Learn Prolog Now 翻译 - 第四章 - 列表 - 第二节，列表成员

内容提要

本章主要介绍使用递归操纵列表的一个实际例子：判断一个元素是否在包含在一个列表中。

是时候介绍第一个Prolog中通过递归操纵列表的程序例子了。我们最感兴趣的事情之一是，某个对象是否是列表中的元素。所以，我们想写一个程序，当假设输入是一个对象X和一个列表L，

得出结果是X是否属于L。这个程序的名字通常是：member，是Prolog程序中使用递归操纵列表最简单的例子，如下：

```
member(X, [X|T]).

member(X, [H|T]) :- member(X, T).
```

这就是全部的代码：一个事实（即member(X, [X|T])）和一个规则（即，member(X, [H|T]) :- member(X, T)）。但是请注意这个规则是递归的（因为函子：member同时出现在规则的头部和

主干），它能够解释为什么这么短的程序就能够达到要求，让我们进一步分析。

首先我们从声明性方面解读这段程序。通过这种解读，会发现它是完全有意义的。第一个子句（即事实）简单地说：如果对象X是一个列表的头部，那么X就是列表的元素。请注意我们使用了

内置的“|”操作符去操作列表。

那么第二个递归的子句呢？它的含义是：如果对象X是一个列表尾部的元素，那么它同时也是这个列表的元素。同样注意使用“|”操作符操作列表。

现在，很明显这个定义具备很好的声明性含义。但是这个程序是否会像它声明性含义一样的去运行？即，程序是否真的能够求出一个对象X是否是一个列表L的元素？如果是，是如何做到的？

为了回答这些问题，我们需要思考程序性的含义。让我们通过以下简单的列子得出结果。

假设我们进行如下的查询：

```
?- member(yolanda, [yolanda, trudy, vincent, jules]).
```

Prolog会立即回答true。为什么？因为程序通过第一个子句（即事实），将yolanda和X合一，所以答案就立即给出。

接下来，思考下面的查询：

```
?- member(vincent, [yolanda, trudy, vincent, jules]).
```

现在第一个子句无法提供答案（vincent和yolanda是不同的原子），所以Prolog继续使用第二个递归规则的子句，它会给出新的目标：

--Benjamin Yan

member(vincent, [trudy, vincent, jules]).

现在第一个子句再一次无法提供答案，所以Prolog继续使用第二个递归规则的子句，它又给出新的目标：

member(vincent, [vincent, jules]).

这一次，第一个子句会提供答案，查询也就成功了。

到此为止一切都还好，但是我们需要问一个重要的问题，如果我们进行的查询失败了会发生什么？比如，如果进行查询：

member(zed, [yolanda, trudy, vincent, jules]).

现在，这个查询明显会失败（毕竟，zed不在列表中）。所以Prolog会如何处理？特别地，我们如何确认Prolog会终止，并且返回false，而不是陷入递归无限循环中？

让我们通过系统性思考来解答这个问题。又一次地，第一个子句无法提供答案，所以Prolog使用递归规则，会给出新的目标：

member(zed, [trudy, vincent, jules])

同样地，第一个子句无法提供答案，所以Prolog使用递归规则并且给出新的目标：

member(zed, [vincent, jules])

同样地，第一个子句无法提供答案，所以Prolog使用递归规则并且给出新的目标：

member(zed, [vincent])

现在第一个子句还是无法提供答案，所以Prolog使用递归规则并且给出新的目标：

member(zed, [])

现在到了有趣的时候，明显地第一个子句还是无法提供答案。但是请注意：递归规则也无法起作用了，为什么？很简单：递归规则依赖于将列表分解为头部和尾部，正如我们之前看到的，

空列表是无法再根据这种方式拆分的。所以递归规则不能再起作用了，所以Prolog停止了进一步搜索解决方案，并且报告false。即，Prolog告诉我们zed不属于这个列表，也正是我们期望

的答案。

我们可以对member/2这个谓词逻辑进行总结：它是一个递归的谓词逻辑，会系统地遍历整个列表去搜索答案。它通过将列表分解为更小的列表，并且搜索每一个更小列表的头部去进行匹

配。这种方法会导致搜索是递归的，而且因为这种递归是安全的（因为，不会陷入无限循环），最终Prolog必须涉及到空列表。空列表无法再分解为更小的列表，所以就可以结束掉递归。

好了，我们现在已经完全明白member/2的工作原理了，但是事实上，这个谓词逻辑远比之前的例子有用。之前我们只是通过它回答一些true/false的例子，但是我们能够在查询中使用变量，

比如，如果我们进行查询：

?- memeber(X, [yolanda, trudy, vincent, jules]).

X = yolanda;

X = trudy;

X = vincent;

X = jules;

false

即，Prolog已经告诉我们一个列表的每一个元素，这是member/2一个特别普通的使用方式。本质上讲，通过变量，我们可以问Prolog： “快！给我这个列表的一些元素！”。在许多的

应用中我们需要从列表中获取元素，这种就是典型的方法。

最后提及一下，上述我们定义member/2的方式是正确的，但是有点冗余。

试想一下，第一个子句是处理列表的头部，虽然列表尾部在第一个子句中是无关的，但是我们依然使用了变量T对其进行合一。类似地，递归规则中处理的是列表的尾部，虽然列表的头部

是无关的，但是我们依然使用了变量H对其进行合一。这些不必要的变量可以被剔除：如果谓词逻辑的定义集中我们关注的概念，而无关的信息使用匿名变量处理，是一种更好的定义方式。

比如，我们对member/2进行如下的重构：

```
member(X, [X | _]).

member(X, [_ | T]) :- member(X, T).
```

这个版本从声明性和程序性上都是和第一个版本相同的。但是这个版本定义更加清晰：当你阅读的时候，你会集中在问题的本质上。

分类: Prolog

标签: Member

« 上一篇: Learn Prolog Now 翻译 - 第四章 - 列表 - 第一节，列表定义和使用
» 下一篇: Learn Prolog Now 翻译 - 第四章 - 列表 - 第三节，递归遍历列表