- **Circumscription**

$$(p \wedge \neg c) \rightarrow q$$

$$r_1 \rightarrow c$$

$$\vdots$$

$$r_n \rightarrow c$$

Modal logics

- **Necessarily**

  $\Box p$

- **Possibly**

  $\Diamond p$

- **Axiom**

  $\Box p \leftrightarrow \neg \Diamond \neg p$

  $\Box (p \to q) \to (\Box p \to \Box q)$

Truth maintenance systems

- Associate with the production of each conclusion its ***justification***. This justification indicates the derivation process for that conclusion. The justification must contain all the ***facts***, ***rules***, and ***assumptions*** used to produce the conclusion.
- Provide a mechanism that, when given a ***contradiction*** along with its justification, finds the set of false assumptions within that justification that led to the conclusion.
- Retract the ***false assumptions***.
- Create a mechanism that follows up the retracted assumptions and retracts any ***conclusion*** that uses a retracted false assumption.

Horn clauses

A Horn clause contains ***at most one positive literal***

$$a \vee \neg b_1 \vee \neg b_2 \vee \ldots \vee \neg b_n$$

To emphasize the key role of the one positive literal in resolutions, we generally write Horn clauses as ***implications*** with the ***positive literal as the conclusion***

$$a \leftarrow b_1 \wedge b_2 \wedge \ldots \wedge b_n$$

Three forms of Horn clauses

- Headless clause or ***goals***

  $\leftarrow a_1 \wedge a_2 \wedge \ldots \wedge a_n$

- ***Facts***

  $a_1 \leftarrow$

  $a_2 \leftarrow$

  $\vdots$

  $a_n \leftarrow$

- ***Rules*** relation

  $a_1 \leftarrow b_1 \wedge b_2 \wedge \ldots \wedge b_n$

Prolog

Given a ***goal***

$$\leftarrow a_1 \wedge a_2 \wedge \ldots \wedge a_n$$

Prolog interpreter sequentially searches for the ***first clause*** in logic program whose head unifies with a_1. if

$$a^1 \leftarrow b_1 \wedge b_2 \wedge \ldots \wedge b_n$$

is the ***reducing clause*** ξ with the unification, the goal clause then becomes

$$\leftarrow (b_1 \wedge b_2 \wedge \ldots \wedge b_n \wedge a_2 \wedge \ldots \wedge a_n)\xi$$

Prolog

The Prolog interpreter then continues by trying to reduce the ***leftmost goal***, b_1

$b^1 \leftarrow c_1 \wedge c_2 \wedge \ldots \wedge c_p$

under unification ζ. the ***goal*** then becomes

$\leftarrow (c_1 \wedge c_2 \wedge \ldots \wedge c_p \wedge b_2 \wedge \ldots \wedge b_m \wedge a_2 \wedge \ldots \wedge a_n)\xi\zeta$

if the goal is reduced to the null clause then the ***composition of unifications*** that made the reductions

$\leftarrow (\square)\xi\zeta \cdots \varsigma$

provide an ***interpretation*** under which the original goal clause was true.

Syntax for Prolog

English	Predicate calculus	Prolog
and	$\wedge$	**,**
or	$\vee$	**;**
only if	$\rightarrow$ 🗨	**:-**
not	$\neg$	**not**

Example

```
likes(george, kate)
likes(george, susie)
likes(george, wine)
likes(susie, wine)
likes(kate, gin)
```

```
?- likes(george, susie).
yes
?- likes(george, gin).
no
?- likes(george, x).
x = kate
;
x = susie
;
x = wine
;
no
```

Example

likes(*george*, *kate*)
likes(*george*, *susie*)
likes(*george*, *wine*)
likes(*susie*, *wine*)
likes(*kate*, *gin*)
friends(*x*, *y*) :- *likes*(*x*, *z*), *likes*(*y*, *z*)

?- *friends*(*george*, *susie*).
yes

Note:
Close world assumption or ***negation as failure***

List

[1, 2, 3, 4]

[[*george*, *kate*], [*allen*, *amy*], [*don*, *pat*]]

[*tom*, *dick*, *harry*, *fred*]

[ ]

Match

- [*tom*, *dick*, *harry*] is matched to [*x* | *y*]

  x = *tom* and *y* = [*dick*, *harry*]
- [*tom*, *dick*, *harry*] is matched to [*x*, *y* | *z*]

  x = *tom*, *y* = *dick* and *z* = [*harry*]
- [*tom*, *dick*, *harry*] is matched to [*x*, *y*, *z* | *w*]

  x = *tom*, *y* = *dick*, *z* = *harry* and *w* = [ ]
- [*tom*, *dick*, *harry*] will matched [*tom*, *x* | [*harry*]]

  x = *dick*
- [*tom*, *dick*, *harry*] will not matched [*x*, *y*, *z*, *w* | *u*]

Member

```
?- member(a, [a, b, c, d, e]).
yes
?- member(a, [1, 2, 3, 4]).
no
?- member(x, [a, b, c]).
x = a
;
x = b
;
x = c
;
no
```

Member in Prolog

1. *member*(x, [x | t]).

2. *member*(x, [y | t]) :- *member*(x, t).

?- *member*(c, [a, b, c]).

    call 1. fail, since $c \neq a$

    call 2. $x = c$, $y = a$, $t = [b, c]$, *member*(c, [b, c])?

        call 1. fail, since $c \neq b$

        call 2. $x = c$, $y = b$, $t = [c]$, *member*(c, [c])?

            call 1. success, since $c = c$

        yes (to second **call 2.**)

    yes (to first **call 2.**)

yes

Reordering of clauses and goals

```
predecessor(Parent, Child) :- parent(Parent, Child).

predecessor(Predecessor, Successor) :-
                        parent(Predecessor, Child),
                        predecessor(Child, Successor).
```

Facts:

```
parent(tom, liz).      parent(pam, bob).
parent(bob, ann).      parent(tom, bob).
parent(bob, pat).      parent(pat, jim).
```

Question:

```
predecessor(tom, pat).
```

Variation 1

```
predecessor(Predecessor, Successor) :-
                        parent(Predecessor, Child),
                        predecessor(Child, Successor).

predecessor(Parent, Child) :- parent(Parent, Child).
```

Facts:

```
parent(tom, liz).        parent(pam, bob).
parent(bob, ann).        parent(tom, bob).
parent(bob, pat).        parent(pat, jim).
```

Question:

```
predecessor(tom, pat).
```

Variation 2

```
predecessor(Parent, Child) :- parent(Parent, Child).

predecessor(Predecessor, Successor) :-
                         predecessor(Predecessor, Child),
                         parent(Child, Successor).
```

Facts:

```
parent(tom, liz).        parent(pam, bob).
parent(bob, ann).        parent(tom, bob).
parent(bob, pat).        parent(pat, jim).
```

Question:

```
predecessor(tom, pat).
```

Variation 3

```
predecessor(Predecessor, Successor) :-
                    predecessor(Predecessor, Child),
                    parent(Child, Successor).

predecessor(Parent, Child) :- parent(Parent, Child).
```

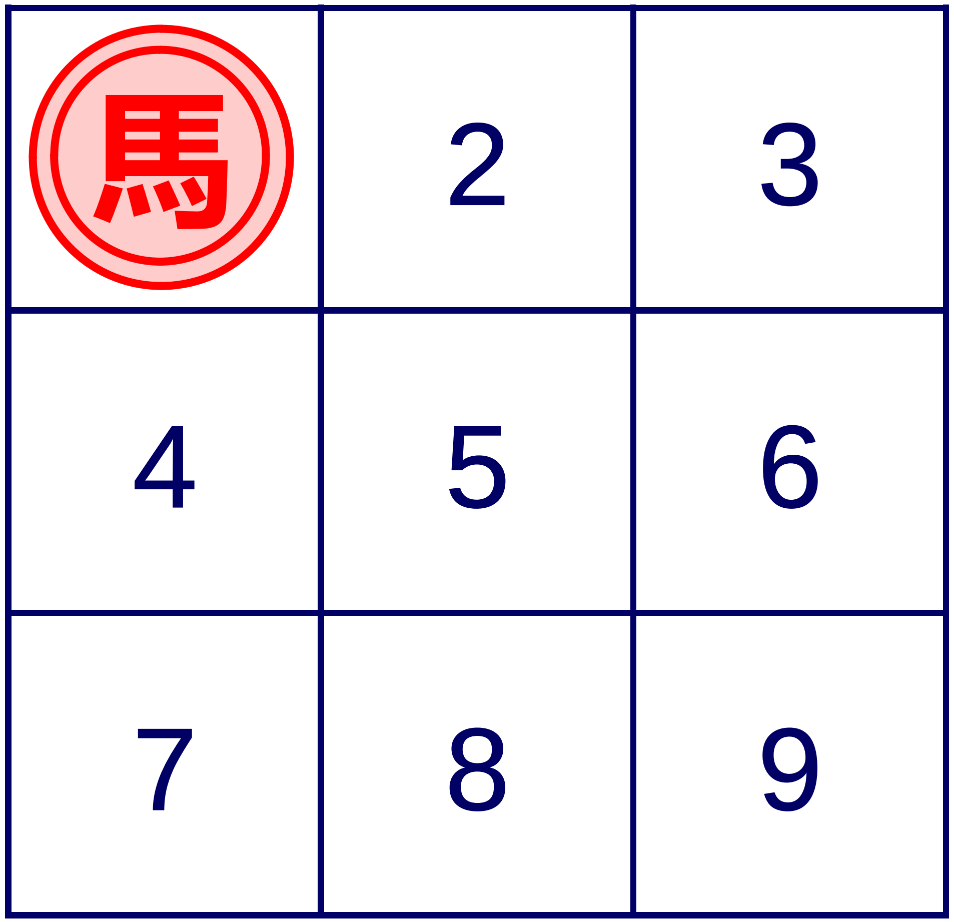
Facts:

```
parent(tom, liz).     parent(pam, bob).
parent(bob, ann).     parent(tom, bob).
parent(bob, pat).     parent(pat, jim).
```

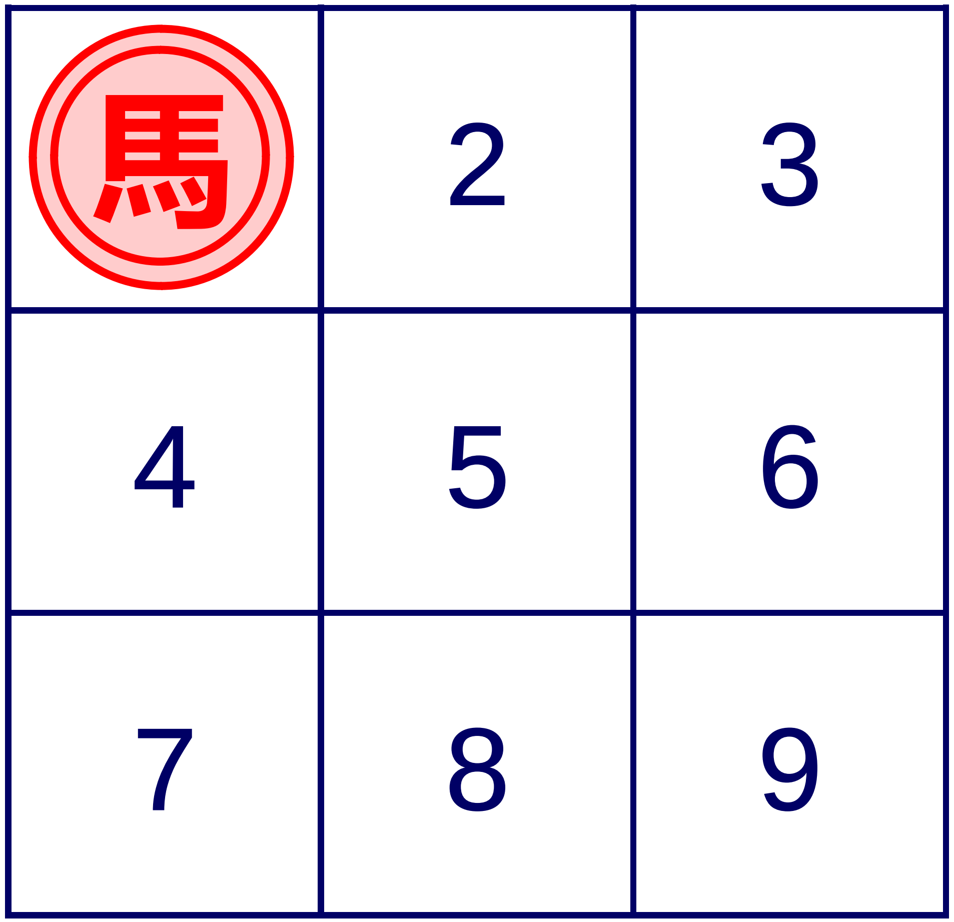
Question:

```
predecessor(tom, pat).
```

Knight's tour problem



Knight's tour problem



Knight's tour problem

1	2	3
4	5	6
7	8	9

move(1, 6), *move*(3, 4), *move*(6, 7), *move*(8, 3).
move(1, 8), *move*(3, 8), *move*(6, 1), *move*(8, 1).
move(2, 7), *move*(4, 3), *move*(7, 6), *move*(9, 4).
move(2, 9), *move*(4, 9), *move*(7, 2), *move*(9, 2).

1. *path*(*z*, *z*, *l*).

2. *path*(*x*, *y*, *l*) :- *move*(*x*, *z*), *not*(*member*(*z*, *l*)), *path*(*z*, *y*, [*z* | *l*]).

Question:

?- *path*(1, 3, [1]).

Knight's tour problem

path(1, 3, [1]) attempts to match **1**. fail $1 \neq 3$.

path(1, 3, [1]) matches **2**. *x* is 1, *y* is 3, *l* is [1]

  move(1, *z*) matches *z* as 6, *not*(*member*(6, [1])) true,
  call *path*(6, 3, [6, 1]).

  path(6, 3, [6, 1]) attempts to match **1**. fail $6 \neq 3$.

  path(6, 3, [6, 1]) matches **2**. *x* is 6, *y* is 3, *l* is [6, 1]

    move(6, *z*) matches *z* as 7, *not*(*member*(7, [6, 1])) is true,
    call *path*(7, 3, [7, 6, 1]).

    path(7, 3, [7, 6, 1]) attempts to match **1**. fail $7 \neq 3$.

    path(7, 3, [7, 6, 1]) matches **2**. *x* is 7, *y* is 3, *l* is [7, 6, 1]

      move(7, *z*) matches *z* as 6, *not*(*member*(6, [7, 6, 1])) fails,
      backtrack!

      move(7, *z*) matches *z* as 2, *not*(*member*(2, [7, 6, 1])) true,
      call *path*(2, 3, [2, 7, 6, 1]).

Knight's tour problem

path(2, 3, [2, 7, 6, 1]) attempts to match **1**. fail $2 \neq 3$.

path(2, 3, [2, 7, 6, 1]) matches **2**. *x* is 2, *y* is 3, *l* is [2, 7, 6, 1]

move matches *z* as 7, *not*(*member*(...)) fails, backtrack!

move matches *z* as 9, *not*(*member*(...)) true,

call *path*(9, 3, [9, 2, 7, 6, 1]).

path fails **1**. $9 \neq 3$.

path matches **2**. *x* is 9, *y* is 3, *l* is [9, 2, 7, 6, 1]

move is $z = 4$, *not*(*member*(...)) true,

call *path*(4, 3, [4, 9, 2, 7, 6, 1]).

path fails **1**. $4 \neq 3$.

path matches **2**. *x* is 4, *y* is 3, *l* is [4, 9, 2, 7, 6, 1]

move is $z = 3$, *not*(*member*(...)) true,

call *path*(3, 3, [3, 4, 9, 2, 7, 6, 1]).

path **1**. true, $3 = 3$, yes.

The use of cut to control search

move(1, 6), *move*(1, 8), *move*(6, 7).
move(6, 1), *move*(8, 3), *move*(8, 1).

path2(*x*, *y*) :- *move*(*x*, *z*), *move*(*z*, *y*).

?- *path2*(1, *w*).
w = 7
;
w = 1
;
w = 3
;
w = 1
;
no

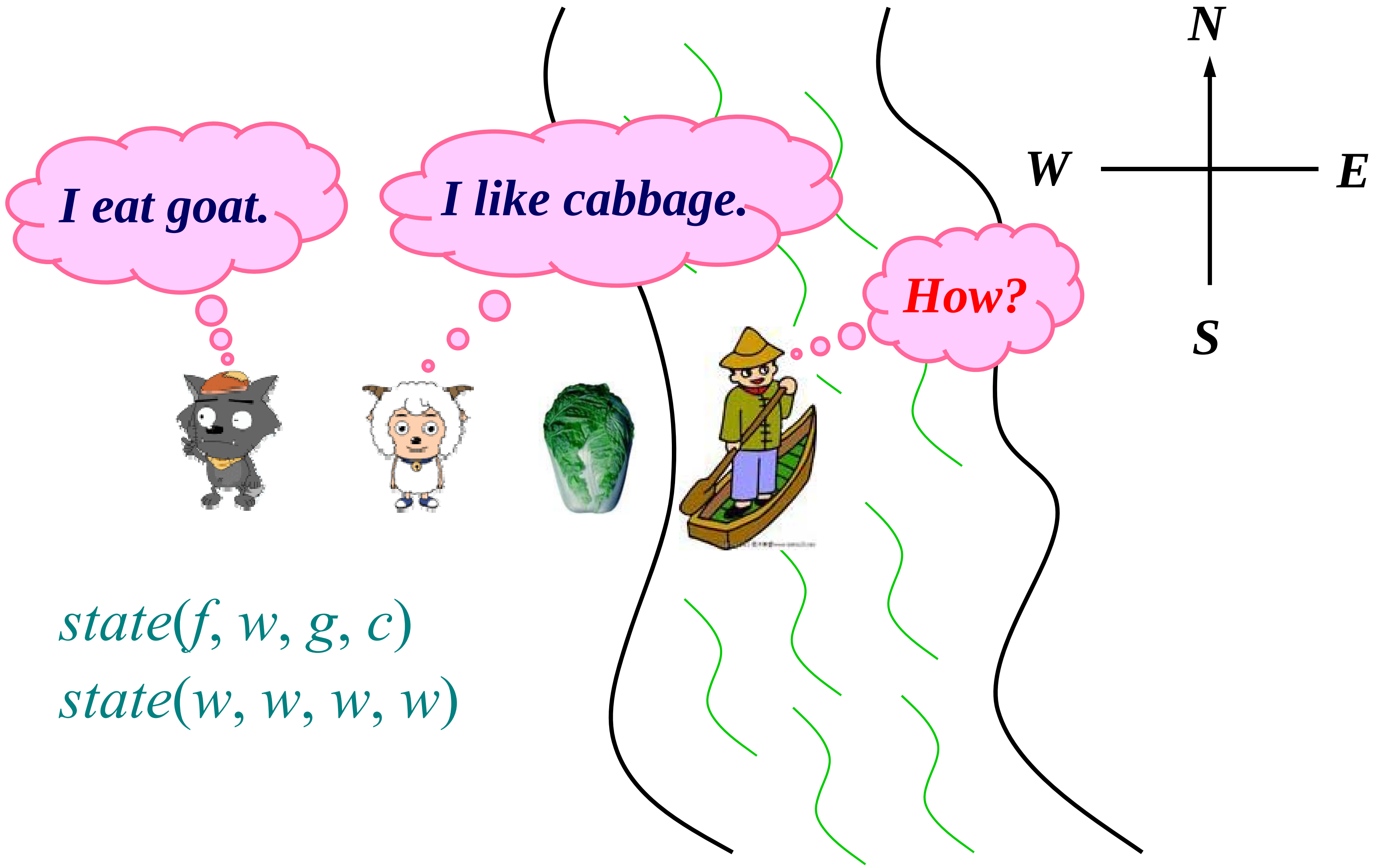
The use of cut to control search

```
move(1, 6), move(1, 8), move(6, 7).
move(6, 1), move(8, 3), move(8, 1).

path3(x, y) :- move(x, z), ! , move(z, y).

?- path3(1, w).
w = 7
;
w = 1
;
no
```

Farmer, wolf, goat, and cabbage



state(f, w, g, c)

state(w, w, w, w)

Farmer, wolf, goat, and cabbage

A ***farmer*** *with its* ***wolf, goat****, and* ***cabbage*** *come to the edge of a river they wish to cross. There is a boat at the river's edge, but, of course, only the farmer can row. the boat also can carry only two things (including the rower) at a time. If the wolf is ever left alone with the goat, the wolf will eat the goat; similarly, if the goat is left alone with the cabbage, the goat will eat the cabbage. Devise a* ***sequence*** *of crossings of the river so that all four characters arrive* ***safely*** *on the other side of the river.*

Farmer, wolf, goat, and cabbage

```
move(state(x, x, g, c), state(y, y, g, c)) :-
  opposite(x, y), not(unsafe(state(y, y, g, c))),
  writelist([' Try farmer takes wolf ', y, y, g, c])
move(state(x, w, x, c), state(y, w, y, c)) :-
  opposite(x, y), not(unsafe(state(y, w, y, c))),
  writelist([' Try farmer takes goat ', y, w, y, c])
move(state(x, w, g, x), state(y, w, g, y)) :-
  opposite(x, y), not(unsafe(state(y, w, g, y))),
  writelist([' Try farmer takes cabbage ', y, w, g, c])
move(state(x, w, g, c), state(y, w, g, c)) :-
  opposite(x, y), not(unsafe(state(y, w, g, c))),
  writelist([' Try farmer takes self ', y, w, g, c])
```

Farmer, wolf, goat, and cabbage

```
move(state(f, w, g, c), state(f, w, g, c)) :-
  writelist([' BACKTRACK from ', f, w, g, c]), fail.
path(goal, goal, been_stack) :-
  write(' Solution path is '), nl,
  reverse_print_stack(been_stack).
path(state, goal, been_stack) :-
  move(state, next_state),
  not(member_stack(next_state, been_stack)),
  stack(next_state, been_stack, new_been_stack),
  path(next_state, goal, new_been_stack), !.
opposite(e, w).
opposite(w, e).
```

Farmer, wolf, goat, and cabbage

```
go(start, goal) :-
  empty_stack(empth_been_stack),
  stack(start, empth_been_stack, been_stack),
  path(state, goal, been_stack)
```

Question:

```
?- go(state(w, w, w, w), state(e, e, e, e)).
```

Farmer, wolf, goat, and cabbage

Process:

Try farmer takes goat *e w e w*
Try farmer takes self *w w e w*
Try farmer takes wolf *e e e w*
Try farmer takes goat *w e w w*
Try farmer takes cabbage *e e w e*
Try farmer takes wolf *w w w e*
Try farmer takes goat *e w e e*
  BACKTRACK from *e, w, e, e*
  BACKTRACK from *e, w, e, e*
Try farmer takes self *w e w e*
Try farmer takes goat *e e e e*

Solution path is:

state(*w*, *w*, *w*, *w*)
state(*e*, *w*, *e*, *w*)
state(*w*, *w*, *e*, *w*)
state(*e*, *e*, *e*, *w*)
state(*w*, *e*, *w*, *w*)
state(*e*, *e*, *w*, *e*)
state(*w*, *e*, *w*, *e*)
state(*e*, *e*, *e*, *e*)

Any question?

Xiaoqing Zheng
Fundan University