4.4 Logic 语言编程

::: details INFO
译者：kam1usec

来源：4.4 Logic Programming

对应：无
:::

在本节中，我们将介绍一种专门为本节内容设计的声明式查询语言 logic。这是一种基于 Prolog 和 计算机程序结构与解释 的声明式语言。logic 解释器是基于前面章节的模式项目的一个完全实现。其中，数据记录被表示为模式列表，查询语句被表示为模式值。

4.4.1 事实和查询

数据库存储着代表系统中事实的记录。而查询解释器的目的是直接从数据库记录中检索事实集合，并使用逻辑推理从数据库中推导出新的事实。logic 中的事实语句由一个或多个跟随关键字 "fact" 的列表组成。简单事实是一个单独的列表。例如，一个对美国总统感兴趣的犬舍老板可以使用 logic 记录她家狗狗的家谱，如下所示：

> 译者注：abraham（亚伯拉罕）、barack(巴拉克)、clinton（克林顿）、eisenhower（艾森豪威尔）等等为总统名；在 logic 语言中，数据存储（数据库）指的是一个集合，其中包含了一些已知事实或关系，这些事实或关系可以被查询解释器引用并用于推理和判断过程中。

```
(fact (parent abraham barack))
(fact (parent abraham clinton))
(fact (parent delano herbert))
(fact (parent fillmore abraham))
(fact (parent fillmore delano))
(fact (parent fillmore grover))
(fact (parent eisenhower fillmore))
```

在以逻辑为基础的系统中，关系通常不像函数或过程一样被应用，而是与查询相匹配。这句话的意思是，当向数据库发出查询请求时，系统会尝试将查询语句与现有的关系进行匹配，以便找到相关信息。查询指定正在寻找的信息（例如，“谁是巴拉克的父母？”），系统会尝试找到与查询匹配的关系（例如，“狗 - 亚伯拉罕是巴拉克的父亲”）。一旦找到一个匹配的关系，系统就可以使用逻辑推理来推导新的信息或根据数据库中已有的事实得出结论。这种逻辑表示和推理方法可以更灵活地推理实体之间的复杂关系。

一个查询包括一个或多个列表，以关键词“query”开头。查询可能包含变量，这些变量是以问号开头的字符串。查询解释器会将这些变量与事实进行匹配：

```
(query (parent abraham ?child))
Success!
child: barack
child: clinton
```

查询解释器会返回 `Success!` 以表示该查询与数据库中的一些事实相匹配，并且下面列出了匹配变量 `?child` 的替换结果。

复合事实：事实可能包含变量以及多个子表达式。一个多表达式的事实以结论开头，后跟假设。为了使结论成立，所有的假设都必须被满足：

```
(fact <conclusion> <hypothesis0> <hypothesis1> ... <hypothesisN>)
```

> `<conclusion>` 代表结论，`<hypothesis0> <hypothesis1> ... <hypothesisN>` 代表前提条件或假设，可以包含变量和多个子表达式。

例如，可以根据数据库中已有的关于父母的事实来声明有关孩子的事实。

```
(fact (child ?c ?p) (parent ?p ?c))
```

以上的事实可以理解为：如果 `?p` 是 `?c` 的父母，则 `?c` 是 `?p` 的孩子。现在，这个查询可以引用这个事实。

```
(query (child ?child fillmore))
Success!
child: abraham
child: delano
child: grover
```

上述查询需要查询解释器将定义孩子的事实与 `Fillmore` 作为父母的事实相结合。用户不需要知道如何组合这些信息，只需要知道结果具有特定的形式。查询解释器需要根据可用的事实证明 `(child abraham fillmore)` 为真。

查询不一定需要包含变量。它可以简单地验证一个事实，如果为真，并返回一个 `Success!`。

```
(query (child herbert delano))
Success!
```

如果查询没有匹配到事实，那么解释器会返回一个 `Failed`。

```
(query (child eisenhower ?parent))
Failed.
```

否定。我们可以使用特殊关键词 `not` 来检查查询是否不与任何事实匹配。

```
(query (not <relation>))
```

这个查询会在 `<relation>` 匹配失败时返回 `Success!`，而在 `<relation>` 匹配成功时返回 `Failed`。这个方法被称为否定即失败。

```
(query (not (parent abraham clinton)))
Failed.
(query (not (parent abraham barack)))
Failed.
```

使用（not）关键词时，若是查询不与事实匹配，结果为真，查询与事实匹配，结果反而为假。我们可以思考以下查询的结果：

```
(query (not (parent abraham ?who)))
```

为什么这个查询会返回 `Failed`？有许多值可以绑定到 `?who`，使得这个查询匹配成功，如果按照 `not` 查询的步骤，那么我们首先要检查关系 `(parent abraham ?who)`。因为 `?who` 可以绑定任意值，即 `barack` 或 `clinton`，所以这个关系为真。因此这个关系的否定查询会返回 `Failed`。

4.4.2 递归事实

logic 语言允许递归事实。也就是说，一个事实的结论可能取决于包含相同符号的假设。例如，血缘关系是用两个事实定义的。如果某个 `?a` 是 `?y` 的祖先，那么它既可以是 `?y` 的父母，也可以是 `?y` 的祖先的父母：

```
(fact (ancestor ?a ?y) (parent ?a ?y))
(fact (ancestor ?a ?y) (parent ?a ?z) (ancestor ?z ?y))
```

A single query can then list all ancestors of herbert:
因此一个查询就可以列出 herbert 的所有祖先。

```
(query (ancestor ?a herbert))
Success!
a: delano
a: fillmore
a: eisenhower
```

复合查询。查询可以包含多个子表达式，所有这些子表达式由相同的符号变量组成。如果一个变量在查询中出现多次，则它在每个上下文中必须取相同的值。使用以下查询查找 Herbert 和 Barack 的祖先：

```
(query (ancestor ?a barack) (ancestor ?a herbert))
Success!
a: fillmore
a: eisenhower
```

递归事实可能需要很长的推理链才能将查询与数据库中的已有事实相匹配，例如要证明 `(ancestor fillmore herbert)` 的事实，我们必须连续证明以下每个事实：

```
(parent delano herbert)       ; (1), a simple fact
(ancestor delano herbert)     ; (2), from (1) and the 1st ancestor fact
(parent fillmore delano)      ; (3), a simple fact
(ancestor fillmore herbert)   ; (4), from (2), (3), & the 2nd ancestor fact
```

这样，只要查询解释器能够发现它们，这个事实意味着大量附加事实甚至无限多的事实。

层级事实。到目前为止，每个事实和查询表达式都是一个符号列表。事实和查询列表可以包含列表，并提供一种表示分层数据的方法。接下来，我们把每只狗的颜色与作为附加记录的名称一起存储：

```
(fact (dog (name abraham) (color white)))
(fact (dog (name barack) (color tan)))
(fact (dog (name clinton) (color white)))
(fact (dog (name delano) (color white)))
(fact (dog (name eisenhower) (color tan)))
(fact (dog (name fillmore) (color brown)))
(fact (dog (name grover) (color tan)))
(fact (dog (name herbert) (color brown)))
```

查询可以表达层级事实的完整结构，也可以将变量与整个列表匹配。

```
(query (dog (name clinton) (color ?color)))
Success!
color: white
(query (dog (name clinton) ?info))
Success!
info: (color white)
```

数据库的许多能力在于查询解释器能够在单个查询中结合多种类型的事实。以下查询查找所有颜色相同且其中一只是另一只祖先的狗对：

```
(query (dog (name ?name) (color ?color))
       (ancestor ?ancestor ?name)
       (dog (name ?ancestor) (color ?color)))
Success!
name: barack color: tan ancestor: eisenhower
name: clinton color: white ancestor: abraham
name: grover color: tan ancestor: eisenhower
name: herbert color: brown ancestor: fillmore
```

变量可以引用分层记录中的列表，也可以使用点符号。跟在点符号后面的变量匹配事实列表的其余部分。点分列表可以出现在事实或查询中。下面的例子通过列出犬类祖先的血统链条来构建狗的家谱。（年轻的巴拉克沿袭了一条历史悠久的总统狗的血统。）

```
(fact (pedigree ?name) (dog (name ?name) . ?details))
(fact (pedigree ?child ?parent . ?rest)
      (parent ?parent ?child)
      (pedigree ?parent . ?rest))
(query (pedigree barack . ?lineage))
Success!
lineage: ()
lineage: (abraham)
lineage: (abraham fillmore)
lineage: (abraham fillmore eisenhower)
```

声明式或逻辑编程以非常高效的方式表达事实之间的关系。例如，如果我们希望两个列表可以连接成一个更长的列表，这个长列表包含第一个列表的元素，后跟第二个列表的元素。我们可以制定两个规则：首先，规定一个基本操作声明，将一个空列表添加到任何列表都会得到该列表；

```
(fact (append-to-form () ?x ?x))
```

第二，在一个递归事实中，一个以第一个元素为 `?a`，剩余部分为 `?r` 的列表与另一个列表 `?y` 拼接起来形成一个新列表，这个新列表以 `?a` 为第一个元素，而一些附加的剩余部分为 `?z`。为了使这种关系成立，必须确保 `?r` 和 `?y` 被拼接起来形成 `?z`。

```
(fact (append-to-form (?a . ?r) ?y (?a . ?z)) (append-to-form ?r ?y ?z))
```

查询解释器可以通过这两个事实，计算附加在一起的任意两个列表。

```
(query (append-to-form (a b c) (d e) ?result))
Success!
result: (a b c d e)
```

此外，它可以计算 `?left` 和 `?right` 所有可能的列表对，并且将它们拼接起来形成列表 `(a b c d e)`：

```
(query (append-to-form ?left ?right (a b c d e)))
Success!
left: () right: (a b c d e)
left: (a) right: (b c d e)
left: (a b) right: (c d e)
left: (a b c) right: (d e)
left: (a b c d) right: (e)
left: (a b c d e) right: ()
```

虽然我们的查询解释器可能显得相当智能，但是我们会看到它是通过多次重复一个简单操作来找到这些组合的：在数据库中匹配包含上述变量的两个列表。