Logic Programming – summary

Contents

1. Introduction

Why different programming languages?

- Choose the right language for the right application
- Eases learning of new programming languages

Main application area of logic programming:

- Artificial intelligence
- Expert systems
- Robotics

Prolog-program consists of <u>facts</u> and <u>rules</u>

Comments in prolog:

% for single line

/* */ for block

Execution of logic programs: ask queries

Prolog uses a closed-world-assumption: if a statement is not implied by the formulas in the program, then it must be false

<u>Variables</u> in programs:

e.g.

human(X).

?-motherOf(X, susanne)

variables in a program are universally quantified

variables in a query are existentially quantified

In prolog, the query determines what is input/output.

Prolog treats combination of queries from left to right and top to down

Prolog constructs a proof tree and backtracks in case of failure. It stops as soon as □ is reached and returns the instantiation of the variables that corresponds to the path from the root to □.

<u>Rules:</u>

Rules are needed to deduce new knowledge from existing knowledge

Head :- body

Recursive rules can lead to non-termination

Logic programming resulted from automated theorem proving, is particularly suitable for AI, deductive data bases, rapid prototyping

2.1 Syntax of predicate logic

- <u>Signature</u>: alphabet for predicate logic

A signature (Σ, Δ) is a pair with $\Sigma = \bigcup_{n\in N} \Sigma n$ and $\Delta = \bigcup_{n\in N} \Delta n$, where all Σn and Δn are pairwise disjoint. Every f∈ Σn is a <u>function symbol</u> of arity n, every p ∈ Δn is a <u>predicate symbol</u> of arity n. We always require Σ_0 (i.e. the set of constants)≠ ∅ .

- Function symbol: 就是各种可以代入变量的那些实例
- Predicate symbol: 就是各种函数

- <u>Terms</u>: are the objects of predicate logic
  Let (Σ, Δ) be a signature, let V be a set of variables with V∩Σ = ∅（这是为了避免冲突）.
  Then τ (Σ, V) is the set of terms (over Σ and V). Here, τ(Σ, V) is the smallest set with:
  1. V ⊆ τ(Σ, V)
  2. f(t1,...,tn) ∈ τ(Σ, V) if f ∈ Σn and t1,...,tn ∈ τ(Σ, V) for some n∈ N

  (本质意思就是 term 就是包含所有变量以及实例的东西)
- τ(Σ) stands for τ(Σ, ∅), i.e. the set of <u>ground terms</u>. （data(2,3,1993)也是 ground term）
- For any term t, V(t) is the set of all variables in t.
- <u>Formulas:</u> represent statements about terms
  Let (Σ, Δ) be a signature, let V be a set of variables. The set of <u>atomic formulas</u> over (Σ, Δ) and V is defined as :
  At(Σ,Δ,V) = {p(t1,. . . , tn) | p ∈Δ n for some n, t1, . . . , tn ∈ τ (Σ , V)}
  (意思就是 atomic formula 就是所有 predicate symbol，带着所有 term 组成的基本式子，比如 motherOf(X,susanne))
  F(Σ, Δ, V) is the set of <u>formulas</u> over (Σ, Δ) and V.
  Here, F(Σ, Δ, V) is the smallest set with:
  1. At(Σ, Δ, V) ⊆ F(Σ, Δ, V)
  2. If ϕ ∈ F(Σ, Δ, V), then ¬ ϕ ∈ F(Σ , Δ , V)
  3. If ϕ1, ϕ2 ∈ F(Σ,Δ, V), then (ϕ1 ∧ ϕ2), (ϕ1 ∨ ϕ2), (ϕ1 → ϕ2), (ϕ1 ↔ ϕ2) ∈F(Σ,Δ, V)
  4. If X ∈ V and ϕ ∈ F(Σ,Δ, V), then (∀X ϕ), (∃X ϕ) ∈ F(Σ,Δ, V)

  (本质就是在原子 formula 基础上，增加了各种 formula 的操作)
- V(φ) is the set of variables in formula φ (term 也拥有这种提取变量的操作)
- A variable X is <u>free</u> in a formula φ if either one of these holds
  1. ϕ is an atomic formula and X ∈ V(Φ)
  2. ϕ = ¬ϕ1 und X is free in ϕ1
  3. ϕ = (ϕ1 · ϕ2) with · ∈ {∧,∨,→,↔} and X is free in ϕ1 or in ϕ2
  4. ϕ = (QY ϕ1) with Q ∈ {∀, ∃}, X is free in ϕ1 and X ≠ Y .(意思就是 Y 被限制了，但 X 没有在要求中，所以是 free 的)
- A formula is <u>closed</u> if it has no free variables (也就是所有变量都有限定).
- A formula is <u>quantifier-free</u> if it does not contain ∀ or ∃
- E.g. motherOf(X, susanne)中 X 就是 free 的
- Every logic program can be translated into a set of formulas. All variables in the program are universally quantified
- Substitution
  A mapping σ : V→ τ(Σ, V) is a substitution if σ(X) ≠ X for finitely(这里要求有限个是为了可以表示这个 substitution，就是 X,Y,Z 替换成啥啥) many X ∈ V.

$DOM(\sigma) = \{X \in V \mid \sigma(X) \neq X\}$ is the <u>domain</u> of σ
$RANGE(\sigma) = \{\sigma(X) \mid X \in DOM(\sigma)\}$ is the <u>range</u> of σ

- Since DOM(σ) is finite, a substitution σ can be represented as a finite set of pairs $\{X/\sigma(X) \mid X \in DOM(\sigma)\}$
- A substitution is a ground substitution if $V(\sigma(X)) = \emptyset$ for all $X \in DOM(\sigma)$，意思就是当前 substitution 中包含的变量都转换成了实例，并不针对某个 formula
- A substitution is a variable renaming if σ is injective(单射的) and $\sigma(X) \in V$ for all $X \in V$. 也就是说 σ(X) =Y, σ(Y) = Z, σ(Z) = Z 不是 renaming，只有一一映射才是。注意上式是对于所有 X 属于 V，所以只有涉及到的变量全部成环（可以多余 1 个），才算 renaming
- Substitutions are also applied to terms(刚才说的都是 finite set of 变量)
  $\sigma: \tau(\Sigma, V) \rightarrow \tau(\Sigma, V)$ is defined as $\sigma(f(t1,\dots,tn)) = f(\sigma(t1),\dots, \sigma(tn))$
- Substitutions are also applied to formulas:
  (1) $\sigma(p(t1, \dots, tn)) = p(\sigma(t1), \dots, \sigma(tn))$
  (2) $\sigma(\neg\phi) = \neg\sigma(\phi)$
  (3) $\sigma(\phi1 \cdot \phi2) = \sigma(\phi1) \cdot \sigma(\phi2)$ for $\cdot \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$
  (4) $\sigma(QX\ \phi) = QX\ \sigma(\phi)$, for $Q \in \{\forall, \exists\}$, if $X \notin V(RANGE(\sigma)) \cup DOM(\sigma)$
  (5) $\sigma(QX\ \phi) = QX'\ \sigma(\delta(\phi))$, for $Q \in \{\forall, \exists\}$, if $X \in V(RANGE(\sigma)) \cup DOM(\sigma)$.
  Here X' is a fresh variable with $X' \notin V(RANGE(\sigma)) \cup DOM(\sigma) \cup V(\phi)$ and $\delta = \{X/X'\}$.
  上面之所以 range 外用 V 提取了一下，因为 range 可以产生实例，也就是非 variable，而 domain 不会。
  分两种情况，看属不属于那两个的并集
  注意 5 中 X'也不属于 V(ϕ)
  本质上就是不替换 qualifier 涉及的东西（而是换个名字），替换掉其他的。
- An instance σ(t) is a ground instance if no in it.(就是所有都转换成非变量实例了)

2.2 Semantics of predicate logic

- <u>Interpretation, structure</u>
  For a signature (Σ, Δ), an interpretation has the form I = (A,α,β)
  A is an arbitrary set with $A \neq \emptyset$ (carrier)
  α maps every $f \in \Sigma n$ to a function $\alpha_f: A^n \rightarrow A$
  α also maps every $p \in \Delta n$ to a $\alpha_p \subseteq A^n$ if $n \geqslant 1$. Else if n=0, $\alpha_p \in \{TRUE, FALSE\}$
  We say that α_f (or α_p) is the meaning of f(or p)
  $\beta: V \rightarrow A$ is called a variable assignment（变量是 β ，别的都是 α ）
- Every interpretation I gives a meaning to every term
  $I : \tau(\Sigma, V) \rightarrow A$ 其实就是:
  $I(X) = \beta(X)$ for all $X \in V$
  $I(f(t1, \dots, tn)) = \alpha f (I(t1), \dots, I(tn))$ for all $f \in \Sigma n$ and $t1, \dots, tn \in \tau(\Sigma, V)$
  说白了，就是替成各自的含义
- For $X \in V$ and $a \in A$, let β [[X/a]] be the variable assignment with β [[X/a]](X) = a and β [[X/a]](Y) = β(Y) for all $Y \in V$ with $Y \neq X$.
  For I = (A,α,β), let I[[X/a]] = (A, α, β[[X/a]]).

An interpretation I = (A,α,β) satisfies a formula ϕ ∈ F(Σ,Δ,V), denoted “I |= ϕ”, iff

1. ϕ = p(t1, . . . , tn) and (I(t1), . . . , I(tn)) ∈ α_p if p ∈ Δ_n and n ⩾ 1
2. ϕ = p and α_p = TRUE if p ∈ Δ_0
3. ϕ = ¬ϕ1 and I ⊭ ϕ1
4. ϕ = ϕ 1 ∧ ϕ 2 and I |= ϕ 1 and I |= ϕ 2
5. ϕ = ϕ 1 ∨ ϕ 2 and (I |= ϕ 1 or I |= ϕ 2)
6. ϕ = ϕ 1 → ϕ 2 and if I |= ϕ 1, then also I |= ϕ 2
7. ϕ = ϕ1 ↔ ϕ2 and (I |= ϕ1 iff I |= ϕ2)
8. ϕ = ∀X ϕ1 and I[[X/a]] |= ϕ1 for all a ∈ A
9. ϕ = ∃X ϕ1 and I[[X/a]] |= ϕ1 there exists a ∈ A

上面这些就是定义什么叫 satisfy，0 维就得是 true
可以看作是某个 query，在相应 interpretation 下，能不能得到 true

- An interpretation I is a model of ϕ iff I |= ϕ. I is a model of a set of formulas Φ iff I|= ϕ for all ϕ∈ Φ
- Two formulas ϕ1, ϕ2 are equivalent iff we have (I|= ϕ1 iff I|= ϕ2) for all interpretations I.
- A formula is satisfiable iff it has a model
- A formula is valid if every interpretation is a model of the formula
- E.g.
  married(gerd, susanne) is satisfiable
  married(2,1) ∨¬married(2,1) is valid
  φ ∧ ¬φ is unsatisfiable
- An interpretation without variable assignment is called a structure S = (A, α)
- If we only regard closed formulas(i.e. no free variables), then satisfiability etc. can be defined for structures:
  S|= ϕ iff I|= ϕ for some interpretation I =(A, α, β)(这里 β 没卵用)

- Substitution σ: V→τ(Σ, V) (syntactic)
  Variable assignment β: V→ A (semantic)
- Substitution lemma
  Let I = (A,α, β) be an interpretation for a signature (Σ, Δ), let σ = {X1/t1, . . . ,Xn/tn}. Then we have:
  (a) I(σ (t)) = I[[X1/I(t1), . . . ,Xn/I(tn)]](t) for all t ∈ τ(Σ, V)
  (b) I |= σ (ϕ) iff I[[X1/I(t1), . . . ,Xn/I(tn)]] |= ϕ for all ϕ ∈ F(Σ, Δ, V)
  这是两个部分，都是正确结论，其中 a 是说明了，先算或后算再替代结果一样
  b 是我们证明时用的，用了来替换一个 universally quantified to a certain one
- Entailment
  A set of formulas Φ entails a formula φ (denoted Φ |= φ) iff for all interpretations I with I |= Φ we also have I |= φ.
  意思就是满足 Φ 所有可能的 interpretation，都有后者。
- 现在|= 有两个含义：
  I |= φ means interpretation I satisfies φ
  Φ |= φ means Φ entails φ
- ∅|=φ, or |= φ means φ is valid
- When asking query ?-male(gerd).

Means prove that Φ |= male(gerd)

Where Φ is the formulas in the program

3.1 Skolem normal form

- Resolution:

  A calculus that prove $\Phi \models \varphi$ in a syntactical(automatable) way

  Sound: if the calculus can deduce φ from Φ, then $\Phi \models \varphi$ really holds

  Complete: if whenever $\Phi \models \varphi$, then the calculus can deduce φ from Φ
- Logic programming uses the resolution calculus, which is sound and complete
- Lemma from entailment to unsatisfiability:

  Let $\phi 1, \dots, \phi k, \phi \in F(\Sigma, \Delta, V)$. Then we have $\{\phi 1, \dots, \phi k\} \models \phi$ iff the formula $\phi 1 \wedge \dots \wedge \phi k \wedge \neg \phi$ is unsatisfiable
- Unsatisfiability as well as entailment are undecidable(so no terminating algorithm), but are also semi-decidable(an algorithm which terminates whenever $\Phi \models \varphi$ holds, but may not terminate when $\Phi \not\models \phi$), and resolution is such a semi-decision algorithm
- Skolem normal form:

  In this form: $\forall X1, \dots, Xn\ \psi$ where ψ is quantifier-free and $V(\psi) \subseteq \{X1,\dots,Xn\}$(i.e. it is closed，也就是 female(X)不行，必有前缀，如果把 X fresh 掉成为 female(a)就成了)
- Prenex normal form:

  A formula ψ is in prenex normal form iff it has the form $Q1\ X1 \dots Qn\ Xn\ \psi$ with $Qi \in \{\forall, \exists\}$ and it has ψ quantifier-free(可以没有前缀，也就是说不一定要 closed)
- Algorithm for prenex normal form:

  replace $\phi 1 \leftrightarrow \phi 2$ in ϕ by $(\phi 1 \rightarrow \phi 2) \wedge (\phi 2 \rightarrow \phi 1)$

  replace $\phi 1 \rightarrow \phi 2$ in ϕ by $\neg \phi 1 \vee \phi 2$

  then use PRENEX:

  1. If ϕ is quantifier-free, then return ϕ.
  2. If $\phi = \neg\phi 1$, then compute PRAENEX($\phi 1$) = $Q1\ X1 \dots Qn\ Xn\ \psi$. Then return $\overline{Q1}\ X1 \dots \overline{Qn}\ Xn \neg\psi$ where $\overline{\forall} = \exists$ and $\overline{\exists} = \forall$.
  3. If $\phi = \phi 1 \cdot \phi 2$ with $\cdot \in \{\wedge, \vee\}$, then compute PRAENEX($\phi 1$) = $Q1\ X1 \dots Qn\ Xn\ \psi 1$ and PRAENEX($\phi 2$) = $R1\ Y1 \dots Rm\ Ym\ \psi 2$. By renaming bound variables, ensure that $X1, \dots, Xn$ do not occur in $R1\ Y1 \dots Rm\ Ym\ \psi 2$ and that $Y1, \dots, Ym$ do not occur in $Q1\ X1 \dots Qn\ Xn\ \psi 1$. Then return $Q1\ X1 \dots Qn\ Xn\ R1\ Y1 \dots Rm\ Ym\ (\psi 1 \cdot \psi 2)$
  4. If $\phi = QX\ \phi 1$ with $Q \in \{\forall, \exists\}$, then compute PRAENEX($\phi 1$) = $Q1\ X1 \dots Qn\ Xn\ \psi$. By renaming bound variables, ensure that $X1, \dots, Xn$ are different from X. Then return $QX\ Q1\ X1 \dots Qn\ Xn\ \psi$.(这个就是在确保名字不一样了后，添在开头，也就是这么长的原因)
- Every formula has equivalent formula in prenex normal form, but this is false for skolem normal form, however, we only need that satisfiability iff between skolem and original
- Algorithm for skolem normal form:

  First transformed to prenex normal form, results in $\varphi 1$

  Let $X1, \dots, Xn$ be the free variables of $\varphi 1$, then transform $\varphi 1$ to $\varphi 2 = \exists X1,\dots,Xn\ \varphi 1$

Now $\varphi 2$ is a closed formula

We eliminate the existential quantifiers from the outside to the inside.

If $\varphi 2$ is $\forall X1, . . . ,Xn \exists Y \psi$, then replace it by $\forall X1, . . . ,Xn\ \psi[Y/f(X1, . . . ,Xn)]$

注意，fresh function of arity n 只跟∃ 外层的∀ 有关系，所以化归的时候顺序不能随便调

3.2 Herbrand structure

- Prove unsatisfiability of formula need to check all interpretations I = (A,α, β) in general, but for those in skolem normal form, search space is restricted
  1. β not needed, since formula is closed
  2. A can be fixed, we only have to consider A = τ(Σ), (i.e. all ground terms)
  3. α_f can be fixed for all f∈ Σ
  4. only α_p is still open for p ∈ Δ
- Herbrand structure

  Let (Σ, Δ) be a signature, a herbrand structure has the form (τ(Σ), α) where for all f∈Σ_n with n∈N, we have: α_f (t1, . . . , tn) = f(t1, . . . , tn).
- If a herbrand structure is a model of φ, then it is called a herbrand model
- Process of using herbrand structure

  Φ |= φ where Φ is set of φ1…φk

  We check $\phi 1 \wedge . . . \wedge \phi k \wedge \neg\phi$ is unsatisfiable

  Transform it into skolem normal form, now we only need to check herbrand structures
- In herbrand structure S, every ground term is interpreted "as itself": S(t) = t for all ground terms t

  Only α_p for p ∈ Δ is still "open"
- Herbrand structure are sufficient for unsatisfication

  Let Φ ⊆F(Σ, Δ, V) be a set of formulas in skolem normal form. Then Φ is satisfiable iff Φ has a herbrand model.
- If Φ contains formulas that are not in skolem normal form, then Φ could be satisfiable, although it has no herbrand model
- To check a formula in skolem normal form whether has a herbrand model, we reduce this problem from predicate logic to propositional logic. Replace all variables by all possible ground terms
- Herbrand expansion of a formula

  Let ϕ ∈ F(Σ,Δ, V) be in skolem normal form, i.e. $\phi = \forall X1, . . . ,Xn\ \psi$

  Then the herbrand expansion of ϕ is defined as:

  $E(\phi) = \{ \psi\ [X1/t1, . . . ,Xn/tn] \mid t1, . . . , tn \in \tau(\Sigma)\}$

  这个 expansion 就是上面说的 predicate 断言变 propositional 命题

  E(ϕ)是个大集合，里面有的 unsatisfiable 的 instance 就是 query 的解，整个也就是 unsatisfiable
- Satisfiability of herbrand expansion

  Let ϕ be a formula in skolem normal form, then ϕ is satisfiable iff E(ϕ) is satisfiable
- We generate herbrand expansion { ψ1, ψ2,…} step by step, first{ ψ1}, check, if not stop, then { ψ1, ψ2}, then on and on(大括号内都是且的关系)

- Compactness of propositional logic
  If a set of Φ of formulas is unsatisfiable, then Φ has a finite subset which is unsatisfiable(i.e. our algorithm is sound and complete)
- Algorithm of Gilmore
  1. Input: $\phi_1 \wedge \ldots \wedge \phi_k \wedge \neg\phi$
  2. Transform to skolem normal form ψ
  3. do herbrand expansion, choose an enumeration $\{\psi_1, \psi_2, \ldots\} = E(\psi)$, replace all atomic formulas by propositional variables
  4. Output true if any unsatisfiable: check $\psi_1, \psi_1 \wedge \psi_2, \psi_1 \wedge \psi_2 \wedge \psi_3, \ldots$

  This is sound and complete, but semi-decision, so when $\{\phi_1 \wedge \ldots \wedge \phi_k\} \not\models \phi$,does not necessarily terminate

3.3 Ground resolution

- To check a formula in skolem normal form($\forall X_1, \ldots, X_n\ \psi$) for unsatisfiability by resolution, one first transform ψ into conjunctive normal form, such formulas can then be represented as clause sets
- CNF, clause, literal
  A formula ψ is in conjunctive normal form iff it is quantifier-free and has the form:
  $(L_{1,1} \vee \ldots \vee L_{1,n_1}) \wedge \ldots \wedge (L_{m,1} \vee \ldots \vee L_{m,n_m})$
  Here, $L_{i,j}$ are literals, i.e., atomic formulas or negated atomic formulas of the form $p(t_1, \ldots, t_k)$ or $\neg p(t_1, \ldots, t_k)$.

  For every literal L, its negation $\overline{L}$ is defined as:

$$\overline{L} = \begin{cases} \neg A, & L = A \in At(\Sigma, \Delta, V) \\ A, & L = \neg A \text{ with } A \in At(\Sigma, \Delta, V) \end{cases}$$

  A clause is a set of literals and it represents the universally quantified disjunction of the literals.
  A clause set represents the conjunction of its clauses.
  So every formula ψ in CNF can be represented as a clause set:
  $K(\psi) = \{\{L_{1,1}, \ldots, L_{1,n_1}\}, \ldots, \{L_{m,1}, \ldots, L_{m,n_m}\}\}$.
  The empty clause is denoted as □, it is defined as unsatisfiable
- Transformation to CNF
  Every quantifier-free formula ψ can be transformed into an equivalent formula ψ' in CNF automatically
  Process:
  First replace all sub-formulas $\psi_1 \leftrightarrow \psi_2$ in ψ by $\psi_1 \rightarrow \psi_2 \wedge \psi_2 \rightarrow \psi_1$
  Then replace all sub-formulas $\psi_1 \rightarrow \psi_2$ in ψ by $\neg \psi_1 \vee \psi_2$
  Then use algorithm CNF:
  Input: ψ(quantifier-free, without ↔ or →)
  Output: formula in CNF
  1. if ψ is an atomic formula, then return ψ.
  2. if$\psi = \psi_1 \wedge \psi_2$, then return $CNF(\psi_1) \wedge CNF(\psi_2)$.

3. if$\psi = \psi 1 \vee \psi 2$, then compute CNF(ψ1) =$\wedge_{i\in\{1,\dots,m1\}} \psi_i'$ and CNF(ψ2) =$\wedge_{j\in\{1,\dots,m2\}} \psi_j''$. return $\wedge_{i\in\{1,\dots,m1\},j\in\{1,\dots,m2\}} \psi_i' \vee \psi_j''$(其实就是(…)∨(…)用了下 distributing law)
4. if $\psi = \neg\psi 1$, then compute CNF(ψ1) =$\wedge_{i\in\{1,\dots,m\}}(\vee_{j\in\{1,\dots,ni\}} Li,j)$.这里 n 有下标 i 是因为内部 literal 的个数视不同 clause 而定。

   De Morgan's law :$\vee_{i\in\{1,\dots,m\}}(\wedge_{j\in\{1,\dots,ni\}} \overline{Li.j})$.

   Use distributing law and return $\wedge_{j1\in\{1,\dots,n1\},\dots,jm\in\{1,\dots,nm\}} (\overline{L1.j_1} \vee \dots \vee \overline{Lm.j_m})$

- propositional resolution(after transform into CNF, we now use algorithm to check)
  Let K1, K2 be 2 clauses without variables.

  Then the clause R is a resolvent of K1 and K2 iff there exists a L∈ K1 with $\overline{L}$ ∈ K2 and

  R = (K1 \ {L}) ∪ (K2 \ { $\overline{L}$ })(注意是单个 L，所以一次只能消一个)

  For a clause set K we define
  Res(K) = K ∪ {R | R is resolvent of 2 clauses from K}.
  $Res^0(K) = K$
  $Res^{n+1}(K) = Res(Res^n(K))$ for all $n \geq 0$
  $Res^*(K) = \cup_{n\geq 0} Res^n(K)$
- construct Res*(K) until one obtains □, which means that K is unsatisfiable
  □∈ Res*(K) iff there is a sequence of clauses K1,…Km with Km = □ where for all 1≤i≤m, we have Ki∈ K or Ki is a resolvent of Kj, Kk for j,k < i
- propositional resolution lemma
  let K be a set of clauses without variables. If K1,K2 ∈ K and R is a resolvent of K1 and K2, then K and K∪{R} are equivalent
- soundness and completeness of proposition resolution
  let K be a set of clauses without variables, then K is unsatisfiable iff □∈ Res*(K)
- ground resolution algorithm
  1. formula $\phi 1 \wedge \dots \wedge \phi k \wedge \neg\phi$
  2. Transform into Skolem normal form $\forall X1, \dots ,Xn\ \psi$
  3. Transform ψ into CNF resp. into clause set K(ψ).(CNF;clause set 两者间是转写一下就成了)
  4. Choose an enumeration {K1,K2, . . .} of all ground instances of the clauses from K(ψ).(需要进一步提高的是这一步,K1,K2 各是一个 clause)
  5. Compute Res*({K1,K2}), Res*({K1,K2,K3}), . . . If one of these sets contains □, stop and return "true".
- Instantiation can unify several literals of the same clause
  Can use several different instantiations of the same clause
  就是上面那些 K1，K2，实例化以后可能变一样了，就会 unify，某个 clause 多实例化就有更大威力，不过 clause 是整个实例化，不能拆开用。

3.4 Resolution in predicate logic

- Unification

A clause K ={L1,…,Ln} is unifiable iff there exists a substitution σ with σ(L1) = . . . = σ(Ln) (i.e. |σ(K)| = 1 转换完了以后只剩下一个了)
such a substitution is called a unifier of K
a unifier σ is most general unifier(mgu) of K iff for every unifier σ′ of K there exists a substitution δ such that σ′ = δ ∘ σ (就是先 σ 后可以变成它)

- If a clause if unifiable, then it has a mgu, the mgu is unique up to variable renaming(i.e.不算变量重命名的话)
- Unification algorithm
  1. Let σ = ∅ be the "identical" substitution.
  2. If |σ(K)| = 1, then stop and return σ.
  3. Otherwise, check all σ(Li) in parallel from left to right, until there are different symbols in two literals.
  4. If none of these symbols is a variable, then stop with clash failure(就是不可能有 unifier).
  5. Otherwise, let X be the variable and t be the subterm in the other literal. If X occurs in t, then stop with occur failure.(因为是对于单个 clause 内的，所以同样的变量设置相同，所以 occur 这种情况不能 unify)
  6. Otherwise, let σ = {X/t} ∘ σ and go back to step 2.

  简而言之，就是找不同，可以的话就把不同的变量设置为另一个包含的东西
- Termination and soundness of unification algorithm
  The unification algorithm terminates for every clause K ≠ □ and it is sound, i.e., it computes a mgu for K iff K is unifiable.
- Resolution in predicate logic(求 resolvent 的一步，和 algorithm 不同)
  let K1,K2 be clauses, then R is a resolvent of K1 and K2 iff:
  1. there exist variable renaming v_1, v_2 such that v_1(K1) and v_2(K2) have no common variables.(注意是 renaming，所以单射，所以 X/U,U/X)
  2. there are L1, . . . , Lm ∈ v_1(K1) and L′1, . . . , L′n∈ v_2(K2) with n,m ≥ 1, such that $\overline{\text{L1}}$, . . . , $\overline{\text{Lm}}$,L′1 , . . . , L′n} is unifiable with a mgu σ.(前者的反和后者可以 unify，就意味着可以相反消除 resolve，下式这个大并集是空集的话，σ 就是解)
     (这里 n,m 可以不同，如果 n=m=1 就是 binary resolution)
     (这里 unifiable 就是 substitution，不需要 renaming 那种单射了)
  3. R = σ((v_1(K1) \ {L1, . . . ,Lm}) ∪ (v_2(K2) \ {L′1, . . . , L′n}))

  For a clause set K we define
  Res(K) = K ∪ {R | R is resolvent of 2 clauses from K}.
  $Res^0(K) = K$
  $Res^{n+1}(K) = Res(Res^n(K))$ for all n ≥0
  $Res^*(K) = \cup_{n\geq 0} Res^n(K)$
  这个和 propositional resolution 是一样的
- Predicate resolution lemma
  Let K be a set of clauses, if K1,K2 ∈ K and R is resolvent of K1 and K2, then K and K∪{R} are equivalent.
- Lifting lemma(from propositional to predicate)

Let K1,K2 be 2 clauses, let K1',K2' be ground instances of K1 and K2. If R' is a propositional resolvent of K1' and K2', then there exists a resolvent R of K1 and K2 such that R' is a ground instance of R.(意思就是 ground instance 时能使相同消掉，那么就有一个替代能使相同)

- Soundness and completeness for predicate resolution
  Let K be a set of clauses, then K is unsatisfiable iff □∈ Res*(K)
- Resolution algorithm:
  1. formula ϕ1 ∧ . . . ∧ ϕk ∧ ¬ϕ
  2. Transform into Skolem normal form ∀X1, . . . ,Xn ψ
  3. Transform ψ into CNF resp. into clause set K(ψ).(clause set 就是转写)
  4. Compute Res*(K(ψ)).
  If one finds □, stop and return "true".
  Otherwise, if Res*(K(ψ)) was computed completely,
  stop and return "false".
  (算法的毛病就是还是算所有可能的 resolution， 以后可以 improve efficiency)
- Summary:
  1st attempt: Gilmore's algorithm, has 2 drawbacks, a)how to instantiate variables? b)how to check unsatisfiability in propositional logic?
  2nd attempt: ground resolution algorithm, solves b)
  3rd attempt: resolution algorithm, also solves a)

3.5 Restrictions of resolution

- Summary:
  Linear resolution: maintains completeness
  Input resolution: not complete on all, but complete on Horn clauses
  SLD resolution: not complete on all, but complete on Horn clauses
- Logic programming only uses Horn clauses
- Linear resolution:
  Let ***K*** be a clause set. □ is derivable from a clause K in ***K*** by linear resolution iff there exists a sequence K1,...,Km with K1=K∈***K*** and Km = □ and for all 2≤i≤m we have, Ki is resolvent of Ki-1 and a clause from {K1,...Ki-1}∪***K***
  (意思就是必须用上一个 clause 以及之前的，包括自己生成的 clause 来 resolve)
- Soundness and completeness of linear resolution
  Let ***K*** be a clause set. Then K is unsatisfiable iff □ can be derived by linear resolution from some K∈***K***. (这里是说的 sound + complete，但是针对存在从 some K 开始 resolve)
  If ***K*** is a minimal unsatisfiable clause set(i.e. every proper subset is satisfiable), then □ can be derived by linear resolution from every K∈***K*** (这里是从任意开始，不过要求 ***K*** 是 minimal)
- Input resolution
  Let ***K*** be a clause set. Then □ is derivable from a clause K∈***K*** by input resolution iff there exists a sequence K1,...,Km with K1=K∈***K*** and Km = □ and for all 2≤i≤m we have, Ki is resolvent of Ki-1 and a clause from ***K*** (和之前的很像，只是进一步要求第二项来源于远古，而不能是自己合成的)

- Input resolution is sound but not complete, so use on Horn clauses
- Horn clause
  A clause K is a Horn clause iff it contains at most one positive literal (i.e., at most one of its literals is an atomic formula and the other literals are negated atomic formulas).
  A Horn clause is negative iff it only contains negative literals, i.e. it has the form {¬A1,…, ¬Ak} for atomic formulas A1,…,Ak.
  A Horn clause is definite iff it contains a positive literal, i.e., it has the from {B, ¬C1,…, ¬Cn} for atomic formulas B, C1,…Cn.
  (意思就是 horn clause 分两类，definite 恰有 1 个 positive，negative 没有 positive)
- A set of definite horn clauses corresponds to a conjunction of implications(→)
  {¬r, ¬p,s} equivalent to (r∧p→s)
- Connection between logic programming and Horn clause
  Facts: s ≙ {s}
  Rules: s:-r,p ≙ {s, ¬r, ¬p}
  Queries: ?- p,q. ≙ {¬p, ¬q}
- s:- ¬r is not allowed in pure logic programming, since it results in {s,r}, but Prolog has a form of negation
- propositional logic:
  general: decidable, NP-complete
  Horn clauses: polynomial time
  Predicate logic:
  General: undecidable, input resolution incomplete
  Horn clauses: undecidable, input resolution complete
- SLD resolution(improved from input resolution)
  Let ***K*** be a set of Horn clauses with ***K*** = $\boldsymbol{K}^d \uplus \boldsymbol{K}^n$, where $\boldsymbol{K}^d$ contains the definite Horn clauses from ***K*** and $\boldsymbol{K}^n$ contains the negative Horn clauses from ***K***. Then □ can be derived from K∈ $\boldsymbol{K}^n$ by SLD resolution iff there is a sequence K1,…,Km with K1=K∈$\boldsymbol{K}^n$ and Km = □ and for all 2⩽i⩽m we have, Ki is resolvent of Ki-1 and a clause from $\boldsymbol{K}^d$
  (说白了就是从一个全 negative 的开始，然后用原本的 definite 的去一个一个填)
- Soundness and completeness of SLD resolution
  Let ***K*** be a set of Horn clauses. Then ***K*** is unsatisfiable iff □ can be derived from some negative clause N∈***K*** by SLD resolution(注意是 some，所以是存在从某一个 K resolve 出□)
- Every set of definite Horn clauses is satisfiable, those unsatisfiable must contains at least one negative Horn clause (就是全是 rules 的世界是可以存在的)
- Binary resolution
  One more restriction on SLD resolution, in each clause, n=m=1
- Soundness and completeness of binary SLD resolution
  Let ***K*** be a set of Horn clauses.
  Then ***K*** is unsatisfiable iff □ can be derived from a negative clause N ∈ ***K*** by binary SLD resolution

4.1a Syntax and semantics of logic programs

- In logic programming, the order of the literals in a clause and the order of the clauses in a program is important(之前在 logic 里面都是 set，但是在编程里面，都是 sequence, can have same several times)
- Syntax of logic programs
  A non-empty finite set ***P*** of <u>definite</u> Horn clauses over a signature (Σ, Δ) is called a logic program over (Σ, Δ) (因为是由 fact 和 rule 组成，所以都是 definite)
  3 forms of program clauses(all formula below are atomic):
  Facts: {B}
  Rules: {B, ¬C1,…, ¬Cn}
  Query: { ¬A1,…, ¬An}
- Conjunction(and), disjunction(or)
- P called with query G = { ¬A1,…, ¬An}, then one has to prove
  P ⊨∃ X1,…,Xp A1 ∧,…, ∧An
  Which is equal to prove unsatisfiability of P∪{G}
- The answer substitution is obtained by composing the mgu's that were used in the proof, and then select those in query
  e.g. {C/sus} ∘ {W/ren} ∘ {Y/C,F/gerd} = {Y/sus, F/gerd,W/ren,C/sus}
  然后答案是{Y/sus}因为 query 里只有 Y 这个变量
  (左式是从右往左依次用的，连续替代，整理成最后结果)
- Define semantics of logic programs in 3 ways:
  1. Declarative
  2. Procedural(operational)
  3. Fixpoint(denotational)
- Declarative semantics
  Let P be a logic program and G = { ¬A1,…, ¬An} be a query. Then the declarative semantics of P with respect to G is
  D[P,G]= {σ(A1 ∧ . . . ∧ Ak) | P ⊨ σ(A1 ∧ . . . ∧ Ak), σ is a ground substitution}.
  (如果解有多个，那么 D 就包含多个，比如{fatherOf(g,s), fatherOf(q,n)})
- Procedural semantics
  Let P be a logic program.
  A <u>configuration</u> is a pair (G,σ) where G is a (query or □) and σ is a substitution
  We have a <u>computation step</u> (G1, σ1) $\vdash_P$ (G2, σ2) iff
  1. G1 = { ¬A1,…, ¬Ak}, k≥1
  2. There exists a K∈ P and a variable renaming ν such that
     ν(K) = {B, ¬C1, . . . , ¬Cn} with n≥0 and
     ν(K) is variable-disjoint from G1 and RANGE(σ1)
     there is an 1≤i≤k such that Ai and B are unifiable with mgu σ
  3. G2 = σ({¬A1, . . . , ¬Ai−1,¬C1, . . . , ¬Cn, ¬Ai+1, . . . ,¬Ak})(这里就是少了 Ai 和 B，中和掉了)
  4. σ2 = σ ∘ σ1

  A <u>computation</u> of P for the query G is a finite or infinite sequence
  (G,∅) $\vdash_P$ (G1, σ1) $\vdash_P$ (G2, σ2) $\vdash_P$ . . .

A computation that starts with (G, ∅) (for G = { ¬A1,..., ¬Ak}) and ends with (□,σ) is successful with the solution σ (A1 ∧. . .∧Ak).
The <u>answer substitution</u> is σ restricted to the variables of G
Then the <u>procedural semantics</u> of P w.r.t. G is
P[P,G] = {σ′(A1 ∧ . . . ∧ Ak) | (G,∅) $\vdash^{+}_{P}$ (□, σ), (特殊推导符号指 transitive closure of $\vdash_{P}$)
σ′(A1∧. . .∧Ak) is a ground instance of σ(A1∧. . .∧Ak)}.
(上面这么一大段，其实就是 resolution 的含义)
(上面所用 SLD 和普通的有 3 个区别
1. standardized SLD resolution: only rename variables in the definite clauses
2. binary SLD
3. clauses are sequences of literals，所以可以重复，按顺序)

- e.g.
  G = {¬fatherOf(gerd, Y)}
  Answer substitution: {Y/susanne}
  P[P,G] = {fatherOf(gerd, susanne)}(所以最后 P 这个语义就似乎解)
- two undecided:
  choose next program clause K to use: influence sequence of results
  choose next literal Ai in the query for resolution: influence termination behavior
- <u>equivalence of declarative and procedural semantics</u>
  let ***P*** be a logic program and G be a query, then we have D[***P***, G] = P[***P***,G]

4.1b Syntax and semantics of logic programs

- fixpoint is f(x) = x，和 y=x 的交点
- <u>trans$_{\boldsymbol{P}}$</u>
  let ***P*** be a logic program over a signature (Σ, Δ), then trans$_{\boldsymbol{P}}$:Pot(At(Σ,Δ,∅)) → Pot(At(Σ,Δ,∅)) (这个 Pot 意思是 set of all sets of ground atomic formulas,感觉这里就是写下 domain, range 的空间)is defined as follows:
  trans$_{\boldsymbol{P}}$(M) = M∪{A′ | {A′,¬B′1, . . . ,¬B′n} is ground instance of a clause {A,¬B1, . . . ,¬Bn} ∈ ***P*** and B′1, . . . ,B′n∈M }
  (其实意思很简单，就是在 M 这些已知的 ground instances 上面，根据 rule，再进一步得到的新集合)
- Then we define:
  $M_{\boldsymbol{P}} = \emptyset \cup trans_{\boldsymbol{P}}(\emptyset) \cup trans^2_{\boldsymbol{P}}(\emptyset) \cup trans^3_{\boldsymbol{P}}(\emptyset) \cup \ldots = \cup_{i\in N}\, trans^i_{\boldsymbol{P}}(\emptyset)$
  (这个 M$_{\boldsymbol{P}}$ 就是所有可以得到的 true statements 的闭包)
  (实际运用中，我们有可能有限步整个集合就稳定不变了)
  (M$_{\boldsymbol{P}}$ 就是我们相对于 trans$_{\boldsymbol{P}}$ 函数的 fixpoint，而且是 least fixpoint,也就是所有别的 fixpoint 的子集)
- Properties of ⊆
  Reflexive: M1⊆M1
  Antisymmetric: M1⊆M2 and M2⊆M1 implies M1 = M2
  Transitive: M1⊆M2 and M2⊆M3 implies M1⊆M3
  Thus ⊆ is a reflexive ordering
- A reflexive ordering is complete iff

1. It has a smallest element
2. Every chain has a least upper bound

- Completeness of ⊆
  The relation ⊆ is complete on Pot(At(Σ,Δ,∅)):
  The smallest set is ∅ and every chain M0⊆M1⊆M2⊆… has the least upper bound:
  M = $\cup_{i\in N}$ Mi
- Properties of the function trans$_{\boldsymbol{P}}$
  Monotonic: if M1⊆M2 then trans$_{\boldsymbol{P}}$(M1) ⊆ trans$_{\boldsymbol{P}}$(M2)
  Continuous: for every chain M0⊆M1⊆M2⊆…, we have trans$_{\boldsymbol{P}}$($\cup_{i\in N}$Mi)=$\cup_{i\in N}$ trans$_{\boldsymbol{P}}$(Mi)
- <u>Fixpoint theorem</u>(由于之前 properties of ⊆ and trans$_{\boldsymbol{P}}$)
  For every logic program ***P***, trans$_{\boldsymbol{P}}$ has a least fixpoint lfp(trans$_{\boldsymbol{P}}$).
  We have lfp(tans$_{\boldsymbol{P}}$) = ∪ $_{i\in N}$ trans$^{i}_{\boldsymbol{P}}$(∅)
- Fixpoint semantics
  Let ***P*** be a logic program, let G = {¬A1, . . . , ¬Ak} be a query. Then the fixpoint semantics of ***P*** w.r.t. the query G is:
  F[***P***,G]= {σ(A1 ∧ . . . ∧ Ak) | σ(Ai) ∈ lfp(trans$_{\boldsymbol{P}}$) for all 1≤i≤k }.
  (说白了意思就是所有 clauses 用这个 substitution 过后都能在 lfp 里面找到自己 true)
- <u>Equivalence of fixpoint semantics to the other semantics</u>
  Let ***P*** be a logic program, let G be a query. Then F[***P***,G] = D[***P***,G] = P[***P***,G]

4.2 Universality of logic programming

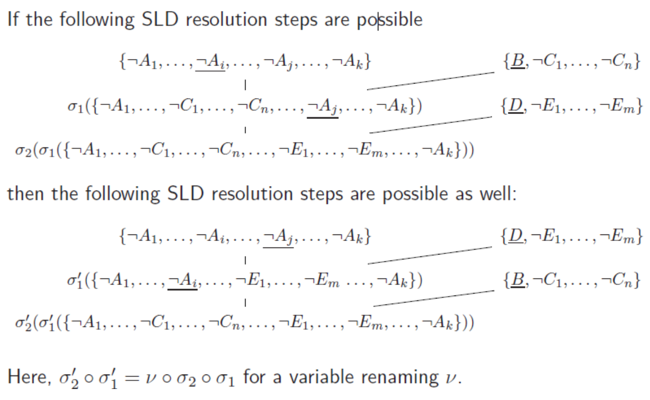
- Logic programming is a universal programming language, and it can compute all computable functions
- Computable functions: Turing machines = Lambda terms = μ-recursive functions
- μ-recursive functions
  the set of μ-recursive functions is the smallest set of functions with:
  1. for all n∈ N, the function null$_n$: N^n→N with
     null$_n$(k1, . . . , kn) = 0

  2. succ(k) = k + 1
  3. proj$_{n,i}$(k1, . . . , kn) = ki
  4. if f and f1,…,fm are μ-recursive, then g is also μ-recursive:
     g(k1, . . . , kn) = f(f1(k1, . . . ,kn), . . . , fm(k1, . . . , kn))
     (这就是 composition，输入用多个 function 处理完然后集成为一)
  5. if f and g are μ-recursive, then h is also:
     h(k1, . . . , kn, 0) = f(k1, . . . , kn)
     h(k1, . . . , kn, k + 1) = g(k1, . . . , kn, k, h(k1, . . . , kn, k))
     (这个是 primitive recursion，第一行是基本情况，后面递归调用，k 是计数)
  6. If f is, then g is.
     g(k1, . . . , kn) = k iff f(k1, . . . , kn, k) = 0 and
     for all 0 ≤k'< k,
     f(k1, . . . , kn, k') is defined and f(k1, . . . , kn, k') > 0
     If no such k, then g(k1,…kn) is undefined
     (这个是 minimalization)

- Functions constructed with 1-5 are <u>primitive recursive functions</u>
- Primitive recursive:
  Addition, subtraction, multiplication
  Non-primitive:
  Division
- Computing arithmetic functions by logic programs
  1. Every number k∈ N is represented by the term k∈ τ(Σ, V) with k = s(…s(0)…) where $0 \in \Sigma_0$ and $s \in \Sigma_1$
  2. A logic program P over (Σ ,Δ) computes a function f iff there is a predicate symbol f∈ Δ_{n+1} such that: f(k1,…kn) = k iff P |= f(k1,…kn,k)
- div(0,Y,0).
  div(s(X),s(Y),s(Z)) :- minus(X,Y,U), div(U,s(Y),Z).
  (这个是往上 round 的)
- <u>universality of logic programs</u>
  every µ-recursive function can be computed by a logic program

4.3 Indeterminism and evaluation

- exchangement lemma
  one can 1st do a resolution step with ¬Ai, then with ¬Aj, then one could exchange these 2 steps and obtain the same result



If the following SLD resolution steps are possible

$$\begin{array}{cc}
\{\neg A_1,\ldots,\underline{\neg A_i},\ldots,\neg A_j,\ldots,\neg A_k\} & \{\underline{B},\neg C_1,\ldots,\neg C_n\} \\
\sigma_1(\{\neg A_1,\ldots,\neg C_1,\ldots,\neg C_n,\ldots,\underline{\neg A_j},\ldots,\neg A_k\}) & \{\underline{D},\neg E_1,\ldots,\neg E_m\} \\
\sigma_2(\sigma_1(\{\neg A_1,\ldots,\neg C_1,\ldots,\neg C_n,\ldots,\neg E_1,\ldots,\neg E_m,\ldots,\neg A_k\})) &
\end{array}$$

then the following SLD resolution steps are possible as well:

$$\begin{array}{cc}
\{\neg A_1,\ldots,\neg A_i,\ldots,\underline{\neg A_j},\ldots,\neg A_k\} & \{\underline{D},\neg E_1,\ldots,\neg E_m\} \\
\sigma_1'(\{\neg A_1,\ldots,\underline{\neg A_i},\ldots,\neg E_1,\ldots,\neg E_m\ \ldots,\neg A_k\}) & \{\underline{B},\neg C_1,\ldots,\neg C_n\} \\
\sigma_2'(\sigma_1'(\{\neg A_1,\ldots,\neg C_1,\ldots,\neg C_n,\ldots,\neg E_1,\ldots,\neg E_m,\ldots,\neg A_k\})) &
\end{array}$$

Here, $\sigma_2' \circ \sigma_1' = \nu \circ \sigma_2 \circ \sigma_1$ for a variable renaming ν.

- Exchangement lemma implies that use arbitrary selection for next literals, every solutions can still be found. Prolog always selects the leftmost literal.
- <u>Canonical computation</u>
  A computation (G1, σ1) ⊢P (G2, σ2) ⊢P. . . is canonical iff in every resolution step one select the leftmost literal of query Gi
- Solving indeterminism 2
  Let P be a logic program, let G be a query, for every computation (G, ∅) $\vdash^{+}_{P}$ (□, σ), there exists a canonical computation (G, ∅) $\vdash^{+}_{P}$(□, σ’) of the same length, and σ and σ’ are the same up to variable renaming
  (其实就是说所有答案都可以通过 canonical 这样只选最左的来得到一个不会更差的结果)
- SLD tree
  Let P be a logic program, let G be a query, the SLD tree of P w.r.t. G is a finite or infinite tree whose nodes are labeled with sequences of atomic formulas and whose edges are

labeled with substitutions. The SLD tree is the smallest tree with:
1.if G = {¬A1, . . . , ¬Ak}, then the root of the tree is labeled with A1, . . . ,Ak
2.if a node is labled with B1, . . . ,Bn and B1 is unifiable with the positive literals of k variable-ranamed program clauses K1, . . . ,Kk (where the clauses appear in this order in the program),then the node has k children. The i-th child is labeled with the atoms that result from a canonical computation step using resolution with Ki. So if this computation has the form ({¬B1, . . . , ¬Bn},∅) $\vdash_P$ ({¬C1, . . . ,¬Cm}, σ) , then the i–th child is labeled with C1, . . . ,Cm and the edge to this child is labeled with substitution σ (restricted to the variable in B1, . . . ,Bn).
(说白了就是构建树，从 query 在 root，然后从左到右依次展开可能的 unifiable 的 clause，也就是每次选 literal 定了，但是用的 clause 都会组成树，我们看从 root 到□的路径上的 substitution 就是解)

- Solving indeterminism 1(choose the clauses to unify)
  1. Breadth-first search: complete but inefficient
  2. Depth-first search: efficient if solution on the leftmost but not complete;programmers have to take care

- Prolog: depth-first search, stop as soon as □, continues when ';'. Prolog is not completely declarative, on has to consider its evaluation

5.1 Arithmetic

- Prolog features:
  Function predicate: start with lower-case, special symbols and strings in quotes
  Variable: start with upper-case, or start with _, or _
  Instantiation of _ will not show in answer substitution
  Allow overloading: p(a,b,s). vs p(a,b).
  Use a variant of unification without occur check, so equal(Y,f(Y)). Y=f(Y),(应该是 occur 的)
  Prolog has a predefined predicate for checked unification:
  ?-unify_with_occurs_check(Y,f(Y)).
  false.
- Arithmetic expression:
  Binary infix symols: +, -, *, //(integer 除法), **(指数),…
  Unary negation:-
  Aboves are symbolic, so equal(3,1+2). Return false.
  Predicates that evaluate arithmetic:
  <,>,=<(注意这个符号),>=,=:=(check equality),=\=(non-equality)
  用上面这些时，必须完全 instantiated
  is/2.
  ?- t1 is t2.(t2 need to be instantiated and evaluated)
  ?-2 is 1+1.
  true.
  ?-1+1 is 2.
  False.

?-X+1 is 2.
False.
?-Y is X+1, X is 3.
error

- = is unification(no evaluation), corresponds to equal(X, X). without occur check
  is has its right argument evaluated
  =:= has both sides evaluated
  == is syntactic equality(就看字面相不相同，只有 true or false)
- add(X,Y,Z):-Z is X+Y.
- number/1 show if is a number
  ?-number(2).
  True.
  ?-number(1+1).
  False.
  ?-X is 1+1, number(X).
  X=2.

5.2 Lists

- we can use nil∈Σ0 and con∈Σ2,
  len(con(3,con(2,nil)),Y).
- we can also use predefined [] and .
  .(t1, t2) = [t1|t2]
  .(t1, []) = [t1]
  .(t1, .(t2, .(t3, t))) = [t1, t2, t3|t]
  .(t1, .(t2, .(t3, []))) = [t1,t2,t3] = [t1,t2|[t3|[]]] = [t1|[t2,t3|[]]] etc.
  ?-[1,2] == [1|[2]].
  True.
  So syntactically identical
- list append,这个是按顺序 X 完全在 Y 前面
  app([],Y,Y).
  app([X|Xs],Y,[X|Zs]):-app(Xs,Y,Zs).

5.3 Operators

- 2+3 is identical to +(2,3)
- Use directives to declare operators:
  :-op(500,yfx,[+,-]).
  :- op(400,yfx,*).

  precedence between 0-1200, the smaller, the stronger binding
  xfx, yfx, xfy
  fx, fy

xf, yf
x is argument whose precedence is < that of f
y is argument whose precedence is ⩽ that of f

precedence of non-operator symbols and arguments is 0
yfx means same precedence can only be in the left.
Yfx association to left
(画树或者是打括号看式子，都是左边先合)
Xfy association to right
就是看 y 在哪一边
Xfx not allow like 1+2+3

- :-op(200,fy,-).
  So -2-3 stands for (-2)-3
- :-op(300,xfx,was) 也可以，就实现了语法树
  ?-Who was the secretary of the head of the department.
  Who = laura.

5.4 The cut predicate and negation

- Green cuts: influence efficiency + termination, but not the results. Removes the cuts still correct
- Red cuts: removes cuts obtains different results
- Meta-variables(treat function symbol the same for instantiation)

```
or(X,Y):-X.
or(X,Y):-Y.

?-or(X=3,Y=2).
X=3;
Y=2.

if(A,B,C) :- A, !, B.
if(A,B,C) :- C.
% if A then B else C
```

- Negation:
  a) Closed world assumption: not derived is false
  b) For a statement cannot be derived, this can be detected in finite time

```
not(A) :- A, !, fail.
not(A).
% not is predefined, as well as \

even(0) :- !.
even(X) :- X > 0, !, X1 is X-1, not(even(X1)).
even(X) :- X1 is X+1, not(even(X1)).
```

5.5 Input and output

-
  ```
  X=3, write(X).
  3
  X=3.
  %write to stream, default screen

  write(‘=’).
  %可以打印字符串

  nl for new line
  ```
- Example1
  ```
  sqr(X,Y) :- Y is X*X.
  sqr :- nl, write(’Please enter a number or "stop": ’),
          nl, read(X),
          proc(X).
  proc(stop) :- !.
  proc(X) :- sqr(X,Y),
          write(’The square of ’), write(X), write(’ is ’), write(Y),
          sqr.
  ```

- Example2
  ```
  sqr(X,Y) :- Y is X*X.
  start :- nl, write(’Please enter the name of an input file: ’),
          nl, read(Inputfile),
          nl, write(’Please enter the name of an output file: ’),
          nl, read(Outputfile),
          see(Inputfile),
          tell(Outputfile),
          sqr,
          seen,
          told.
  sqr :- read(X),
          proc(X).
  proc(end_of_file) :- !.
  proc(X) :- sqr(X,Y),
          write(’The square of ’), write(X), write(’ is ’), write(Y),
          nl,
          sqr.
  ```

5.6.1 Meta-programming: terms & atomic formulas

- Some predefined predicate:
  var(t) is true iff t is an uninstantiated variable
  nonvar(t) is true iff t is not a variable:
  ?-X=2, nonvar(X).
  X=2.
  %别的是 true, false，但如果这么问，就是没有。

  atomic(t) is true iff t is a function/predicate symbol of arity 0 or a number
  ?-atomic(a).
  True.

  ?-atomic(X).
  False.

  ?-atomic(a(a)).
  False

  compound(t) is true iff t is a term or an atomic formula that is not just a symbol of arity 0 or a number or a variable(意思就是除了这些单个的)
  ?-compound(a).
  False

  ?-compound(a(a)).
  True

  ?-compound(3+2).
  True
- =..
  t =.. l iff l is the list representation of the term t

  ?- f(a,b) =.. L.
  L =[f,a,b].

  ?-1+2 =..L.
  L=[+,1,2].

  ?-T =.. [f].
  T=f
  %如果没有后面中括号就 error

  =.. only converts on the top level:
  ?- p(f(X),2,g(X,Y)) =.. L.

L = [p, f(X), 2, g(X,Y)].

- functor(t,f,n) is true iff f is the leading function symbol of the term t and the arity of f is n.
  (本质就是提取 t 这个 term 的函数，以及 arity 是多少)
  (也可以反其道而行之)
  ?-functor(T,g,3).
  T = g(_6508, _6510, _6512).

- arg(n,t,a) is true iff a is the n-th argument of the term t.(start with 1)
  ?- functor(D,date,3),
  arg(1,D,29),
  arg(2,D,june),
  arg(3,D,1982).
  D = date(29,june,1982).
  %这里意思就是确定 D 是 date(,,)，然后告诉 3 项中每一项各自是啥，所以就有 D
- 自己写 ground/1 where gound(t) is true iff t is a ground term(i.e. t has no variables)
  ```
  ground(T) :- nonvar(T),
          T =.. [Functor|Argumentlist],
          groundlist(Argumentlist).
  groundlist([]).
  groundlist([T|Ts]) :- ground(T), groundlist(Ts).
  ```
  %注意 groundlist 又调用了 ground，所以是层层递进，都不能有变量，每一层内用 list 来遍历

5.6.2 Meta-programming: manipulating programs

- Prolog read its own code during runtime using predefined predicate clause/2
  Clause(t1,t2) is true iff there is a program clause B:-C1,…Ck such that clause(t1,t2) unifies with clause(B,(C1,…Ck))
  (本质就是通过 unify 其实来看这个 predicate 的结构)
  times(_,0,0).
  times(X,Y,Z) :- Y > 0, Y1 is Y-1, times(X,Y1,Z1), Z is Z1+X.
  (上面这个是相乘的函数)
  ?- clause(times(X,Y,Z),Body).
  Y = 0, Z= 0 ,Body = true;
  Body = Y > 0, Y1 is Y-1, times(X,Y1,Z1), Z is Z1+X.
  (如上，Body 就会读出 times 这个 clause 的内容，单纯的 Body 赋值成:-右侧，但是左侧参数限制如果有也得写上)

- assert/1: assert(t) always succeeds and clause t is added at the end of program(可以加 fact，也可以加 rule)
  assert(square(X,Y):-times(X,X,Y)).
  True.

%注意，这种用到 is 的函数，不是能随意调用的

```
?- square(9,Y).
Y = 81 ;
Y = 81 ;
False.

?- square(X,64).
```

Error, because in times, Y is not instantiated before is.

- asserta/1: add at the beginning
  assertz/1: add at the end, like assert.
- Modify program need the predicate to be dynamic
  Use directive:
  `:- dynamic times/3.`

  Now we can:
  `?-asserta(times(X,1,X)).`
- retract/1 is used to remove clauses of dynamic predicates
  retract(t) succeeds iff a program clause unifies with t. the 1st clause is removed

  %这个可以删除掉 facts 和 rule

  ```
  ?- retract(times(X,Y,Z):-B).
  Y = Z, Z = 0,
  B = true ;
  B =    (Y>0, _11204 is Y+ -1, times(X, _11204, _11214), Z is _11214+X).
  ```

  %这个只能删 facts
  `?-retract(times(X,Y,Z)).`
- 可以编写带有 assert，retract 的代码，比如构建一个动态 table

  ```
  maketable :- L = [0,1,2,3,4,5,6,7,8,9],
               member(X,L),
               member(Y,L),
               Z is X * Y,
               assert(times(X,Y,Z)),
               fail.

  ?-maketable.
  ```

  %这样就有了 100 个 facts，上面 fail 是为了 backtrack
- findall is predefined.
  findall(t,g,l) is true iff one builds up the complete SLD tree for the query g and computes all answer subtitutions σ1,…σn. l is the list [σ1(t),...,σn(t)]

  %t 是要的东西,g 是 query,l 是结果

  ```
  ?- findall(vaterVon(gerd,K), vaterVon(gerd,K), L).
  L = [vaterVon(gerd, susanne), vaterVon(gerd, peter)].
  ```

```
%we can implement the findall by ourselves
findall(X, Anfrage, Xlist) :- Anfrage,
        assert(loesung(X)),
        fail
        ;
        sammleLoesungen(Xlist).
sammleLoesungen([X | Rest]) :- retract(loesung(X)),
        !,
        sammleLoesungen(Rest).
sammleLoesungen([]).
```
(本质就是先把所有答案给 assert，通过 fail 来得多所有结果，最后 retract 来集合)

- we can implement interpreters for prolog

```
prove(true,0) :- !.
prove((Goal1,Goal2),N) :- !, prove(Goal1,N1), prove(Goal2,N2),
        N is N1+N2.
prove(Goal,N) :- clause(Goal, Body), prove(Body,N1),
        N is N1+1.
```

(上面这个就是对 interpreter 加上了计数多少个 literal 要 prove)

6.1 Syntax and semantics of constraint logic programs

- <u>Constraints</u>
  constrains are atomic formulas over a certain subsignature. We use a special treatment for the predicate symbols =, true, fail
  <u>Constraint Signature</u>: (Σ ;Δ ;Σ';Δ') with
  1. true; fail ∈ Δ0 and = ∈Δ2
  2. Σ' ⊆ Σ and Δ' ⊆ Δ
  3. Δ' does not contain true, fail, or =

  <u>Constraints:</u>
  True, fail, t1=t2(t1,t2∈Σ)
  p(t1,…tn) with p∈ Δ' and ti ∈ Σ'
- Example:
  Σ'0 = Z
  Σ'1 = {−, abs}
  Σ'2 = {+,−, ∗, /, mod, min, max}
  Δ'2 = {#>=, #=<, #=, #\=, #>, #<}
  (subsignature 也叫 finite domain)
- Constraint theory:
  Let (Σ,Δ,Σ',Δ') be a constraint signature. If CT ⊆ F(Σ',Δ', V) is satisfiable and only contains closed formulas, then CT is called a constraint theory.
  CT is not even semi-decidable
- Syntax of CLP

A non-empty finite set of P of definite Horn clauses over a constraint signature (Σ,Δ,Σ′,Δ′) is a CLP, if {true} ∈ P, {X=X}∈ P, and for all other clauses {B, ¬C1, . . . , ¬Cn} ∈ P we have:
(a) if B = p(t1, . . . , tm), then p∈Δ′ ∪ {true, fail,=}.
(b) if Ci = p(t1, . . . , tm) and p ∈ Δ′, then tj ∈ τ (Σ′, V) for all 1 ≤ j ≤ m.
(b 也限定 all queries,这一条就是那个只有 sub 中的 predicate 才能用在 sub 的 term 上)

- 自己随手写 minus，注意不要用 is，也必须得拎出来写，不能只改 play(X,Y-1,Z+1).
  play(X,0,X).
  play(X,Y,Z):- A = Y-1,   play(X,A,U), Z = U-1.
- <u>Declarative semantics of CLP</u>
  Let P be a CLP, let CT be the constraint theory, let G= {¬A1, . . . , ¬Ak}, then the declarative semantics of P and CT w.r.t. G is :
  D[P,CT,G]= {σ(A1 ∧ . . . ∧ Ak) | P ∪ CT |= σ(A1 ∧ . . . ∧ Ak), σ is ground substitution}.
- LP is a special case of CLP.
  Let P be a CLP with Σ′ = ∅ und Δ′ = ∅, then for all queries G we have:
  D[P,∅,G] = D[P,G].
- CT is not handled by SLD-resolution, but by the black-box constraint solver.
- Equality between atoms

  Let A,B be atomic formulas, then we define the formula $\overline{A=B}$ as follows:

  1. $\overline{A=B}$ is the fail if A=p(…), B=q(…), p≠q
  2. $\overline{A=B}$ is true if A=B=p ∈ Δ0
  3. $\overline{A=B}$ is the formula s1 = t1 ∧ . . . ∧ sn = tn, if A = p(s1, . . . , sn) and B = p(t1, . . . , tn)
- Procedural semantics of CLP
  Let P be a CLP and let CT be a corresponding constraint theory.
  A configuration is a pair(G,CO), where G is a query or □, and CO is a conjunction of constraints.

There is a computation step $(G_1, CO_1) \vdash_{\mathcal{P}} (G_2, CO_2)$ iff

$G_1 = \{\neg A_1, \dots, \neg A_k\}$ with $k \geq 1$ and one of (A) or (B) holds:

(A) Some A_i is not a constraint. Then:
- there exists a $K \in \mathcal{P}$ with $\nu(K) = \{B, \neg C_1, \dots, \neg C_n\}$ such that
  - $\nu(K)$ has no common variables with G_1 or CO_1
  - $CT \cup \{\forall X\ X = X, \text{true}\} \models \exists\,(CO_1 \wedge \overline{A_i = B})$
- $G_2 = \{\neg A_1, \dots, \neg A_{i-1}, \neg C_1, \dots, \neg C_n, \neg A_{i+1}, \dots, \neg A_k\}$
- $CO_2 = CO_1 \wedge \overline{A_i = B}$

(B) Some A_i is a constraint. Then:
- $CT \cup \{\forall X\ X = X, \text{true}\} \models \exists\ CO_1 \wedge A_i$
- $G_2 = \{\neg A_1, \dots, \neg A_{i-1}, \neg A_{i+1}, \dots, \neg A_k\}$
- $CO_2 = CO_1 \wedge A_i$

$$P[\![\mathcal{P}, CT, G]\!] = \{\sigma(A_1 \wedge \dots \wedge A_k) \mid (G, \text{true}) \vdash^{+}_{\mathcal{P}} (\Box, CO),\ \sigma \text{ is ground substitution with } CT \cup \{\forall X\ X = X, \text{true}\} \models \sigma(CO)\}$$

- Equivalence of procedural and declarative semantics
  Let P be a CLP, let CT be a corresponding constraint theory, let G be a query, then:
  D[P,CT,G] = P[P,CT,G].
- Also use canonical computation(leftmost), and depth-first search in SLD tree.
- LP:
  fac(0,1).
  fac(X,Y) :- X > 0, X1 is X-1, fac(X1,Y1), Y is X*Y1.

  CLP:
  fact(0,1).
  fact(X,Y) :- X #> 0, X1 #= X-1, fact(X1,Y1), Y #= X*Y1.
  CLP 这个有优点,more efficient and bidirectional

  ?-fact(X,1). In LP
  X=0.
  找不到 fact(1,1)因为那时 X>0 还没 instantiate 没法判定

  ?-fact(X,1) in CLP, both X=0 and X=1 can be found.

6.2 CLP in prolog

- Predicate use_module is predefined
  :- use_module(library(clpfd)).

- Entailment is undecidable for constraint theory CT_{FD}, prolog approximates and only check path consistency of CO instead of satisfiability of CO.
- Path consistency
  let CO = $\phi 1 \wedge . . . \wedge \phi m$ be a conjunction of constraints with $\phi i \in At(\Sigma_{FD},\Delta_{FD}, V)$. let X1, . . . ,Xn be the variable of CO, let D1, . . . ,Dn be subsets of Z. We say that D1, . . . ,Dn are admissible domains for X1, . . . ,Xn w.r.t. CO iff for all constraints ϕi with $1 \le i \le m$ and all variables Xj with $1 \le j \le n$ we have: for all $aj \in Dj$ there exists $a1 \in D1, . . . , aj−1 \in Dj−1, aj+1 \in Dj+1, . . . , an \in Dn$ such that $CT_{FD} \models \phi i$ [X1/a1, . . . ,Xn/an]. If there exist admissible domains D1, . . . ,Dn which are all non-empty, then CO is path-consistent.
  (difference between path-consistency and satisfiability:path 把所有 constraints 单独看)
- 一般先把 Dj 都设置为 Z,之后通过迭代 constraints 来减小 domain Dj 到不变为止
  例子中显示，是循环查看 constraint，而且是针对不同 variable 挨个检查
  X1#> 5 ∧ X1#< X2 ∧ X2#< 9

	D1=Z,D2=Z
X1#> 5 ; X1	D1={6,7…},D2=Z
X1#< X2 ; X1	D1={6,7…},D2=Z
X1#< X2 ; X2	D1={6,7…},D2={7,8…}
X2#< 9 ; X2	D1={6,7…},D2={7,8}
X1#> 5 ; X1	D1={6,7},D2={7,8}

最后 D1={6,7},D2={7,8},所以成功，path-consistent

- Predefined predicates: in and label
  ?- X1 #> 5, X1 #< X2, X2 #< 9, label([X1,X2]).
  X1 = 6,
  X2 = 7 ;
  X1 = 6,
  X2 = 8 ;
  X1 = 7,
  X2 = 8.
  (label 只能用在有限的 admissible 域里，不用 label，prolog 就会返回一遍规则)
- Path-consistent, but unsatisfiable
  ?- X1 #> X2, X1 #=< X2.
  这个的域死活不变小，但是当然是不可行的
- N-queens problem in CLP

```
:- use module(library(clpfd)).
queens(N,L) :- length(L, N),
        L ins 1 .. N,
        all_different(L),
        safe(L),
        label(L).
safe([]).
safe([X|Xs]) :- safe_between(X, Xs, 1),
        safe(Xs).
safe_between(X, [], M).
safe_between(X, [Y|Ys], M) :- no_attack(X, Y, M),
        M1 #= M + 1,
        safe_between(X, Ys, M1).
no_attack(X, Y, N) :- X+N #\= Y, X-N #\= Y.
```

- Predefined:
  length, ins, all_different
- Negation: \+ can be use on one argument
  female(X) :- \+ male(X).

  别的时候\+ X=Y 等价 X\=Y

Help sheet

- times(X,Y,Z) :- Z = X*Y.
  times(X,X,64).
  False.

  times(X,Y,Z) :- Z =:= X*Y.
  times(X,X,64).

Error

times(X,Y,Z) :- Z is X*Y.
times(X,X,64).
Error

- times(X,Y,Z) :- Z #= X*Y, X in 1..Z, Y in 1..Z.
  times(X,X,64).
  X = 8.
- :- use_module(library(arithmetic)).
  :- op(400, xfx, ^^).
  :- arithmetic_function('^^'/2).
  ^^(X,1,X).
  ^^(X,Y,Z):- Y>1, B is Y-1,^^(X,B, T), Z is T*X.

  %上面这样就可以做到使用 3^^3,但是必须用在 is 之类的 evaluate 的右边，直接 add(3,2^^2,Y).是会报错的
- =：=会两边 evaluate 后比较，但是 X=:=3 不行，因为 X 还没有 instantiate。
- =是两边不 evaluate 就比较，或者不 evaluate 就赋值变量
- 不存在:=，或者说就应该是 is
- X is a. error 因为 is 要的是 arithmetic， 3 is 3. 返回 true
- 用 not(X)或者\+ X 都可以表示 X 是 false 的
- p([X])：-这种就是匹配单个元素的数列
- X 既不是 atomic 也不是 compound，只能用 var(X)来识别
- Y = [Xs]可以给 list 外面再加一层括号，所以写的时候注意去掉括号
- 在 clp 限定时，比起用 X#=<N,不如用 X in 0..N
- add([X|Xs],L):- 之后递归时用 Xs 就好，不要再套[]，因为 Xs 本身就是 list
- <u>greedy 200ky:</u>
  weight([] ,0).
  weight([person(W)| PS],R) :- weight(PS ,R2), R is R2 + W.
  dist([] ,[]).
  dist([person(W)| PS],[boat([person(W)|G])| BS]) :- dist(PS ,[boat(G)| BS]),weight(G,A), A + W =< 200 , !.
  dist([person(W)| PS],[boat([person(W)])| BS]) :- dist(PS ,BS).
- neq(0,s(_)).
  neq(s(_),0).
  neq(s(X),s(Y)):-neq(X,Y).
- coll(1,0):-!.
  coll(X, Z):- 0 =:= X mod 2, !, T is X/2, coll(T, N), Z is N+1.
  coll(X, Z):- T is X*3+1, coll(T, N), Z is N+1.
  第一行的叹号 cut 是必须的。而且那个式子必须放在第一行，否则停不下来。
- cover(X,R):- length(R, X), R ins 1..X, all_distinct(R), check(R), label(R).
  check([]).
  check([X,Y,Z|Xs]):- Z#= X+Y, check(Xs).
  注意第一行的 length，如果没有这个限制，就会没有长度，也就无从 check 到头。

- asserta()可以加 fact 或 rule，但是都没有句号
- 证明下面头尾是 iff

  $$\{\varphi_1, \ldots, \varphi_k\} \models \psi'$$

  iff for all interpretations I with $I \models \{\varphi_1, \ldots, \varphi_k\}$ we have $I \models \psi'$

  iff there is no interpretation I with $I \models \{\varphi_1, \ldots, \varphi_k\}$ and $I \models \neg\psi'$

  iff $\varphi_1 \wedge \ldots \wedge \varphi_k \wedge \neg\psi'$ is unsatisfiable
- Universality: 写常数 n 时应该是 $s^n(0)$，这种细节注意
- Domino:
  ```
  add([],p(_,_)).
  add([p(X,Y)], p(Y,_)).
  add([p(X,Y),p(Y,Z)|Xs], T):- add([p(Y,Z)|Xs], T).
  add([p(X,Y),p(W,Z)|Xs], T):- Y\=W, !, T=p(Y,W).
  ```
- // and div are integer divisions

  / is for float, only integer when arguments and result are exact

-
  ```
  fib(1,0):-!.
  fib(2,1):-!.
  fib(N, R):- T is N -1, P is N-2, fib(T,L), fib(P, Q), R is L+Q.
  ```
  叹号 cut 是必须有的
- a sequence start from 5 and -1+2-3+4…
  ```
  nth(1,5):-!.
  nth(N,K):- T is N-1, nth(T, P), M is N mod 2, K is P+2*M*T-T.
  ```
- s 下的相加
  ```
  ins(M,0,M):-!.
  ins(M,s(Y),s(Z)):- ins(M, Y, Z).
  ```
  注意，第一行的叹号没有也可以运行，但是会返回结果，然后等待输入分号，之后返回 false，而用了 cut 就直接返回结果，句号。
- 获得那些单独的元素
  ```
  nodupl(X,R):- nohelp(X,R,X).
  nohelp([],[],Y).
  nohelp([X|Xs], [X|Rs], Y):- single(X,Y), !, nohelp(Xs, Rs, Y).
  nohelp([X|Xs], Rs, Y):- nohelp(Xs, Rs, Y).

  single(X,Y):- count(X,Y, R), R<2.
  count(X,[], 0):-!.
  count(X, [X|L], R):- !, count(X, L, Z), R is Z+1.
  count(X, [Y|L], R):- count(X, L, R).
  ```
  重点注意就是基本情况一定要写，比如 nohelp([],[],Y).

-
  ```
  tetration(X, 1, X):-!.
  tetration(X,Y,Z):- Y1 is Y-1, tetration(X, Y1, P), Z is P**X.
  ```
  第一行加叹号，或者第二行加 Y>1.