第 3 章

逻辑推理与专家系统

知识表示完成的是人类知识在机器中的体系化存储和管理，并为实现高效的推理奠定基础。本章将重点关注机器如何完成这样的推理。早期的推理方法大多数来源于定理自动证明，主要任务是基于规则形式的公理系统，查询新的结论是否正确。本章将首先介绍命题逻辑，从中可以看到机器推理的基本过程。然后将进入一阶逻辑。一阶逻辑又称为谓词逻辑、谓词演算等，出发点是为了克服命题逻辑的局限性。它具有更加通用的知识表示能力和更强的推理能力。逻辑推理方法在专家系统中的应用效果显著，成功减小了人工智能研究在第一次“AI 之冬”中受到的冲击。本章的最后将介绍专家系统的相关内容。

3.1 命题逻辑

3.1.1 命题的基本概念及其运算

命题逻辑是逻辑系统中最简单的一类，它采用命题来表示知识。命题的基本形式是陈述句。例如，“小张的 AI 成绩比小李高”“今天是晴天”等。命题有相应的语法。比如我们常常说“小王的父亲是老王”或“老王是小王的父亲”，而不说“小王老王父亲是”。命题还有自身的语义，即该命题实际所指代的事物和关系。比如对于命题“这张桌子是实木的”，我们所理解的含义不会变成“天上的星星在闪光”。我们为每一个命题赋予一个二元取值（记为{真，假}、{T，F}或{0，1}），称为真值，以表示其语义上的正确性。形式化的写法是

$$p \rightarrow \{0,1\}$$

p 代表命题语句，0 代表假，1 代表真。命题具有三个最直接的判断特征：①是陈述句；②能判断真假；③真值唯一。思考以下几个句子并判断是否为命题：

① 请不要吸烟！

② x 大于 y。

③ $\sqrt{2}$ 是无理数。

④ 今天是星期二。

⑤ 11+1=100。

⑥ 我正在说谎。

逐一分析之。显然，第 1 句是祈使句而非陈述句，故不是命题。第 2 句中，x 和 y 是变量，无法判断真假，也不是命题。第 3 句符合三个标准，是命题。第 4 句和第 5 句的真值需要视实际情况而定，但仍然是确定的，因此是命题。最后一句出现了矛盾，是悖论，其真值是不确定的，因此不是命题。

关于命题的真值需要说明两点。首先，真值通常代表的是命题语义上的真假，而命题语句只是一种符号表示，二者并无关系。不同环境下的同一个命题可能有不同的真值。例如在不同的天气状况下，命题“今天是晴天”的真值可能会发生改变（但只能取 T/F 中的一个）。其次，真值是人为赋予的。因此理论上讲，真值可以完全脱离实际情况（在上一章的本体一节，我们也看到类似的无关性）。例如即使现实中是晴天，仍然可以赋予命题“今天是晴天”的真值为 F（虽然几乎没有人这样做）。

给定一组命题，为每一条命题都赋予一个真值。它们就构成了一个可能的“世界”，称为模型。假设命题集合中共有 n 条语句，分别将其真值赋为真或假，可以得到 2^n 个模型。通常我们将这组命题的真值都赋为真，称为公理，并以此作为知识库证明另外的命题是否为真。通过推理被证明为真的命题称为定理。如果在每一个使命题 p（可以是公理也可以是定理）为真的模型中，都有命题 q 为真，则称 p 蕴含 q，记为

$$p \models q$$

简单的、不能再分的命题称为简单命题或命题词。由简单命题通过逻辑连接词构造而成的命题称为复合命题。常用的逻辑连接词有以下五种：

¬（非）：表示一个命题的否定。

∧（合取）：两个命题 p 和 q 的合取式表示为 $p \wedge q$。

∨（析取）：两个命题 p 和 q 的析取式表示为 $p \vee q$。

⇒（或→）（蕴含）：如 $p \Rightarrow q$ 称为蕴含式，p 称为前提（前项），q 称为结论（后项）。蕴含式又称为规则或“If…Then”，也用符号⊃或→表示。

⇔（或↔）（双向蕴含）：$p \Leftrightarrow q$ 表示两个命题 p 和 q 等价，也记为 $p \equiv q$。

五种运算符中，¬ 具有最高优先级。双向蕴含符号“⇔”意为“当且仅当”，即充分必要条件。用 T 表示命题为真，F 表示命题为假，则复合命题的真值可由表 3-1 确定。需特别注意，当 p 为假时，蕴含式 $p \Rightarrow q$ 是一个永真式（恒为真）。这可以理解为，当条件为假时，我们对结论不做评论。常见的逻辑符号还包括⊨（或⊢）。它表示“推论出”或“推导出”。$p \models q$ 表示以 p 为条件能够推导出 q。

表 3-1　复合命题的真值表

p	q	$\neg p$	$p \wedge q$	$p \vee q$	$p \Rightarrow q$	$p \Leftrightarrow q$
F	F	T	F	F	T	T
F	T	T	F	T	T	F
T	F	F	F	T	F	F
T	T	F	T	T	T	T

复合命题还有一些常见运算法则，列举如下：

$p \wedge q \equiv q \wedge p$，$p \vee q \equiv q \vee p$（交换律）

$(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$，$(p \vee q) \vee r \equiv p \vee (q \vee r)$（结合律）

$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$，$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$（分配律）

$\neg(\neg p) \equiv p$（二次取非消去）

$p \Rightarrow q \equiv \neg q \Rightarrow \neg p$（逆否定理）

$p \Rightarrow q \equiv \neg p \vee q$（蕴含消去）

$p \Leftrightarrow q \equiv (p \Rightarrow q) \wedge (q \Rightarrow p)$（双向蕴含消去）

$\neg(p \wedge q) \equiv \neg p \vee \neg q$，$\neg(p \vee q) \equiv \neg p \wedge \neg q$（德摩根定律）

如果一个命题在某个模型中为真，则称该命题是可满足的。如果一个命题在所有模型中都为真，则称该命题是有效的。命题 p 是有效的，当且仅当 $\neg p$ 是不可满足的。因此，对蕴含消去法则，要证明 $p \Rightarrow q$，只需证明 $\neg(\neg p \vee q)$ 即 $p \wedge \neg q$ 是不可满足的，这正是反证法的理论基础。

3.1.2　命题逻辑的推理规则

逻辑系统的核心是推理。命题逻辑的推理规则有

假言规则：

$$\frac{p \Rightarrow q,\ p}{q}$$

式中，横线上方为给定条件，下方为结论。假言规则的含义是，给出任意的 $p \Rightarrow q$ 和条件 p，则可以得到结论 q。

消去规则：

$$\frac{p \wedge q}{p},\quad \frac{p \wedge q}{q}$$

上式意为给定合取式 $p \wedge q$，则可以得到其中每一个子句为真的结论。

逻辑等价规则：所有的复合命题运算法则都可以用来作为推理规则。例如双向蕴含消去：

$$\frac{p \Leftrightarrow q}{(p \Rightarrow q) \wedge (q \Rightarrow p)},\quad \frac{(p \Rightarrow q) \wedge (q \Rightarrow p)}{p \Leftrightarrow q}$$

蕴含消去：

$$\frac{p \Rightarrow q}{\neg p \vee q}$$

例 3-1（罪犯识别问题）　某地发生一起命案，公安部门介入调查，锁定了四个嫌疑人甲、乙、丙、丁。四个侦查员分别得到以下四条线索：若甲不是凶手，则乙或丙是凶手；若丙是凶手，则甲也是凶手；若乙是凶手，则丁也是凶手；甲不是凶手。请问丁是否是凶手？

解：将命题用字母表示，a："甲是凶手"，b："乙是凶手"，c："丙是凶手"，d："丁是凶手"。知识库为

$$\neg a \Rightarrow b \vee c \tag{3-1-1}$$

$$c \Rightarrow a \tag{3-1-2}$$

$$b \Rightarrow d \tag{3-1-3}$$

$$\neg a \tag{3-1-4}$$

一个简单的推理过程如下。对式(3-1-1)和式(3-1-4)用假言推理得到

$$\frac{\neg a \Rightarrow b \vee c, \neg a}{b \vee c} \tag{3-1-5}$$

对式(3-1-2)用逆否定理得到

$$\frac{c \Rightarrow a}{\neg a \Rightarrow \neg c} \tag{3-1-6}$$

对式(3-1-6)再用假言推理得到

$$\frac{\neg a \Rightarrow \neg c, \neg a}{\neg c} \tag{3-1-7}$$

再对式(3-1-5)的结论,反向用蕴含消去得到

$$\frac{b \vee c}{\neg c \Rightarrow b} \tag{3-1-8}$$

因此根据式(3-1-7)和式(3-1-8)的结论,用假言推理得到

$$\frac{\neg c \Rightarrow b, \neg c}{b} \tag{3-1-9}$$

最后由式(3-1-3)和式(3-1-9)的结论,用假言推理得到

$$\frac{b \Rightarrow d, b}{d}$$

因此丁是凶手。

3.1.3 鲁滨逊归结原理

前一小节中,罪犯识别问题的推理过程是手动给出的。对于包含大量命题的知识库而言,这样的推理是低效的。1965 年美国人鲁滨逊(Robinson)提出的归结原理给出了一种自动推理的可行方法[1]。归结原理的基本思想是,若要证明知识库 KB 蕴含结论 p,只需证明 $KB \wedge \neg p$ 是不可满足的。具体而言,归结操作是

$$\frac{p \vee q_1 \vee q_2 \vee \cdots \vee q_m, \neg p}{q_1 \vee q_2 \vee \cdots \vee q_m}$$

即将互为相反原子子句 p 和 $\neg p$ 消去,剩下的析取式作为结论。这称为元归结。进一步推广到析取式的情况称为全归结

$$\frac{p_1 \vee \cdots \vee \neg p_k, p_k \vee q_1 \vee \cdots \vee q_n}{p_1 \vee \cdots \vee p_{k-1} \vee q_1 \vee \cdots \vee q_n}$$

即将两个析取式中的单个相反文字消去,剩余的部分作为新的析取式。注意,全归结仅限于消去单个文字。对于多个相反的文字不适用。应用归结原理证明定理时,首先将结论转换成否定形式,其次将知识库中的相关语句化为析取式,然后用否定的结论与知识库中的相关语句做归结操作,不断消去相反子句。这个过程最终会出现两种可能的结果:(1)归结最终得到空集,那么上一步必定是形如 p 和 $\neg p$ 的两个子句归结,这导致矛盾,表明原知识库能够推导出结论;(2)归结最终无法得到空集,表明无法推出矛盾,原知识库不能推出结论。

仍然以上一小节的罪犯识别问题为例，采用归结方法证明命题 d（丁是凶手）的过程如下。将结论转换为否定：

$$\neg d \tag{3-1-10}$$

将知识库化为析取式：

$$a \vee b \vee c \tag{3-1-11}$$

$$\neg c \vee a \tag{3-1-12}$$

$$\neg b \vee d \tag{3-1-13}$$

$$\neg a$$

式(3-1-4)和式(3-1-11)归结得到：$b \vee c$ (3-1-14)

式(3-1-4)和式(3-1-12)归结得到：$\neg c$ (3-1-15)

式(3-1-10)和式(3-1-13)归结得到：$\neg b$ (3-1-16)

式(3-1-14)和式(3-1-15)归结得到：b (3-1-17)

(3-1-16)式和(3-1-17)式归结得到空集{}。因此，得出矛盾，结论成立，丁是凶手。归结的证明树如图 3-1 所示。

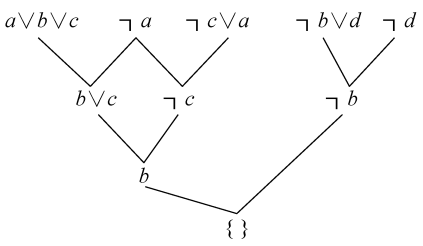

图 3-1　罪犯识别问题归结证明的证明树

在机器自动推理时，我们通常用特定的语句形式表示知识，以方便编写统一的程序处理，提高效率[2]。常用的形式有合取范式、限定子句和 Horn 子句。合取范式是形如下式的语句

$$(p_1 \vee q_1) \wedge (\neg p_2 \vee q_2) \wedge (p_3 \vee \neg q_3)$$

即整个语句是合取式，每一个子语句是析取式，非号位于原子语句之前。将一个语句化为合取范式的方法就是不断使用复合命题的运算法则。

例 3-2　将知识库 $(p \Rightarrow \neg q) \Leftrightarrow r$，$\neg q \Rightarrow r$ 化为合取范式。

解：首先明确，知识库中各语句是合取的关系，因此有

$$[(p \Rightarrow \neg q) \Leftrightarrow r] \wedge [\neg q \Rightarrow r]$$

采用双向蕴含消去：$[(p \Rightarrow \neg q) \Rightarrow r] \wedge [r \Rightarrow (p \Rightarrow \neg q)] \wedge [\neg q \Rightarrow r]$

采用蕴含消去：$[\neg(\neg p \vee \neg q) \vee r] \wedge [\neg r \vee (\neg p \vee \neg q)] \wedge [q \vee r]$

采用德摩根定律将非号分配到原子语句：$[(p \wedge q) \vee r] \wedge [\neg r \vee (\neg p \vee \neg q)] \wedge [q \vee r]$

最后采用分配律得到合取范式：$(p \vee r) \wedge (q \vee r) \wedge (\neg p \vee \neg q \vee \neg r) \wedge (q \vee r)$。限定子句是指恰好只含一个正文字的析取式。例如，$\neg c \vee a$ 和 $\neg b \vee d$ 是限定子句，而 $\neg p \vee \neg q \vee \neg r$ 和 $q \vee r$ 不是。Horn 子句是指至多包含一个正文字的析取式[3,4]。从这个定义上看，所有限定子句都是 Horn 子句。因此，$\neg c \vee a$、$\neg b \vee d$ 和 $\neg p \vee \neg q \vee \neg r$ 都是 Horn 子句，而 $q \vee r$ 不是。

现在给出归结算法的伪代码，如表 3-2 所示，其中 CNF 即合取范式。在每一步归结操作时，若存在重复的原子语句，保留一个即可。如 $a \vee b$ 和 $a \vee \neg b$ 归结，得到 $a \vee a$，只需保留一个 a 即可。最后我们指出，归结算法是完备的。算法的完备性是指，当解存在时，算法一定能够求得解。

表 3-2　命题逻辑的归结算法伪代码

```
PL-Resolution(KB, α)
Input: KB, the knowledge base, a sentence in propositional logic; α, the query, a sentence in propositional logic.
Output: true or false.
clause ← CNF representation of KB ∧ ¬α
new ← {}
Loop do
  For each pair of clauses C_i, C_j in clauses, do
    resolvents ← PL-Resolution(C_i, C_j)
    If resolvents contains the empty clause then return true
    new ← new ∪ resolvents
  If new ⊆ clauses then return false
  clauses ← clauses ∪ new
```

3.2　一阶逻辑

命题逻辑研究的基本元素是命题，具有一定的局限性。首先，命题是有真假意义的事实陈述，但对陈述的结构和成分是不予考虑的。这无法表达事物之间的联系。例如，命题逻辑认为“张三是人”“凡人必死”“张三必死”是三个互相独立的命题。但从语义上讲，显然他们之间是有内部联系的。另外，命题逻辑在表达一般性规则时也不简洁，有时甚至是不可能的。考虑知识“所有的恒星都会发光”。在命题逻辑中，这仅仅是一条命题。若要将其推广，就需要添加适用于每一颗恒星的命题，如“太阳会发光”。这对于数以万计的恒星而言显然是不现实的。为克服命题逻辑的局限性，我们需要深入到命题内部，对命题的结构、成分以及命题间的共同特性加以分析，这就是一阶逻辑[5]。

3.2.1　一阶逻辑的基本概念

一阶逻辑又称一阶谓词逻辑或一阶谓词演算，是将命题拆解为对象和关系（或称主语和谓语）进一步考察。对象（即知识表示一章的实体）是指代具体事物的名词或名词短语，如房子、草坪、Beckham 等。关系是描述事物性质的动词或动词短语。一元关系又称为属性，表示事物自身的性质，如“是红色的”“是整洁的”。多元关系表示事物之间的联系，如“比……大”“是……的哥哥”。函数用来指代与某事物相关的概念，如“……的哥哥”“……的开始”“……最好的朋友”。可以认为，几乎所有的命题都是由对象、关系和函数组成。例如，“张三是人”，对象“张三”，关系“是人”；“所有的恒星都会发光”，对象“恒星”，关系“会发光”。

与命题逻辑一样，一阶逻辑也有语法和语义。与对象、关系、函数相对应，一阶逻辑的基本句法元素是表示对象的常量符号、表示关系的谓词符号和表示函数的函词。通常这些符号首字母大写，而具体名称的选择则不做统一规定，但应尽量简洁明了。例如用“Brother (John)”表示“约翰的兄弟”。其中 Brother 是函词，John 是常量符号，“……的兄弟”是函词 Brother 的解释。前面讲到，符号的解释可以任意给出，但我们仍然按照常识进行解释，这

能够避免不必要的麻烦。函数可以进行复合，即可以有 Brother(Brother(John))，表示“约翰兄弟的兄弟”。这里需要说明，函数表示的是经过限定的对象，其本质仍然是对象。因此，函数与常量符号通常具有同等的地位。为统一表述，有文献将常量视为是参数为零的零元函数，不过本书仍然将常量和函数区别对待。在常量和函数的基础上，我们进一步定义项。项是常量符号和函数的统称，是指代某对象的逻辑表达式，如“John”和“Brother(John)”。有了指代对象的项和指代关系的谓词，可以将它们放在一起形成原子语句。例如“IsBrother(John, Brother(John))”表示“约翰和约翰的兄弟是兄弟”(虽然有点拗口且毫无意义，但形式上允许这样表示)。原子语句定义为只包含一个谓词的语句。与命题逻辑一样，我们也可以用逻辑连接词连接原子语句构成复合语句，如“$\neg Married(Father(Bob), Linda)$”“$\neg Brother(Peter) \Rightarrow Sister(Peter)$”。

对于通用规则的描述，一阶逻辑借助量词来实现简洁表达。量词包括全称量词和存在量词两种。全称量词∀，读作“对所有的……”(符号∀即为单词 All 的首字母倒置)。比如“凡人必死”可表达为

$$\forall x \ IsHuman(x) \Rightarrow Die(x)$$

其中 x 称为变量，不含变量的项称为基项。该表达式意为“对所有的 x，若 x 是人，则 x 会死”。若将变量 x 代不同的常量，就得到不同的知识。如代 x 为“Zhangsan”就得到 $IsHuman(Zhangsan) \Rightarrow Die(Zhangsan)$。存在量词∃，读作“存在……”或“存在某个……”(符号∃即为单词 Exist 的首字母倒置)。表达式 $\exists xP$ 的含义是至少存在一个对象使得 P 成立。例如知识“Mary 已经与某人结婚”可表示为

$$\exists x Married(Marry, x)$$

存在量词的另外一种扩展是∃1 和∃!，表示“存在唯一的……”。可以采用多个量词嵌套来表示更加复杂的情况。比如 $\forall x \ \forall y Father(x, y) \Rightarrow Kinsfolk(x, y)$，表示“所有的父子都是亲属”。也可写为 $\forall x, y Father(x, y) \Rightarrow Kinsfolk(x, y)$。再比如 $\forall x \ \exists y Marry(x, y)$，表示“每个人都会跟另外的人结婚”。这里需要排除 x 和 y 指代同一对象的情况。我们常常用等词＝来表示。x 和 y 指代不同对象表示为 $x \neq y$ 或 $\neg(x = y)$。

全称量词和存在量词可以相互转化。断言“所有人都不喜欢樱桃”等价于断言“不存在任何人喜欢樱桃”。断言“存在某人不喜欢樱桃”等价于断言“不是所有人都喜欢樱桃”。因此，两种量词的转化为

$$\forall x \neg Likes(x, Cherry) \Leftrightarrow \neg \exists x Likes(x, Cherry),$$
$$\exists x \neg Likes(x, Cherry) \Leftrightarrow \neg \forall x Likes(x, Cherry)$$

一阶逻辑能够转化为命题逻辑，使用的方法是量词的实例化。假设有公理“所有的学生都喜欢上 AI 课程”

$$\forall x Student(x) \Rightarrow Likes(x, AI)$$

对于包含有限个对象的知识库，我们可以将 x 置换为每个对象的常量符号得到

$$Student(Alice) \Rightarrow Likes(Alice, AI),$$
$$Student(Bob) \Rightarrow Likes(Bob, AI), \cdots$$

可见全称量词实例化后，考察对象的全集中每个对象都对应一条语句。对于存在量词限定的变量，其意义是该变量可能取到知识库中的某一个常量，从而使得后继公式成立。我们可以引入一个暂时无法确定值的一般性常量来代替这个变量。而这个一般性常量可能等

于知识库中的任何一个现有常量。例如有公理

$$\exists x\ Student(x) \Rightarrow \neg Likes(x, AI)$$

使用未出现过的常量符号 S_1 来表示一般性常量。那么公式可去掉存在量词，实例化为

$$Student(S_1) \Rightarrow \neg Likes(S_1, AI)$$

我们使用

$$\frac{\forall v, \alpha}{SUBST(\{v/k\}, \alpha)}$$

表示将变量 v 置换为基项 k 并应用于表达式 α 中。那么上面的两个实例化可表示为置换 $\{x/Alice\}$，$\{x/Bob\}$，……$\{x/S_1\}$。将存在量词实例化为一个更一般的常量符号，此过程称为 Skolem 化。对于包含量词的公式，若给其中的每一个变量都指定一个常量，那么此时公式就不再表示一般知识，从而量词也就失去了意义，可以去掉。为每个变量指定一个常量得到的公式，称为原公式的一个解释。

消去量词后，得到的一阶逻辑表达式的每一个原子语句，可以看成是命题，从而一阶逻辑就转化成了命题逻辑。需要注意的是，如果知识库中包含有函词，那么置换可能是无限的。如 $Father(Tom)$、$Father(Father(Tom))$、$Father(Father(Father(Tom)))$……不过有定理保证如果语句被原始一阶知识库蕴含，则能够找到有限嵌套深度的命题证明。

3.2.2 合一算法

在进入一阶逻辑推理之前，首先介绍合一算法[6,7]。合一是指寻找使得不同逻辑表达式变得相同的置换，即 $UNIFY(p, q) = \theta$ 使得 $SUBST(\theta, p) = SUBST(\theta, q)$。比如查询 $Hates(Bob, x)$（*Bob* 讨厌谁?），可以通过寻找知识库所有匹配该查询的置换得到结果：

$UNIFY(Hates(Bob, x), Hates(Bob, Charles)) = \{x/Charles\}$

$UNIFY(Hates(Bob, x), Hates(y, Alice)) = \{x/Alice, y/Bob\}$

$UNIFY(Hates(Bob, x), Hates(y, Opponent(y))) = \{x/Opponent(Bob), y/Bob\}$

$UNIFY(Hates(Bob, x), Hates(x, Debby)) = fail$

最后一个合一失败是因为变量 x 不能同时取 *Bob* 和 $Debby$（这实际上加入了隐藏条件“任何人不能讨厌自己”）。解决这个问题的办法类似于第二个式子，对合一的两个表达式采用不同的变量符号，这称为变量的标准化分离。因此，有

$$UNIFY(Hates(Bob, x), Hates(y, Debby)) = \{x/Debby, y/Bob\}$$

一般而言，合一是不唯一的，存在最一般的合一置换，称为 MGU(Most General Unification)。表 3-3 给出了合一算法的伪代码。

表 3-3 合一算法伪代码

```
UNIFY(p, q)
Input: p, a constant, variable, list or expression
       q, a constant, variable, list or expression
Output: a substitution
If p and q are both constants or empty lists,
  If p = q, then return null;
  Else return failure;
Else If p is a variable,
```

续表

```
   If p appears in q, then return failure;
   Else return {q/p};
Else If q is a variable,
   If q appears in p, then return failure;
   Else return {p/q};
Else If either p or q is empty, then return failure;
Else
   hp: = the first element of p;
   hq: = the first element of q;
   SUBST_1: = UNIFY(hp,hq);
   If SUBST_1 = failure, then return failure;
   Else
     tp: = Apply(SUBST_1, the rest of p);
     tq: = Apply(SUBST_1, the rest of q);
     SUBST_2: = UNIFY(tp,tq);
     If SUBST_2 = failure, then return failure;
     Else return {SUBST_1, SUBST_2}.
```

3.2.3　前向链接和反向链接

顾名思义，前向链接是不断检查知识库中是否有满足条件的语句，并将其结论加入到知识库中。该过程迭代进行直到无法再添加新的知识为止。思考下面的例子。

例 3-3（学生评价问题）　所有顺利毕业的且通过 AI 考试的学生是优秀的。任何肯学习或幸运的学生能够通过 AI 考试。小张不爱学习但很幸运。任何学生只要幸运就能毕业。求证：小张是优秀的。

执行前向链接算法需要将知识库中的语句都化成一阶确定子句的形式，以方便计算机操作。一阶确定子句是一个蕴含语句或者原子语句。其中蕴含语句的条件是正文字的合取式，结论是一个单独的正文字。原子语句就是一个单独的正文字。例如，将例题中的知识写成一阶确定子句为

$$Graduate(x) \wedge PassAI(x) \Rightarrow Excellent(x) \tag{3-2-1}$$

$$Study(y) \Rightarrow PassAI(y) \tag{3-2-2}$$

$$Lucky(z) \Rightarrow PassAI(z) \tag{3-2-3}$$

$$Lucky(Zhang) \tag{3-2-4}$$

$$Lucky(w) \Rightarrow Graduate(w) \tag{3-2-5}$$

注意，知识库中的全称量词没有明确写出，这不影响推理。对于知识“小张不学习”，由于是否定形式 $\neg Study(Zhang)$，因此一阶确定子句中不包含。这实际上是采用了封闭世界假设，即凡是知识库中不包含的语句均认为是假。

前向链接的迭代过程如下。首先，第一轮迭代遍历知识库，规则(3-2-3)得到满足，相应的置换为$\{z/Zhang\}$，添加语句 $PassAI(Zhang)$；规则(3-2-5)得到满足，相应的置换为$\{w/Zhang\}$，添加语句 $Graduate(Zhang)$。第二轮迭代，规则(3-2-1)得到满足，置换为

$\{x/Zhang\}$，得到结论

$$Graduate(Zhang) \wedge PassAI(Zhang) \Rightarrow Excellent(Zhang)$$

对于确定子句的知识库，前向链接算法总是能得到正确的查询结论。这被称为前向链接算法具有完备性。

与前向链接相反，反向链接首先考虑结论，然后查询知识库是否满足前提中的每一个合取子句。这又可以分为两种情况，一种是知识库中直接包含合取子句，另一种是在某置换下包含合取子句。例如上面的学生评价例子，首先考虑结论 $Excellent(Zhang)$，查询知识库得到规则(3-2-1)的前提 $Graduate(Zhang) \wedge PassAI(Zhang)$，置换为$\{x/Zhang\}$。然后将 $Graduate(Zhang)$和 $PassAI(Zhang)$分别作为结论，查询得到 $Lucky(Zhang)$，置换为$\{w/Zhang\}$，以及 $Study(Zhang)$，置换为$\{y/Zhang\}$或 $Lucky(Zhang)$，置换为$\{z/Zhang\}$。第三轮将 $Lucky(Zhang)$、$Study(Zhang)$作为结论，查询得到规则(3-2-4)满足要求。注意第二轮计算时，$Study(Zhang)$和 $Lucky(Zhang)$是析取关系，其中之一成立即可。

3.2.4 归结证明

鲁滨逊归结原理在一阶逻辑下仍然适用。与命题逻辑类似，一阶逻辑的归结证明也需要先将知识库化为合取范式。例如，将知识“帮助所有失学儿童的人也会得到其他人的帮助”化成合取范式，步骤如下：

首先写出表达式

$$\forall x[\forall y\ Dropout(y) \Rightarrow Help(x,y)] \Rightarrow [\exists y\ Help(y,x)]$$

消去蕴含词

$$\forall x[\neg \forall y \neg Dropou(y) \vee Help(x,y)] \vee [\exists y\ Help(y,x)]$$

¬号内移

$$\forall x[\exists y \neg (\neg Dropout(y) \vee Help(x,y)] \vee [\exists y\ Help(y,x)]$$

$$\forall x[\exists y\ Dropout(y) \wedge \neg Help(x,y)] \vee [\exists y\ Help(y,x)]$$

变量标准化。对于符号相同的不同变量，改变其中一个变量符号，加以区分

$$\forall x[\exists y\ Dropout(y) \wedge \neg Help(x,y)] \vee [\exists z\ Help(z,x)]$$

Skolem 化。按照前面命题逻辑介绍的方法，直接使用另外的一个变量名代替存在量词变量，得到

$$\forall x[Dropout(A) \wedge \neg Help(x,A)] \vee [Help(B,x)]$$

显然，该语句表示“对于所有帮助失学儿童 A 的人，B 都会帮助他”。它与我们想要表达的意思——“帮助所有失学儿童的人都会得到另外某个人的帮助”——并不一致。因此，引入两个 Skolem 函数 H 和G，其结果依赖于量词 x 和z，得到

$$\forall x[Dropout(H(x)) \wedge \neg Help(x,H(x))] \vee [Help(G(z),x)]$$

删除全称量词，得到

$$[Dropout(H(x)) \wedge \neg Help(x,H(x))] \vee [Help(G(z),x)]$$

最后采用分配律得到合取范式

$$[Dropout(H(x)) \vee Help(G(z),x)] \wedge [\neg Help(x,H(x)) \vee Help(G(z),x)]$$

对于已完成变量标准化且没有共享变量的子句，我们可以归结互补文字。归结方法仍

然是

$$\frac{l_1 \vee \cdots \vee l_k, m_1 \vee \cdots \vee m_n}{SUBST(\theta, l_1 \vee \cdots \vee l_{i-1} \vee l_{i+1} \vee \cdots \vee l_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n)}$$

置换 θ 使得 l_i 和 m_j 成为互补文字。请再考虑前面的学生评价例子。知识库化成合取范式是

$$[\neg Graduate(x) \vee \neg PassAI(x) \vee Excellent(x)] \wedge [\neg Study(y) \vee PassAI(y)] \wedge [\neg Lucky(z) \vee PassAI(z)] \wedge Lucky(Zhang) \wedge [\neg Lucky(w) \vee Graduate(w)]$$

将上述合取范式中的每个子句单独书写并编号：

$$\neg Graduate(x) \vee \neg PassAI(x) \vee Excellent(x) \quad (3\text{-}2\text{-}6)$$

$$\neg Study(y) \vee PassAI(y) \quad (3\text{-}2\text{-}7)$$

$$\neg Lucky(z) \vee PassAI(z) \quad (3\text{-}2\text{-}8)$$

$$Lucky(Zhang) \quad (3\text{-}2\text{-}9)$$

$$\neg Lucky(w) \vee Graduate(w) \quad (3\text{-}2\text{-}10)$$

待证结论的否定为

$$\neg Excellent(Zhang) \quad (3\text{-}2\text{-}11)$$

归结证明过程如下：

(3-2-6) 和(3-2-11) 作置换$\{x/Zhang\}$，归结得到 $\neg Graduate(Zhang) \vee \neg PassAI(Zhang)$ (3-2-12)

(3-2-8) 和(3-2-12) 作置换$\{z/Zhang\}$，归结得到 $\neg Graduate(Zhang) \vee \neg Lucky(Zhang)$ (3-2-13)

(3-2-9) 和(3-2-13) 归结得到 $\neg Graduate(Zhang)$ (3-2-14)

(3-2-10) 和(3-2-14) 作置换$\{w/Zhang\}$，归结得到 $\neg Lucky(Zhang)$ (3-2-15)

(3-2-9)和(3-2-15)归结得到空集{}。证毕。

上述过程的证明树如图 3-2 所示，注意第(3-2-7)条知识并未使用。

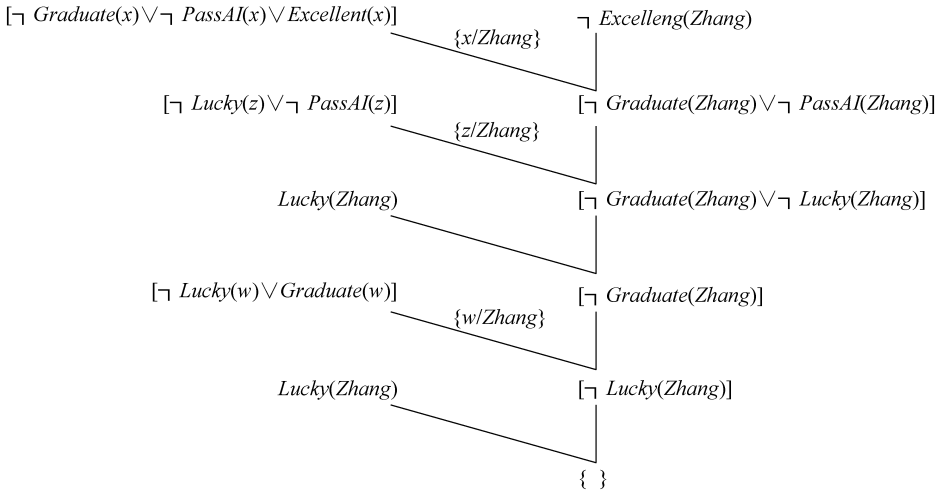

图 3-2　学生评价问题的归结证明树

3.3 Herbrand 定理

在上一节中，我们已经看到给定知识库和待查询结论，基于封闭世界假设，归结原理能在有限步内判定该结论是否是永真或永假的。那么对于所有的查询结论，归结原理是否都能在有限步内判定其永真或永假性呢？1930 年法国数学家 Herbrand 回答了这一问题。Herbrand 定理是逻辑推理中的一个核心定理。今天，人工智能中定理证明的最高奖项就是以 Herbrand 的名字命名的。本节将主要讨论 Herbrand 定理的基本内容。所述内容涉及形式化逻辑的推导，理论性较强，目的是起抛砖引玉之作用。

给定一个知识库 K，如果要证明查询结论 G 是永真的，根据前面逻辑的复合运算，只需证明公式 $D=K \wedge \neg G$ 是不可满足的，从而只需证明 D 在 K 中的所有解释是假的（请回忆解释的定义）。这里 K 称为论域。对于一般的一阶逻辑知识库 K 而言，证明是困难的。原因在于 K 上的解释往往是无限的。一个思路是如果能够将 K 上的解释映射到另一个较为简单的论域上，并且该论域上的解释可数，那么证明就会变得简单。Herbrand 域就是满足要求的一个简单域，简称为 H 域。在讲述 H 域的构造之前，先来看看公式 Skolem 化的一般形式。

考虑含有 n 个变量的一阶逻辑公式

$$(Q_1x_1)(Q_2x_2)\cdots(Q_nx_n)D \tag{3-3-1}$$

其中，x_i 是变量，Q_i 是量词 $\forall$ 或 $\exists$，D 是不含量词的合取范式。上式称为前束范式，D 称为母式。注意，任意一阶逻辑公式都可以化为前束范式。方法是按照下面的规则，将每个变量约束前移（左移），再将非量词部分化为合取范式：

（1）$\neg \forall xD$ 前移为 $\exists x \neg D$；

（2）$\neg \exists xD$ 前移为 $\forall x \neg D$；

（3）$QxB(x) \vee C$ 化为 $Qx[B(x) \vee C(x)]$，若 C 中含有约束变量 x；

（4）$B \vee QxC(x)$ 化为 $Qx[B(x) \vee C(x)]$，若 B 中含有约束变量 x。

前束范式的 Skolem 化需要从左至右依次消去存在量词。具体讲，对 x_i

（1）若 x_i 的前方（左方）不存在全称量词变量，则将 D 中的 x_i 用一般常量 a 代替。a 是 D 中未出现过的常量符号。

（2）若 x_i 的前方存在全称量词变量 $x_{k_1},\cdots,x_{k_m}$，$1 \leqslant k_1 < \cdots < k_m < i$，则引入函数 $f(x_{k_1},\cdots,x_{k_m})$ 代替 D 中的变量 x_i。f 是 D 中未出现过的函数符号，具体形式没有要求，它表示当前面的全称量词变量取定后，映射到原公式论域上的一个 x_i 取值（即存在性）。

（3）删除前缀中的约束 (Q_ix_i)。

例如，将公式 $(\exists x)(\forall y)(\forall z)(\exists u)(\forall v)(\exists w)P(x) \wedge [Q(y) \vee R(z)] \wedge [M(u) \vee N(v) \vee L(w)]$ 化为 Skolem 范式：

消去变量 x：$(\forall y)(\forall z)(\exists u)(\forall v)(\exists w)P(a) \wedge [Q(y) \vee R(z)] \wedge [M(u) \vee N(v) \vee L(w)]$

消去变量 u：$(\forall y)(\forall z)(\forall v)(\exists w)P(a) \wedge [Q(y) \vee R(z)] \wedge [M(f(y,z)) \vee N(v) \vee L(w)]$

消去变量 w：$(\forall y)(\forall z)(\forall v)P(a)\wedge[Q(y)\vee R(z)]\wedge[M(f(y,z))\vee N(v)\vee L(g(y,z,v))]$

$f(y,z)$和 $g(y,z,v)$分别称为 u 和 w 的 Skolem 函数。

定理 3-1　设 $M=(Q_1x_1)(Q_2x_2)\cdots(Q_nx_n)D[x_1,\cdots,x_n]$是前束范式，$Q_r$ 是从左到右的第一个存在量词，令

$M'=(Q_1x_1)\cdots(Q_{r-1}x_{r-1})(Q_{r+1}x_{r+1})\cdots(Q_nx_n)D[x_1,\cdots,x_{r-1},f(x_1,\cdots,x_{r-1}),x_{r+1},\cdots,x_n]$

是消去 Q_r 的结果，f 是 Skolem 函数。那么 M 和 M'在不可满足性上等价，即 M 不可满足⇔M'不可满足。

证明：⇒：首先明确，M 和 M'的论域是相同的，记为 K。当 M 不可满足，反设 M'可满足。那么存在一个解释使得 M'的值为真。因此对任意的 $x_1,\cdots,x_{r-1}$，存在 $f(x_1,\cdots,x_{r-1})$使得

$$(Q_{r+1}x_{r+1})\cdots(Q_nx_n)D[x_1,\cdots,x_{r-1},f(x_1,\cdots,x_{r-1}),x_{r+1},\cdots,x_n]$$

为真。而 $f(x_1,\cdots,x_{r-1})$是 K 上的一个元素，因此对任意的 $x_1,\cdots,x_{r-1}$，

$$(\exists x_r)(Q_{r+1}x_{r+1})\cdots(Q_nx_n)D[x_1,\cdots,x_{r-1},x_r,x_{r+1},\cdots,x_n]$$

为真，即$(\forall x_1)\cdots(\forall x_{r-1})(\exists x_r)(Q_{r+1}x_{r+1})\cdots(Q_nx_n)D[x_1,\cdots,x_n]$为真，即 M 是可满足的，矛盾。

⇐：当 M'不可满足，反设 M 可满足，则存在一个解释使得 M 为真，记为 I。因此对于任意的 $x_1,\cdots,x_{r-1}$，存在 $x_r\in K$ 使得

$$(Q_{r+1}x_{r+1})\cdots(Q_nx_n)D[x_1,\cdots,x_{r-1},x_r,x_{r+1},\cdots,x_n]$$

为真。扩充解释使其包含对 Skolem 函数的指定，即令 $I'=I\cup\{x_r=f(x_1,\cdots,x_{r-1})\}$。那么对任意的 $x_1,\cdots,x_{r-1}$，有

$$(Q_{r+1}x_{r+1})\cdots(Q_nx_n)D[x_1,\cdots,x_{r-1},f(x_1,\cdots,x_{r-1}),x_{r+1},\cdots,x_n]$$

因此 M'可满足，矛盾。

如果公式中包含多个存在量词，可采用数学归纳法按上述定理类似地证明。

定理 3-1 保证了要证明某公式不可满足，只需证明它的 Skolem 范式不可满足。从前面的例子中可以看出，Skolem 范式的基本形式是合取符号连接了多个析取式。我们将每一个析取式称为子句，所有的析取式构成子句集。比如例子中的子句集可表示为

$$D=\{P(a),Q(y)\vee R(z),M(f(y,z))\vee N(v)\vee L(g(y,z,v))\}$$

因此，证明 Skolem 范式不可满足，只需证明子句集不相容（即至少有一个子句为假）。下面讨论如何建立 H 域，将子句集的无限解释转换为可数的，进而最终转换成有限的解释来证明公式。H 域的构造方法如下：令 $H_0=\{D$ 上的常量集合$\}$。若 D 中没有常量，则引入一个一般常量 c。$H_{i+1}=H_i\cup\{f(c_1^i,\cdots,c_n^i)\}$，其中，$f$ 是 D 中的所有函数符号，$c_j^i\in H_i$ 是 H_i 中的元素。按此递推，H_∞称为公式 D 的 H 域。例如，求子句集 $D=\{P(a),Q(x)\vee R(f(x))\}$的 H 域，其中 a 是常量，f 是函数。按照构造方法

$$H_0=\{a\},$$
$$H_1=H_0\cup\{f(a)\}=\{a,f(a)\},$$
$$H_2=H_1\cup\{f(a),f(f(a))\}=\{a,f(a),f(f(a))\},\cdots$$

所以 H 域为 $H_\infty=\{a,f(a),f(f(a)),f(f(f(a))),\cdots\}$。对于子句集 D 中的每个谓词，为其变量指定一个 H 域上的元素（常量或函数），得到的谓词集合称为 D 的原子集。

上面例子中 D 的原子集为

$\{P(a),Q(a),R(f(a)),Q(f(f(a))),R(f(f(f(a)))),\cdots\}$。注意每个谓词的变量要取遍 H 域。相应地，由原子构成的子句集 D 称为基例，如$\{P(a),Q(a)\vee R(f(a))\}$，$\{P(a),Q(f(a))\vee R(f(f(a)))\}$，…。进一步，对原子集中的每个原子指定一个真值，所对应的 D 的真值称为 D 在 H 域上的一个解释(再次提醒 D 是各子句的合取)。根据 H 域的构造特点，H_∞是无限可数的，因此 D 在 H 上的解释也是无限可数的。另一方面，H 域实质上是列出了论域K 所有可能的变量取值。因此每一个在 K 上的解释都对应着在H 域上的解释。这个解释可用语义树(Semantic Tree)来表示。

语义树是原子集中的每个原子与它的否定排列成树形结构。例如对子句集 $D=\{P(a),Q(x)\vee R(f(x))\}$，它的原子集为

$$\{P(a),Q(a),R(f(a)),Q(f(f(a))),R(f(f(f(a)))),\cdots\}$$

那么语义树如图 3-3 所示。语义树有以下几个特点：

(1) 每一层代表一个原子的取值。原子之间的顺序可以任意排定。

(2) 由于 H 域的原子是无限的，因此语义树的深度也是无限的。

(3) 从根结点出发到叶结点的每一条路径(无穷深度的语义树叶结点视为在无穷远处)对应一个 H 域上的解释。从根结点出发到某个中间结点的路径对应了 H 域上的一个部分解释。

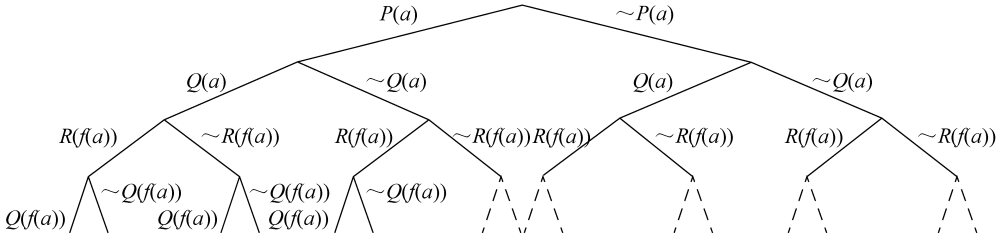

图 3-3　语义树

若按语义树的构造方法，将所有的原子 P 及其互补原子$\sim P$ 都布置于树上时，此时的语义树称为完备语义树。完备语义树对应着所有的 H 解释。若语义树上根结点到某个中间结点 N 的部分解释为假，而到 N 的父结点的部分解释都为真，那么称 N 是一个失败结点或假结点。失败结点的存在为我们提供了将无限路径转化为有限路径的可能。这是因为失败结点对应的解释已经使得子句集为假(至少有一个子句为假)，继续扩展语义树路径并不改变此结果(为假的子句仍然存在)。若每一个叶结点都是失败结点，此时的语义树称为封闭的语义树。基于此，Herbrand 提出了以下定理[8]。

定理 3-2(Herbrand 定理 1)　子句集 D 是不可满足的，当且仅当对应于 D 的任一棵完备语义树，都存在一棵有限的封闭语义树。

定理 3-3(Herbrand 定理 2)　子句集 D 是不可满足的，当且仅当存在一个有限不可满足的 D 的基础实例集合 D'。

容易验证两个版本的 Herbrand 定理是等价的。定理的证明按照语义树的思路也容易完成，此处不再赘述[9,10]。Herbrand 定理的本质，是将一阶逻辑的证明转化成了有限的命题逻辑的证明，从而使证明变得简单。需要说明的是，Herbrand 定理仅仅是指明了不可满

足性的证明能够转化为寻找封闭语义树。但是如何寻找这样的封闭语义树，定理并没有给出。按照经典的方法，通常需要：①构造 H 域；②构造原子集；③构造完备的语义树；④求失败结点；⑤形成封闭的语义树。不幸的是，该过程是指数复杂度的。Davis 等人引入了四条启发式规则，使得计算效率大大提高[11,12]：

(1) 重言式规则。形如 $P \vee \neg P$ 的表达式称为重言式。重言式规则指出，如果子句集中包含重言式，则应将其删除，因为重言式不会为不可满足性提供任何信息。

(2) 单文字规则。若子句集 D 中包含一个单文字子句 L，则消去 D 中所有含 L 的子句，得到子句集 D'。若 D'为空，则 D 是可满足的。否则，继续从 D'中消去～L 得到 D''。D''不可满足当且仅当 D 不可满足。

(3) 纯文字规则。若 D 中的子句包含文字 L，但不包含～L，则称 L 是 D 的纯文字。若 L 是 D 的纯文字，则消去所有含 L 的子句，得到 D'。若 D'为空集，则 D 是可满足的，否则，D'不可满足当且仅当 D 不可满足。

(4) 分裂规则。设 $D=(L \vee A_1) \wedge \cdots \wedge (L \vee A_m) \wedge (\sim L \vee B_1) \wedge \cdots \wedge (\sim L \vee B_n) \wedge C$，其中 A_i、B_i、C 不含 L 和～L。令 $D'=A_1 \wedge \cdots \wedge A_m \wedge C$，$D''=B_1 \wedge \cdots \wedge B_n \wedge C$，则有 D 不可满足当且仅当 $D' \vee D''$不可满足。

上述四条启发式规则为后来归结原理的提出奠定了基础。

3.4　逻辑系统编程语言

在了解逻辑推理的基本原理之后，本节讲述如何在程序中完成这个过程。从理论上讲，图灵计算的完备性保证了任何编程语言都是等价的。由此出发，可以采用 C、Java 或者其他任何编程语言来实现逻辑推理的程序。然而，这并非它们的长处所在。研究逻辑系统的 AI 专家早已开发出了适合的编程语言。灵活使用它们能够简洁而迅速地完成逻辑系统的开发与调试。另一方面，逻辑编程也是构建编译器的基础之一，对它们的学习也能够了解编译器的设计思想。本书并非程序设计书籍，因此下面只简单介绍两种逻辑编程语言的基本特点和示例。

3.4.1　Prolog

Prolog 的全称是 Programming in Logic。从命名就能看出，该语言是专为逻辑系统编程设计的[13,14]。事实上，Prolog 可能是该领域使用最广泛的语言之一了，它最早出现于 1972 年，由 Aix-Marseille 大学开发。它是一种基于确定子句集的陈述性程序设计语言。Prolog 用大写字母代表变量、小写字母代表常量。Prolog 的语句有三种，分别是事实、规则和问题。典型的事实陈述是以谓词给出的，如 $PassAI(Zhang).$、$Married(Bob,Kate).$表示“Zhang 通过 AI 考试”“Bob 和 Kate 结婚”。注意每条语句都是以点结尾的。特别地，事实也可以只有谓词而没有参量，如 $Hot.$。典型的规则表示用符号：－给出，这个符号意为“if”，也可以直接写成 if。例如

$$Excellent(X):-Graduate(X),PassAI(X).$$

即表示规则 $Graduate(x) \wedge PassAI(x) \Rightarrow Excellent(x)$。：－符号左侧是规则的“Then”部分(也

称规则头),右侧是"If"部分(也称规则体),逗号代表合取关系。规则中的谓词也可以只有谓词而没有参量。问题是用户的查询目标语句,以符号?-给出。例如?-*Excellent*(*Zhang*).。将上一节中学生评价问题改写成 Prolog 程序如下:

```
Lucky(Zhang).
PassAI(Y):-Study(Y).
PassAI(Z):-Lucky(Z).
Graduate(W):-Lucky(W).
    Excellent(X):-Graduate(X),PassAI(X).
        ?- Excellent(Zhang).
```

这个程序有一条事实、四条规则和一个问题。其中事实、规则和问题都分行书写。规则和事实可连续排列在一起,其顺序可随意安排,但同一谓词名的事实或规则必须集中排列在一起;问题不能与规则及事实排在一起,它作为程序的目标要么单独列出,要么在程序运行时临时给出。可以看到,Prolog 程序其实就是确定性子句的排列,具备很强的表示能力。例如,当事实和规则是某学科的公理时,那么问题就是待证命题;当事实和规则是某些数据和关系时,那么问题就是数据查询语句;当事实和规则是特定领域知识时,那么问题就是利用这些知识求解的问题;当事实和规则是某初始状态和状态变化规律时,那么问题就是目标状态。最后一点表明,同过程性语言相比,Prolog 程序的问题就相当于主程序,规则就相当于子程序,而事实就相当于数据。

对于上面的学生评价 Prolog 程序,机器将执行反向链接过程来证明结论。首先,系统扫描知识库尝试寻找与问题?-*Excellent*(*Zhang*).具有相同谓词的子句,显然只有规则

$$Excellent(X):-Graduate(X),PassAI(X).$$

满足。系统会尝试向变量分配常量使得两个子句头部的谓词匹配。这其实是求置换的过程,被称为变量绑定(Bindings)。相应地,将变量的值解除称为解绑。在 Prolog 中,谓词的匹配是指两个谓词的谓词名、参量个数、参量类型都相同,并且对应参量满足:

(1) 如果两个参量都是常量,则必须相同;

(2) 如果两个都是被绑定的变量,则所绑定的常量必须相同;

(3) 如果一个是常量,一个是被绑定的变量,则所绑定的值与常量必须相同;

(4) 至少有一个参量是未绑定变量。

在这里的例子中,当变量 X 被绑定为常量 *Zhang* 时,即作置换$\{X/Zhang\}$时,问题与规则头部匹配,系统将目标转化为

$$Graduate(Zhang),PassAI(Zhang)$$

这相当于反向链接完成了第一轮,继续查询条件。下一步,系统将依次求解两个子目标,求解过程与上面的一样。这个递归过程不断地寻找与当前目标匹配的谓词及置换,并将目标转换为子目标。需要明确的是,系统在求解第一个子目标 *Graduate*(*Zhang*)时,如果生成新的子目标,将会优先继续求解它。这被称为深度优先搜索。搜索最终的结束条件有两种:一种是当前目标与知识库中的部分子句完全匹配,表明证明成功,查询结论正确,记录每一步递归的置换。另一种是扫描完整个知识库时没有发现匹配的谓词,表明证明失败,查询结论错误,此时系统会将本轮递归绑定的变量解绑。程序都会退回到上一轮递归,检查其他未求解的子目标。这个过程称为回溯。一般而言,Prolog 系统只返回第一个搜索到的

证明。用户也可以控制程序继续搜索以给出全部证明(如果存在)。如果不需要寻找所有证明过程,则可以用操作符 cut 终止回溯。但是,使用 cut 要尤其小心,因为它可能会导致漏掉解!

Prolog 程序的执行将交由系统自动完成,用户不需要过多地设计程序具体实现细节,而只需要给出已知事实、规则和待查询的结论。这使得用户能够更多地关注任务而非实现。但是,在非常特殊的情况下,Prolog 可能会产生无限循环而无法证明结论。另外,深度优先搜索的复杂度是指数级的,也会导致推理过程中出现大量的中间步骤而使计算变得低效。目前,Prolog 语言有很多改进版本。其中使用得比较广泛的是 SWI-Prolog。它基于 Prolog 内核做了计算性能上的优化。另外还有一些推理引擎能够支持逻辑程序,如基于 Python 语言的 pyDatalog。

3.4.2　LISP

LISP 的英文全称是 List Processing,最初是由人工智能之父 J. McCarthy 于 1956 年设计的。时至今日,它仍然是一种广泛使用的 AI 编程语言。目前使用较多的是 Common Lisp。事实上,当代程序设计语言很多都采纳了 LISP 的思想,比如函数式编程模型、垃圾自动回收机制等。LISP 以函数递归为主要实现形式,具有与图灵机相同的计算能力。除了能处理一般的数值计算,LISP 还有一套符号处理函数,具备符号集上的递归能力,原则上能够处理人工智能中的任何符号计算问题。LISP 的数据和程序是一致的,程序可以作为数据来处理,数据也可以作为程序来执行。从这个意义上讲,LISP 能够自己编写程序! LISP 有着丰富的内涵,限于篇幅,我们这里只能对其符号计算作一简单讨论。

与 Prolog 稍有不同,LISP 采用以下形式陈述事实

$$(parent\ donald\ nacy)$$

括号表示 LISP 的基本数据结构,称为表(List)。此表中包含三个元素,第一个元素是谓词(或函数),后面两个是参量。这条事实意为“Donald 是 Nacy 的家长”。规则的表示形式为

$$(<\!\!-\ (child\ ?x\ ?y)(parent\ ?y\ ?x))$$

这条规则使用问号作为前缀来表示变量,意为:如果 y 是 x 的家长,那么 x 是 y 的孩子。一般地,规则的书写形式为

$$(<\!\!-\ head\ body)$$

head 是规则头,body 是规则体。规则可以是一个复杂的表达式,例如规则“如果 x 是 y 的家长,并且 x 是男性,那么 x 是 y 的父亲”可以写成

$$(<\!\!-\ (father\ ?x\ ?y)(and(parent\ ?x\ ?y)(male\ ?x)))$$

符号“<-”是一个宏,后面两个是参量。所不同的是,参量是由另外的函数充当。LISP 在计算中首先对参量函数求值,将返回值作为参量传入。事实陈述也同样可以写成上述形式从而使得知识库表示方式统一

$$(<\!\!-\ (parent\ donald\ nacy))$$

对于求置换,我们定义一个函数 match 用来返回变量的绑定关系

```
(defun match (x y &optional binds)
  (cond
   ((eql x y) (values binds t))
```

```
    ((assoc x binds) (match (binding x binds) y binds))
    ((assoc y binds) (match x (binding y binds) binds))
    ((var? x) (values (cons (cons x y) binds) t))
    ((var? y) (values (cons (cons y x) binds) t))
    (t
     (when (and (consp x) (consp y))
       (multiple-value-bind (b2 yes)
         (match (car x) (car y) binds)
         (and yes (match (cdr x) (cdr y) b2)))))))
(defun var? (x)
  (and (symbolp x)
       (eql (char (symbol-name x) 0) #\?)))
(defun binding (x binds)
  (let ((b (assoc x binds)))
    (if b
        (or (binding (cdr b) binds)
            (cdr b)))))
```

这段程序看上去有些复杂。然而，相比于用其他语言实现同样的功能，它已经足够简单了(这从另一个侧面印证了 LISP 的强大)。这里定义了三个函数。函数 var? 判断传入参数是否为已经绑定的变量，函数 binding 尝试绑定变量，函数 match 使用递归搜索所有绑定。用前面的 match 函数查询

$$(parent\ ?y\ ?x)$$

得到

```
> (match '(parent donald nacy) '(parent ?y ?x))
((?Y . DONALD) (?X . NACY))
T
```

当 match 函数逐个元素地比较它的参数的时候，它把 binds 参数中的值分配给变量。如果成功匹配，match 函数返回生成的绑定；否则，返回 nil。例如查询

```
> (match '(parent donald nacy) '(parent bob ?x))
NIL
```

定义完匹配函数后，接下来我们构造一个用于反向链接的证明函数

```
(defun prove (expr &optional binds)
  (case (car expr)
    (and (prove-and (reverse (cdr expr)) binds))
    (or (prove-or (cdr expr) binds))
    (not (prove-not (cadr expr) binds))
    (t (prove-simple (car expr) (cdr expr) binds))))
(defun prove-simple (pred args binds)
  (mapcan #'(lambda (r)
              (multiple-value-bind (b2 yes)
                                   (match args (car r) binds)
                (when yes
                  (if (cdr r)
                      (prove (cdr r) b2)
```

```
                    (list b2)))))
          (mapcar #'change-vars
                  (gethash pred *rules*))))
(defun change-vars (r)
  (sublis (mapcar #'(lambda (v) (cons v (gensym "?")))
                  (vars-in r))
          r))
(defun vars-in (expr)
  (if (atom expr)
      (if (var? expr) (list expr))
    (union (vars-in (car expr))
           (vars-in (cdr expr)))))
```

其中 prove 函数是推论进行的枢纽。它接受一个表达式和一个可选的绑定列表作为参数。如果表达式不包含逻辑操作,它调用 prove-simple 函数。可以看到,prove-simple 函数内部又会调用 prov 函数,我们反向链接由此产生。这个函数查看所有拥有正确判断式的规则,并尝试对每一个规则的 head 部分和它想要证明的事实做匹配。对于每一个匹配的 head,使用匹配所产生的新的绑定在 body 上调用 prove。对 prove 的调用所产生的绑定列表被 mapcan 收集并返回。*rules* 代表我们的规则库,目前有一条规则:

$$(<\!\!-(parent\ donald\ nacy))$$

查询两条问题

```
> (prove-simple 'parent '(donald nancy) nil)
(NIL)
> (prove-simple 'child '(?x ?y) nil)
(((#:?6 . NANCY) (#:?5 . DONALD) (?Y . #:?5) (?X . #:?6)))
```

第一个返回 NIL 表示证明失败。这是因为尽管查询式与规则的参数值相等,但内存中它们并不是一个变量(即变量地址不同)。只有当是同一个变量时,LISP 才返回真。第二个查询式返回所有的绑定,即?x 和?y 被间接绑定到 nancy 和 donald。

对比 LISP 和 Prolog 两种逻辑系统编程语言,可以看到前者更加基础。事实上,LISP 能够实现 Prolog 解释器。前面的例子展示出,LISP 为开发人员提供了更大的灵活性,用户可以自主控制证明的过程。当然,享受这种灵活性也需要程序员具备更高的技术基础。通俗地讲,LISP 是逻辑系统编程的“汇编语言”,而 Prolog 则是该领域的“高级语言”。

3.5　专家系统

专家系统是早期人工智能的重要分支,基本特点是模仿人类领域专家来分析求解复杂问题。传统上,专家系统通常采用一阶逻辑知识库加上推理机实现。人类专家的经验和知识以一阶逻辑 If…Then…规则的形式写入知识库,称为产生式规则(Production-Rule)。以产生式规则作为知识存储形式的系统称为产生式规则系统。推理机又称为规则解释器,实现了逻辑推理算法,用来模拟人类专家的分析思考过程。当然,知识库并不仅仅局限于产生式规则一种形式,上一章的知识图谱以及本书后面将要讲到的神经网络等都可以用来作为知识存储与管理的方式。相应地,推理机也可以采用图搜索等其他方法实现。然而,专家系

统最早是从一阶逻辑发展而来，因此本节仍然重点介绍基于一阶逻辑的专家系统[15]。

逻辑专家系统首先需要构建产生式规则库，将人类专家的经验知识予以机器化表示。该过程通常涉及以下几个步骤：

(1) 任务确定。专家系统知识库是面向特定领域、解决特定问题的。因此，知识工程师应先划定拟建系统的任务范围，明确知识库支持哪些问题的查询。

(2) 知识搜集。实际问题领域的专家往往并不熟悉知识的表示。他们经常凭借自身的专业直觉和累积经验来处理问题。比如专业医生对病因的诊断常常依赖于从医经验。这就要求知识工程师必须根据任务范围，迭代地与领域专家交流沟通，熟悉专家的思考模式、评判指标等，帮助提取经验知识。本阶段提取到的知识可视为是经验的初步总结，仍然以自然语言的形式表述。

(3) 确定词汇表。本步需要将所提取知识中的概念、实体、关系等转换为机器可识别的逻辑词汇，包括谓词、常量和函数。这些逻辑词汇可以本体或知识图谱的方式编码保存。

(4) 通用知识编码。除本体和知识图谱表示的概念、实体间的基本关系外，知识库的另一个重要组成部分是领域通用知识。通用知识一般只涉及概念和类别，以一阶逻辑形式写入知识库，称为公理。知识工程师需要注意所制定的公理应尽可能覆盖词汇表中所有的概念和类别。若存在未覆盖的类别，则应重新检查此概念是否是任务不相关的类别。

(5) 特定实例知识编码。关于实例的基本关系(如实例与类的关系、实例间的关联关系等)，在确定词汇表时就应该纳入考虑。这表现为本体或知识图谱的 ABox 部分(也有文献称为数据层)。除此之外，如果搜集知识中还包含另外的事实，本步骤将补充完整。补充的事实一般以原子语句的形式存在。

(6) 查询测试并调试。产生式规则库构建的最后一步是测试和调试。这需要通过一系列测试用例来调试知识库使其达到用户期望的状态。比如期望的链式推理在过程中意外停止了，那么原因可能是缺少某条公理所致。

专家系统的第二个组成部分是推理机。正如我们在一阶逻辑章节介绍的一样，基本推理规则有两种，即前向推理和反向推理。前向推理又称演绎推理，基本形式是

$$\frac{p,p\Rightarrow q}{q}$$

例如由知识“If it is raining, Then the street is wet”和条件“It is raining”可推出结论“The street is wet”。反向推理又称溯因推理，基本形式是

$$\frac{q,p\Rightarrow q}{p}$$

例如已知知识“If it is raining, Then the street is wet”和结论“The street is wet”，可以推出条件“It is raining”。推理机的推理过程可被视为是识别-动作(Recognize-Act)循环：

(1) 将工作内存中的事实陈述与规则前提匹配，称为模式匹配；

(2) 如果有多于一条规则被匹配成功，则根据冲突消解策略选择其中一条规则；如果没有规则匹配成功，则推理停止；

(3) 执行所选规则。执行结果可能是向工作内存中增加规则结论部分的事实陈述，也可能是删除已有的事实陈述。执行完毕后，若达到终止条件，则推理停止，否则转步骤 1。

识别-动作循环的终止条件一般设为某终止状态的达成，或循环次数达到预先设定的最

大值(如 100 次)。匹配是指将规则中的变量绑定为常量的过程。考虑以下知识库

$$IsHorse(x) \wedge Parent(x,y) \wedge Fast(y) \Rightarrow Valuable(x)$$

和事实

$$IsHorse(Comet), IsHorse(Prancer), Parent(Comet, Dasher),$$
$$Parent(Comet, Prancer), Fast(Prancer), Parent(Dasher, Thunder),$$
$$Fast(Thunder), IsHorse(Thunder), IsHorse(Dasher), IsLion(Aslan)$$

查询满足规则的 x 和 y 绑定。分别考察规则前提中的每一个文字。满足 $IsHorse(x)$ 的绑定包括置换

$$\{x/Comet\}, \{x/Prancer\}, \{x/Thunder\}, \{x/Dasher\}$$

满足 $Fast(y)$ 的绑定包括置换

$$\{y/Prancer\}, \{y/Thunder\}$$

满足 $Parent(x,y)$ 的绑定为

$$\{x/Comet, y/Dasher\}, \{x/Comet, y/Prancer\}, \{x/Dasher, y/Thunder\}$$

规则前提是合取式,寻找上述三个置换集合中的公共置换,得到满足规则的绑定为

$$\{x/Comet, y/Prancer\}, \{x/Dasher, y/Thunder\}$$

因此执行推理规则得到新的事实:$Valuable(Comet)$ 和 $Valuable(Dasher)$。推理机将这两条新事实加入工作内存中。需要注意的是,本例中是一条规则同时匹配成功多条事实,因此并不存在冲突。这与识别-动作循环指出的同时满足多条规则而调用冲突消解策略是有区别的。显然,本例是使用的前向推理。推理将不断添加新的事实到工作内存中,直至无法推出新的事实为止。该过程可用图 3-4 所示的流程表示。与之相对,反向推理是从结论出发(Then 部分),不断查找当前工作内存中是否已经包含能够得到目标结论的某条规则条件(If 部分)。若所有能导出结论的规则条件均不包含,则将条件作为子目标加入搜索。反向推理的过程如图 3-5 所示。推理的后半段将得到的结论加入工作内存。可以看到,前向推理和反向推理具有以下区别:

(1) 两种推理都是向着匹配规则数增加的方向进行的(因为越来越多的其他可能规则会被查询)。因此,前向规则适合用于起始状态较少的情况,而反向推理更适合于目标状态较少的情况。这样做能够扩展搜索路径,便于找到可行的策略。

(2) 前向推理适合于有新的事实到来,并在此基础上推理得到其他潜在知识的场景。而反向推理适合于有明确查询结论的场景。

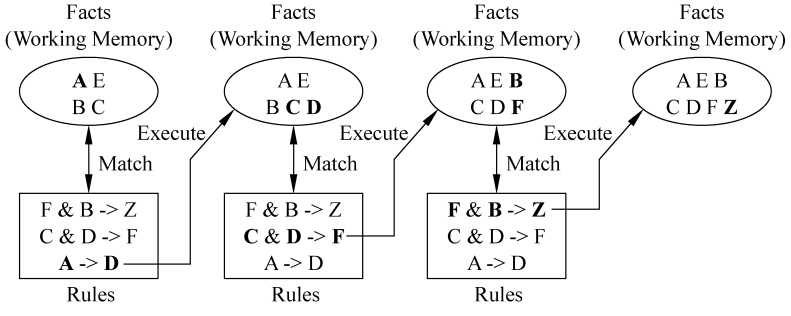

图 3-4　前向推理过程

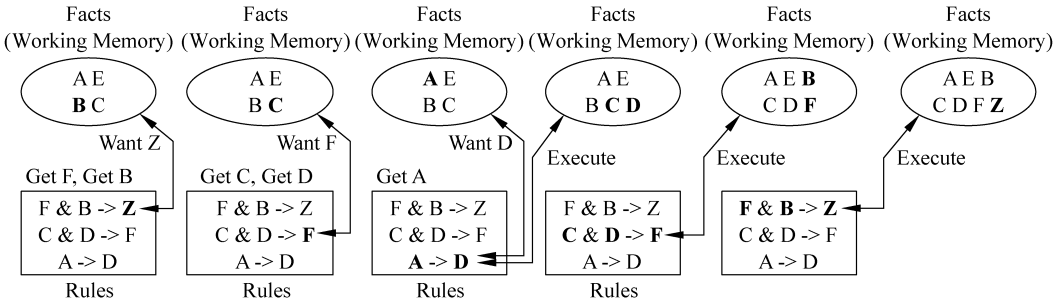

图 3-5　反向推理过程

直接的前向和反向推理是低效的，这在产生式规则系统中是一个经常碰到的问题。以前向推理为例，考虑有 n 条规则的知识库，每条规则平均带有 m 个条件文字，工作内存中有 k 条事实，那么系统在每一轮推理时将检查所有事实与所有规则条件文字的匹配程度，即要检查 $n \cdot m \cdot k$ 种可能的匹配。注意到不同规则的条件文字可能存在重复，并且当新的事实到来时，只有部分工作内存需要更新。基于这样的结构相似性和时间冗余性，卡内基梅隆大学的 Forgy 博士提出了 Rete 算法来提高推理效率[16]。Rete 在拉丁文中的意思是“网络”(net)，即先将所有规则编码成一个网络(称为规则编译)，然后再对工作内存中的事实做匹配(称为运行时执行)。Rete 网络如图 3-6 所示，包含根结点、类型结点、alpha 结点和 beta 结点四类。根结点是虚拟结点，用来创建网络。类型结点用于检查变量参数的类型，起到对事实过滤的作用。Alpha 结点用于记录事实对应的文字。Beta 结点代表规则条件。每一条多文字条件的规则都对应着一个 beta 结点。当 beta 结点被激活时，相应的规则也被激活，其推理输出的集合称为冲突集。Rete 网络编译算法是：

(1) 创建根结点。

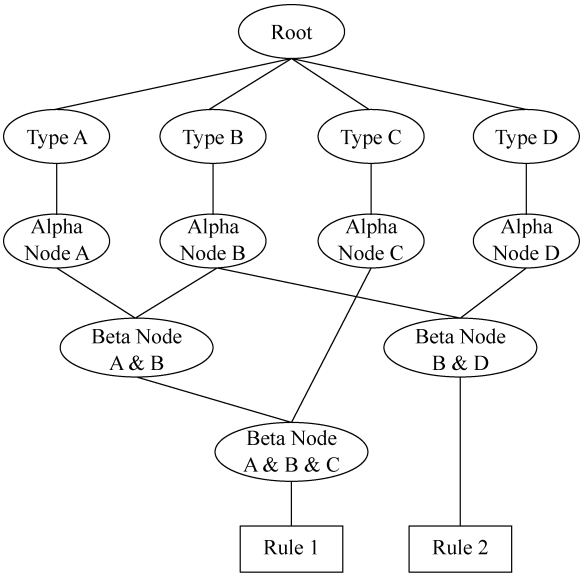

图 3-6　Rete 网络

(2) 加入规则 1：①取出文字 1，检查参数类型结点是否存在。若不存在则创建该类型结点。②检查文字 1 对应的 alpha 结点是否存在。若存在则记录结点位置；若不存在则创建，并建立 alpha 结点的内存表。③重复①和②直到所有的文字完成。④以 alpha(1)和 alpha(2)为父结点生成 beta(2)结点，以 beta($i-1$)和 alpha(i)为父结点生成 beta(i)结点($i>2$)，并将两个父结点的内存表连接成自身的内存表。⑤重复 d 完成所有 beta 结点。⑥将 beta(i)对应的规则结论作为输出结点。

(3) 重复(2)步直至所有规则编译完成。

Rete 网路对事实的匹配过程是：

(1) 对于工作内存中的每条事实，使用类型结点过滤，使其沿着网络到达合适的 Alpha 结点。

(2) Alpha 结点在收到事实后进行匹配。如果成功则记录并使其到达适当的 Beta 结点。

(3) Beta 结点将判断来自父结点的两个事实，如果匹配成功就记录并继续往后继 Beta 结点传递。

(4) 若匹配最终到达某一个输出结点，则将输出结论加入到冲突集中。

匹配完成后，系统只需考察冲突集是否包含新的事实。Rete 算法的实质是将过去匹配成功的事实信息保存在网络中，从而减少不必要的重复推理，是一种以存储空间换推理时间的方法。

在识别-动作循环中，冲突消解是指当事实成功匹配多于两条规则时，需要选取其中的一条规则执行。常用的冲突消解策略分为：删除已经触发规则的原事实直至冲突集为空(避免无限循环)；为工作内存中的事实赋予相应的生成时间，选取最新事实所匹配的第一个条件文字规则执行(使用最符合现实情况的条件)；选取条件中匹配文字数最多的规则(使用最具体的事实)；随机选择等。

3.6　本章小结

逻辑推理最初源于定理自动证明的研究，是人工智能早期发展的结晶。逻辑推理重点强调推理过程的正确性(Soundness，即所得结论一定与知识库一致)和完备性(Completeness，即推理一定能得到结论)。这两条性质也成为所有逻辑学家关注的基本特征。推理的基本方法有前向推理、反向推理和归结推理。Herbrand 定理是归结原理的理论基础。在逻辑编程中，Prolog 容易掌握，但采用深度搜索使其推理效率较低。LISP 则是更加底层的逻辑语言，它赋予了程序员更大的自由度和灵活性。逻辑推理在早期的直接应用是专家系统或产生式规则系统。系统由工作内存和规则库组成。在识别-动作循环的模式下，专家系统能够模拟人类领域专家思考、求解问题的过程，因而广泛地应用于多个行业。逻辑推理也可以和本体、知识图谱等其他知识表示形式结合，甚至可以引入概率来完成更为多样化的推理。

参考文献

[1] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle[J]. Journal of the ACM, 1965,12(1): 23-41.

[2] A. Leitsch. The Resolution Calculus. EATCS Monographs in Theoretical Computer Science[M], Springer,1997: 11.

[3] A. Horn. On sentences which are true of direct unions of algebras[J]. Journal of Symbolic Logic, 1951,16(1): 14-21.

[4] W. F. Dowling and J. H. Gallier. Linear-time algorithms for testing the satisfiability of propositional Horn formulae[J]. Journal of Logic Programming,1984,1(3): 267-284.

[5] R. M. Smullyan. First-order Logic[M]. Springer-Verlag,Berlin,Heidelberg,1968.

[6] J. A. Robinson. Computational Logic: The Unification Computation[J]. Machine Intelligence,1971,6: 63-72.

[7] C. McBride. First-Order Unification by Structural Recursion[J]. Journal of Functional Programming, 2003,13(6): 1061-1076.

[8] S. R. Buss. Handbook of Proof Theory[M]. Elsevier,1998.

[9] E. J. Gerritse. Herbrand's Theorem. Bachelor Thesis,Department of Computer Science[M],Radboud University,2016.

[10] S. R. Buss. On Herbrand's Theorem. In D. Leivant (eds),Logic and Computational Complexity,LCC 1994,Lecture Notes in Computer Science[M],Springer,Berlin,Heidelberg,1995,960: 195-209.

[11] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory[J]. Journal of the ACM,1960,7(3): 201-215.

[12] J. Beckford, G. Logemann and D. Loveland. A Machine Program for Theorem Proving[J]. Communications of the ACM,1962,5(7): 394-397.

[13] J. W. Lloyd. Foundations of Logic Programming[M]. Springer-Verlag,Berlin,1984.

[14] W. F. Clocksin and C. S. Mellish. Programming in Prolog[M]. Springer-Verlag,Berlin,2003.

[15] P. Jackson. Introduction to Expert Systems. 2nd Edition[M]. Addison-Wesley,1990.

[16] C. Forgy. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem[J]. Artificial Intelligence,1982,19: 17-37.