Learn Prolog Now 翻译 - 第三章 - 递归 - 第一节，递归的定义

在Prolog中，谓词可以递归地定义。简要地讲，一个谓词是递归定义的，如果一个或者多个规则的定义中包含了规则自身。

例子1：消化

考虑如下的知识库：

```
is_digesting(X, Y) :- just_ate(X, Y).
is_digesting(X, Y) :- just_ate(X, Z), is_digesting(Z, Y).

just_ate(mosquito, blood(john)).
just_ate(frog, mosquito).
just_ate(stork, frog).
```

第一眼看上面的知识库定义会感觉很简单：知识库中只包含了3个事实和2个规则。但是谓词is_digesting/2的定义是递归的。请注意is_digesting/2的定义中包含了自身

（至少部分的定义）。因为is_digesting/2的函子出现在第二个规则的头部和主干上；最重要的是，还存在一个”逃离“递归的定义，这个定义就是第一个规则中的谓词just_ate/2

（很明显，这个规则的定义中，主干部分没有提及谓词is_digesting/2）。让我们从声明性和程序性两个方面讨论这个定义。

单词”声明性”常常用于讨论Prolog知识库的逻辑含义。即，Prolog知识库的声明性简单来说就是“这个知识库在说什么？”，或者如果存在逻辑状况的集合，就是“这意味着什么？”。

上面的递归定义的含义十分直白：第一个子句（“逃离”子句，没有递归的那个子句，或者常常称为基础子句）简单地理解为：如果X已经把Y吃掉了，那么X就在消化Y。这明显是正确的定义。

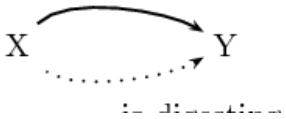
那么第二个子句呢，这是一个递归的子句。它在说：如果X已经吃掉了Z，并且Z在消化Y，那么X就在消化Y。这明显也是正确的定义。

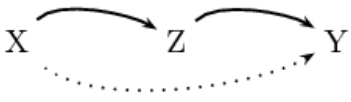
所以我们知道了这个递归定义的含义，但是当我们实际查询时是如何使用这个规则的呢？即，这个规则实际是如何工作的？使用Prolog的术语，它的程序意义是什么？

其实也很容易理解。第一个基础规则就像我们学习的之前的那些规则一样，即如果我们问X是否在消化Y，Prolog会运用这个规则将其转换为另一个问题：X是否直接吃掉Y？

那么递归那个子句呢？它给出了另外一种X是否在消化Y的策略：试图找到Z，X已经直接吃掉了Z，而且Z在消化Y。即这个规则让Prolog把问题分解为两个子问题，并且希望分解后的子问题

足够简单，可以直接在知识库中找到答案。下图总结了上面提及的两种情况：





让我们看看递归是如何运作的，如果我们进行查询：

```
? — is_digesting(stork, mosquito).
```

Prolog会像这样运作：首先，它会尝试使用关于is_digesting的第一个规则，即基础规则。这个规则说如果X直接吃掉Y了话X就在消化Y，通过将X和stork，Y和mosquito

合一，得到如下的目标：

```
just_ate(stork, mosquito).
```

但是知识库中并不存在这样的事实，所以这个尝试失败了。所以Prolog接着使用第二个规则尝试，通过将X和stork，Y和mosquito合一得到如下的目标：

```
just_ate(stork, Z), is_digesting(Z, mosquito).
```

即，为了满足is_digesting(stork, mosquito)，Prolog需要找到一个Z，既要满足：just_ate(stork, Z)，又要满足：is_digesting(Z, mosquito)，确实存在这样的Z，即frog。

5. Learn Prolog Now 翻译 - 第六章 - 列表补遗 - 第一节，列表合并(753)

评论排行榜

1. Learn Prolog Now 翻译 - 第一章 - 事实，规则和查询 - 第一节， 一些简单的例子(2)

推荐排行榜

1. Haskell学习笔记二：自定义类型(1)
2. Learn Prolog Now 翻译 - 第三章 - 递归 - 第四节，更多的实践和练习(1)

因为just_ate(stork, frog)直接能够满足，因为这是知识库中的事实；is_digesting(frog, mosquito)推导也很简单，因为is_digesting/2将其分解为新的目标：

just_ate(frog, mosquito)，这也是知识库中的事实。

上面是我们第一个递归的规则例子，我们接下来会学习更多。但是有一个实际的提醒要马上给出：当我们写一个递归的谓词逻辑时，应该至少包含两个子句：一个是基础子句（用于

在某些条件下停止递归），另一个是递归子句。如果没有这样做，那么Prolog就会陷入死循环中。比如，下面是一个及其简单的递归定义：

```
p :- p.
```

没有其他内容，这个定义很简单也很优雅。而且从声明性方面来看，也是说的通的：即如果p为真，那么p能够为真。但是从程序方面看，这个一个很危险的规则。事实上，任何只有

递归定义，而没有基础子句定义的规则都是危险的，因为我们无法结束递归。试想如果我们提出查询：

```
?- p.
```

Prolog会问自己：“我如何能够证明p？”，然后它会意识到：“哦，我有一个关于p的规则可以使用，如果p为真，我就能证明p”，所以它有问自己：“如何能够证明p为真”，然后

它又意识到：“哦，我有一个关于p的规则可以使用，如果p为真，我就能证明p”，就这样一直循环下去。如果你进行这样的查询，Prolog不会回答你，而是一直尝试搜索，不会停止，

直到你终止程序的运行。当然，你也可以进行跟踪调试，一步一步地跟上去，直到你看着Prolog的循环而生病^^。

例子2： 后辈

现在我们已经了解了Prolog中的递归是什么，那么为什么递归很重要呢？事实上，这个问题能够从很多不同的层次去回答，但是从实际编写Prolog的角度而言，递归是否真的如何重要？

如果是，为什么？

让我们思考下面的例子，如果我们有一个定义了“父子”关系的知识库如下：

```
child(bridget, caroline).

child(caroline, donna).
```

即，caroline是bridget的儿子，donna是caroline的儿子。现在，假设我们希望定义后辈的关系，即定义儿子，儿子的儿子，儿子的儿子的儿子，等等。下面是关于这个的第一次尝试，

我们在知识库中添加了两个非递归的规则：

```
descend(X, Y) :- child(X, Y).

descend(X, Y) :- child(X, Z), child(Z, Y).
```

现在，很明显这些规则能够满足要求，但是也有明确的限制：只能够定义两代和两代之内的后辈关系。如果假设我们有如下的一些事实：

```
child(anne, bridget).

child(bridget, caroline).

child(caroline, donna).

child(donna, emily).
```

那么我们之前定义的两个规则就不适用了，比如，如果我们查询：

```
?- descend(anne, donna).
```

或者

```
?- descend(bridget, emily).
```

Prolog会回答false，这不是我们期望的。当然，我们可以通过添加如下的两个规则去修复这个问题：

```
descend(X, Y):- child(X, Z_1), child(Z_1, Z2), child(Z_2, Y).

descend(X, Y) :- child(X, Z_1), child(Z_1, Z_2), child(Z_2, Z_3), child(Z_3, Y).
```

但是，让我们面对现实吧，这些规则定义很笨拙、可读性也很差。而且，如果我们有更深层次和更多的父子事实，关于后辈的规则就会愈发膨胀，就像：

```
descend(X, Y) :- child(X, Z_1), child(Z_1, Z_2), child(Z_2, Z_3), child(Z_3, Z_4), ..., child(Z_18, Z_19), child(Z_19, Y).
```

这可不是解决问题的有效方式！

但是我们不必这么做，我们可以避免定义过长过多的规则。下面的递归谓词逻辑就完美地符合了我们的期望：

```
descend(X, Y) :- child(X, Y).

descend(X, Y) :- child(X, Z), descend(Z, Y).
```

如何理解这个定义？上面谓词逻辑的基础子句的声明性含义是：如果Y是X的儿子，那么Y就是X的后辈，这明显是正确的。那么递归子句呢？它的声明性含义是：如果Z是X的儿子，并且Y

是Z的后辈，那么Y也是X的后辈。同样也是正确的。

我们继续通过一个例子分析上面递归谓词逻辑的程序性含义，如果我们查询：

```
?- descend(anne, donna).
```

Prolog会首先尝试使用第一个规则。规则头部的变量X和anne合一，变量Y和donna合一，所以Prolog会尝试证明：

```
child(anne, donna)
```

这个尝试会失败，因为知识库中没有这样的事实，也无法根据规则推导出。所以Prolog回溯，并且去找descned(anne, donna)的另外证明方式。它找到知识库中第二个规则，并且构成

如下的子目标列表：

```
child(anne, _G633), descend(_G633, donna).
```

Prolog首先取出第一个子目标并且尝试和知识库中的某些事实进行合一。它找到了事实child(anne, bridget)，并且将变量_G633初始化为bridget。第一个子目标已经满足，Prolog

尝试证明第二个子目标：descend(bridget, donna)。

这是谓词descend/2的第一次递归。和之前步骤类似，Prolog开始使用第一个规则，但是目标：child(bridget, donna)无法被证明。通过回溯，Prolog找到第二种证明目标的方式，即

使用规则2，然后Prolog会得出如下新的子目标列表：

```
child(bridget, _G178), descend(_G178, donna).
```

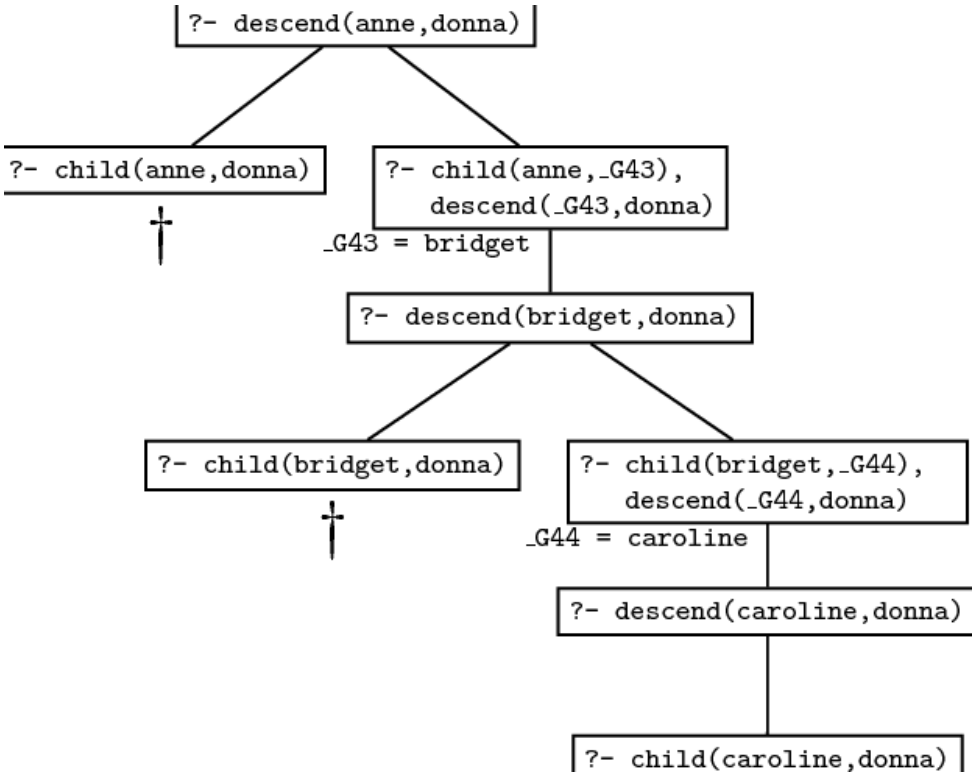
第一个子目标通过知识库中事实child(bridget, caroline)可以合一，所以变量_G178被初始化为caroline。接下来Prolog会尝试证明：descend(caroline, donna)。这是谓词逻辑

descend/2的第二次递归。和之前的步骤类似，首先尝试第一个规则，得到如下的新目标：child(caroline, donna)，这一次，Prolog成功了，由于child(caroline, donna)正是知识

库中的事实。Prolog已经证明了目标descend(caroline, donna)（第二次递归调用）是成功的，同时这意味着descend(bridget, donna)（第一次递归调用）也是成功的，同时也意味

着我们的原始查询descend(anne, donna)同样是成功的。

如下是查询descend(anne, donna)的搜索树。请确保你能够理解搜索树和上面的文字分析是如何对应的，即Prolog如何根据搜索树来证明原始查询是正确的：



很明显，无论我们添加多个代的儿子事实，总能够证明出后辈的关系。即，递归定义是通用和紧凑的：它包含了所有非递归规则的信息，而且还更多。非递归的规则只是根据固定的数字

生成几代的后辈关系，我们需要写无数的非递归规则才能获取完整的信息，但是这是不可行的。本质上说，这就是递归规则对我们的意义：它能够通过两三行代码组合出任何代的后辈信息。

递归的规则是十分重要的，它能够将海量的信息以紧凑的形式表达出，并且很自然地定义出谓词逻辑。作为一个Prolog程序员最常做的工作就是定义各种递归规则。

例子3：Successor

在上一章学习中，我们说到通过合一构建结构是Prolog编程的一个关键点。现在我们知道递归也是关键点，下面是一个有趣的演示。

当前，如果人类使用数字，一般会使用十进制表示法（0,1,2,3,4,5,6,7，8，8,10,11,12，等等），但是，你可能知道，还有其他的很多表示法。比如，因为计算机硬件普遍都是基于

数字电路，计算机通常使用二进制表示数字（0,1,10,11,100,101,110,111,1000，等等），其中通过开关关闭代表0，开关闭合代表1。另外一些文化使用了不同的表示法。比如，古巴比

伦人使用60进制，而古罗马人使用比较特殊的表示法（I,II,III,IV,V,VI,VII,VIII,IX,X）。最后罗马人的例子说明，数字表示法是很重要的，不信的话，你可能尝试使用罗马数字做做

大数字的除法，你会发现，这是一个艰难的工作。很明显罗马人有一个专家团队来专门做这个事情（据现代会计师分析认为的）。

现在有另外一种数字表示法，有时候会被运用在数学逻辑方面。这种表示法只使用了4个符号：0，succ，和左右小括号。这种数字表示法可以使用如下的指令定义：

1. 0是一个数字。
2. 如果X是一个数字，那么succ(X)也是数字。

succ就是successor的缩写。即，succ(X)代表的数字是X代表的数字加1.所以这是一种非常简单的表示法：它简单地认为0是一个数字，并且所有的其他数字都是通过前端累加succ符号构

建的。（事实上，正式由于其表达的简单性，这种表示法经常被用于数学逻辑上，虽然用它做财务计算很困难，但是用它证明一些东西确实很管用）

现在，我们把这种表示法的定义转换为Prolog的程序，如下面的知识库所示：

```
numeral(0).
numeral(succ(X)) :- numeral(X).
```

所以，如果我们进行查询：

```
?- numeral(succ(succ(succ(0)))).
```

Prolog会回答true。但是我们可以做一些更有趣的事情，比如当我们进行查询：

```
?- numeral(X).
```

即，我们说：”好吧，给我显示一些数字吧！“，Prolog会回答会构成下面的图形：

```
X = 0;
X = succ(0);
X = succ(succ(0));
X = succ(succ(succ(0)));
X = succ(succ(succ(succ(0))));
X = succ(succ(succ(succ(succ(0)))));
...
```

是的，Prolog在数数，但是重要的是，它如何做到的？十分简单，它通过递归定义进行回溯，通过合一实际地构建数字。这是一个标志性的例子，理解它很重要。最好的方法就是坐下并

实际操作，并且开启trace一步一步看如何运行。

构建和绑定，递归，合一，证明搜索，这些都是Prolog编程的核心概念。当我们需要生成或者分析递归结构的对象时，使用这些概念使得Prolog成为一种强有力的工具。比如，在下一章

里面，我们会介绍列表，一个十分重要的递归数据结构，同时我们也会看到Prolog是一门天生处理列表的语言。许多的应用程序（计算机语言学是最主要的例子）十分依赖递归结构对象的

使用，比如像树和特征结构体。所以Prolog十分擅长构建这类应用程序也就不足为奇了。

例子4： 加法

最后一个例子，我们看看是否能够通过上一节的数字表达方式，进行一些简单的运算。我们尝试定义加法，即我们定义一个谓词逻辑add/3，其中前两个参数作为加数，最后一个参数作为

结果返回。比如：

```
?- add(numeral(succ(0)), numeral(succ(succ(0))), numeral(succ(succ(succ(0))))).
```

Prolog会回答true；

```
?- add(numeral(succ(0)), numeral(succ(succ(0))), Y).
```

Prolog会回答：Y = numeral(succ(succ(succ(0)))).

这里有两个重要的提示：

1. 无论什么情况下，如果第一个参数为0，那么第三个参数一定和第二个参数相等

```
?- add(numeral(0), numeral(succ(succ(0))), Y).

Y = numeral(succ(succ(0))).

?- add(numeral(0), numeral(0), Y).

Y = numeral(0).
```

这是我们基础子句需要的。

2. 假设我们把X和Y进行加和（比如numeral(succ(succ(succ(0))))和numeral(succ(succ(0)))），并且X不是numeral(0)。那么，如果X1是比X少一层succ的数字（即

numeral(succ(succ(0)))）如果我们知道X1和Y加和的结果——比如称为Z（等于
numeral(succ(succ(succ(succ(0))))），那么就十分容易计算X和Y的加和：我们只需要在Z的结果中多加

一层succ。这就是我们递归子句需要的。

如下就是根据我们之前描述定义的谓词逻辑：

```
add(numeral(0), Y, Y).

add(numeral(succ(X)), Y, numeral(succ(Z))) :- add(numeral(X), Y, numeral(Z)).
```

那么如果我们进行如下的查询，会发生什么？

```
?- add(numeral(succ(succ(succ(0)))), numeral(succ(succ(0))), R).
```

当我们一步一步地分析Prolog如何进行此次查询。相关的跟踪和搜索树如下：

因为第一个参数不是numeral(0)，就意味着只有add/3的第二个字句能够使用。这会导致递归地调用add/3。第一个参数最外层的succ会被去掉，而且结果会称为递归查询的第一个参数；

第二个参数会被原封不动地传入递归查询中，第三个参数在递归查询中是一个变量，即追踪过程中的中间变量
_G648；注意变量_G648还没有被初始化，但是它和R共享值（R是我们用于原始

查询中的第三个参数，即结果值），因为根据第二个字句，R会被初始化为numeral(succ(_G648))；同样R在此时也还没有完成初始化，它这时是一个复杂语句，并且有一个未初始化的变量

在其中。

第二步本质上是相同的。在这一步中，第一个参数会更少一层succ，追踪和搜索树都很明确地展示了这点。同时，通过每一步，succ会被加在R中，并且保持最里层的变量没有被初始化。

经过第一轮递归后，R变成numeral(succ(_G648))，经过第二轮递归后，_G648被初始化为succ(_G650)，所以R变成
numeral(succ(succ(_G650)))；经过第三轮递归，_G650被初始

化为succ(_G652)，R则变成numeral(succ(succ(succ(_G652))))。搜索树会一步一步展示这种初始化信息。

当第一个参数的所有succ都被去掉，我们就会使用基础子句。然后第三个参数就会等于第二个参数，所以代表R的复杂语句中的“洞”（没有被初始化的变量）会被最终填上，结果就得出

了。如下是完成的查询追踪：

```
Call: (6) add(numeral(succ(succ(succ(0)))), numeral(succ(succ(0))), R)

Call: (7) add(numeral(succ(succ(0))), numeral(succ(succ(0))), _G648)

Call: (8) add(numeral(succ(0)), numeral(succ(succ(0))), _G650)

Call: (9) add(numeral(0), numeral(succ(succ(0))), _G652)
```

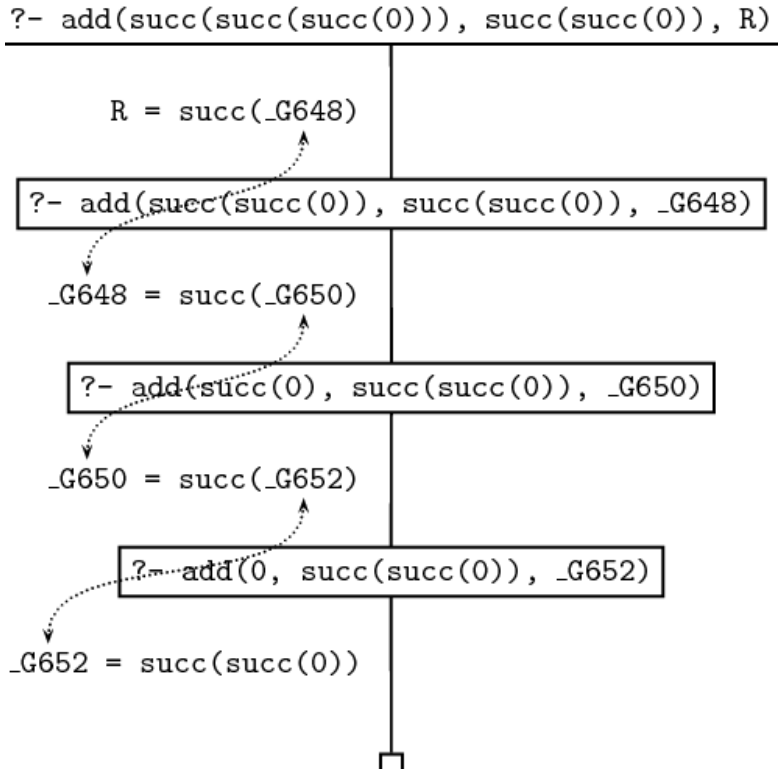
```
Exit: (9) add(numeral(0), numeral(succ(succ(0))), numeral(succ(succ(0))))

Exit: (8) add(numeral(succ(0)), numeral(succ(succ(0))), numeral(succ(succ(succ(0)))))

Exit: (7) add(numeral(succ(succ(0))), numeral(succ(succ(0))), numeral(succ(succ(succ(succ(0))))))

Exit: (6)  add(numeral(succ(succ(succ(0)))), numeral(succ(succ(0))),
numeral(succ(succ(succ(succ(succ(0)))))))
```

如下是搜索树：



分类: Prolog

标签: 递归, recursive

« 上一篇: Learn Prolog Now 翻译 - 第二章 - 合一和证明搜索 - 第三节， 练习题和答案
» 下一篇: Learn Prolog Now 翻译 - 第三章 - 递归 - 第二节，规则顺序，目标顺序，终止

posted on 2015-07-07 09:53 seaman.kingfall 阅读(1169) 评论(0) 编辑 收藏