Learn Prolog Now 翻译 - 第九章 - 语句深究 - 第三节， 语句的检查

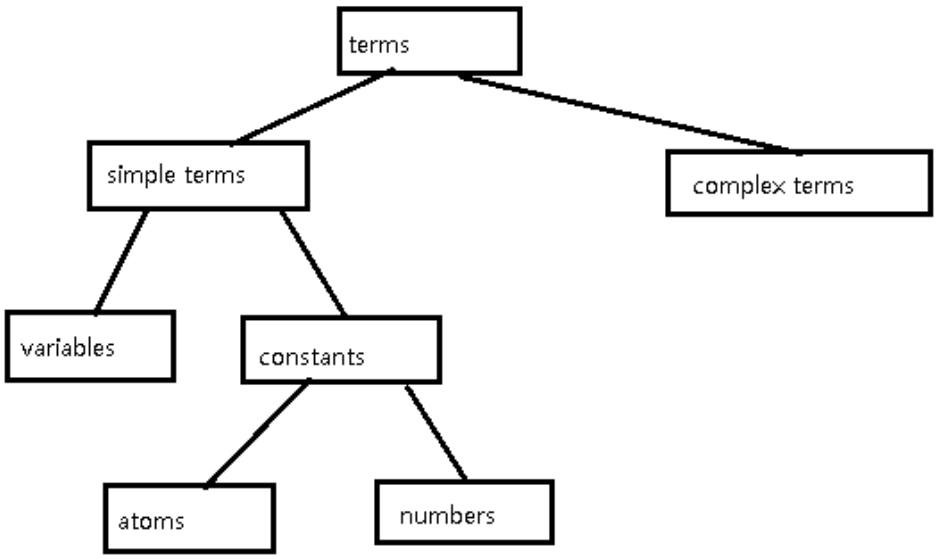
内容提要

- 语句的类型检查
- 语句的结构检查
- 字符串

在本节中，我们将学习一些内置谓词，这些谓词可以对语句进行检查。首先，我们将会看到的谓词会测试其参数的语句是否为某种特定的类型（比如，是否是原子或者数字）。接下来，会介绍一些可以揭示复杂语句结构的谓词。

语句的类型检查

回忆一下在第一章我们学习关于Prolog语句的内容：一共有四种类型的语句，分别为：变量，原子，数字和复杂语句。更进一步来说，原子和数字可以归类为常量，常量和变量归类为简单语句。下面的树状图总结的这些内容：



有时能够知道给出的语句的类型是什么是很用的。比如，你也许想要写一个谓词，其中不得不对不同类型的语句，选择不同的处理方式。Prolog提供了几个内置的谓词，能够检查语句是否是特定的类型：

atom/1：检查参数语句类型是否为原子?

integer/1：检查参数语句类型是否为整数?

float/1：检查参数语句类型是否为浮点数?

number/1：检查参数语句类型是否为数字?

atomic/1：检查参数语句类型是否为常量?

var/1：检查参数语句类型是否为未初始化的变量?

nonvar/1：检查参数语句类型是否为已经被初始化的变量或者其他非未初始化变量的类型?

让我们观察一下这些谓词的行为：

```
?- atom(a).
true

?- atom(7).
false

?- atom(loves(vincent, mia)).
false
```

这里的三个例子的结果正如我们的期望。但是，如果我们将变量作为参数传入atom/1，会发生什么?

```
?- atom(X).
false
```

这很好理解，因为一个为初始化的变量不是一个原子。但是如果我们将X初始化为一个原则，然后再查询atom(X)，Prolog会回答true:

```
?- X = a, atom(X).
X = a
true
```

但是这里重要的一点是初始化必须在检查之前完成:

```
?- atom(X), X = a.
false
```

谓词integer/1和float/1的作用类似，可以自己尝试一些例子。

下面来学习number/1和atomic/1两个谓词。首先，number/1会检查语句是否为整数或者浮点数，即，number/1为真如果integer/1为真或者float/1为真，number/1为假如果integer/1和float/1都为假。其次是atomic/1，会检查语句是否为常量，即语句是否为原子或者数字。所以，atomic/1为真如果atom/1或者number/1为真，atomic/1为假如果atom/1和number/1都为假。

```
?- atomic(mia).
true

?- atomic(8).
true

?- atomic(3.25).
true

?- atomic(loves(vincent, mia)).
false

?- atomic(X)
false
```

那么变量如何检查呢？首先有var/1这个谓词，可以检查语句是否为未初始化的变量:

```
?- var(X).
true

?- var(mia).
false

?- var(8).
false
```

```
?- var(3.25)
false

?- var(loves(vincent, mia)).
false
```

其次还有nonvar/1谓词，当var/1为假时，这个谓词结果为真，即它可以检查语句不是未被初始化的变量：

```
?- nonvar(X).
false

?- nonvar(mia).
true

?- nonvar(8).
true

?- nonvar(3.25).
true

?- nonvar(loves(vincent,mia)).
true
```

请注意一个包含了未初始化变量的复杂语句本身不是一个未初始化的变量（它还是一个复杂语句），所以：

```
?- var(loves(_, mia)).
false

?- nonvar(loves(_, mia)).
true
```

当X被初始化，var(X)和nonvar(X)的结果依赖于它们是在X初始化前被调用，还是后被调用，如下：

```
?- X = a, var(X).
false

?- X = a, nonvar(X).
X = a
true

?- var(X), X = a.
X = a
true

?- nonvar(X), X = a.
false
```

语句的结构检查

假设有一个未知结构的复杂语句（比如一些谓词将复杂语句作为输出参数），我们希望从中获取哪些信息？明显的回答是：它的函子，元数，及其参数看上去是什么样的。Prolog提供了内置的谓词可以获取这些信息。谓词functor/3可以获取复杂语句的函子和元数信息。比如：

```
?- functor(f(a, b), F, A).
A = 2
F = f
true

?- functor([a, b, c], X, Y).
X = '.'
```

```
Y = 2
true
```

注意到查询列表时，Prolog返回的函子是.，这是列表在Prolog内部的表示法。

当对常量使用functor/3时，会发生什么？让我们试试：

```
?- functor(mia, F, A).
A = 0
F = mia
true

?- functor(8, F, A).
A = 0
F = 8
true

?- functor(3.25, F, A).
A = 0
F = 3.25
true
```

所以我们可以使用谓词functor/3找出任何语句的函子和元数，包括一些元数为0的语句（比如常量）。

我们也能够使用functor/3构建语句。如何做？通过将第二个和第三个参数传入具体的值，而第一个参数是未初始化的变量，比如：

```
?- functor(T, f, 7).
T = f(_G286, _G287, _G288, _G289, _G290, _G291, _G292)
true
```

请注意第一个，或者第二个和第三个参数必须被初始化。比如，如果查询 functor(T, f, N)，Prolog就会报错。如果你想查询的问题有意义，Prolog会用合理的方式回答。但是要求Prolog构建一个复杂语句但是不告诉它需要几个参数，这就不是有意义的要求了，Prolog就处理不了。

现在我们已经学习了functor/3，让我们实际使用它完成一些工作。在本节前面的内容中，我们讨论了一些内置的检查语句类型的谓词，包括检查语句是否为：原子，数字，常量，或者变量。但是没有谓词可以检查语句是否为复杂语句。为了使得检查类型谓词列表完整，我们自己定义一个谓词。使用functor/3可以很容易完成，我们要做的就是检查参数是否是一个合理的函子，而且有输入参数（即，它的元数大于0），下面是定义：

```
?- complexterm(X) :-
    nonvar(X),
    functor(X, _, A),
    A > 0.
```

函子到此为止，那么参数呢？作为functor/3的补充，Prolog为我们提供了谓词arg/3，可以获取复杂语句中的参数。它获取数字N和复杂语句T，返回复杂语句第N个参数，可以使用这个谓词访问一个参数的值：

```
?- arg(2, loves(vincent, mia), X).
X = mia
true
```

或者初始化参数：

```
?- arg(2, loves(vincent, X), mia).
X = mia
true
```

试图访问一个不存在的参数，会失败：

```
?- arg(2, happy(yolanda), X)
false
```

谓词functor/3和arg/3允许我们访问所有我们希望知道的关于复杂语句的信息，同时，Prolog还提供了第三个内置谓词：=../2（等号后面加上两个句号），它会获取一个复杂语句，并且返回一个列表，其中复杂语句的函子作为列表头元素，复杂语句的所有参数，构成列表的尾部，所以，如果我们查询：

```
?- =..(loves(vincent, mia), X).
X = [loves, vincent, mia]
```

这个谓词（也称为univ）能够被当作中缀操作符使用，如下：

```
?- cause(vincent, dead(zed)) =.. X.
X = [cause, vincent, dead(zed)]
true

?- X =.. [a, b(c), d].
X = a(b(c), d)
true

?- footmassage(Y, mia) =.. X.
Y = _G303
X = [footmassage, _G303, mia]
true
```

当某些情况下，需要访问一个复杂语句所有的参数时，univ会派上用场，由于返回的是一个列表，所以所有列表操作的方法和策略都可以使用。

字符串

Prolog中通过字符（ASCII）编码的列表来表示字符串。但是，由于使用列表语法去操作字符串很别扭，所以Prolog提供了更加友好的字符串语法：使用双引号，尝试一下查询：

```
?- S = "Vicky".
S = [86, 105, 99, 107, 121]
true
```

这里将变量S和字符串"Vicky"合一，实际上是有5个数字的列表，每一个数字代表一个字符编码。（比如，86是字符V的编码，105是字符i的编码，等等）

换句话讲，Prolog中的字符串实际上是数字的列表。绝大多数的Prolog方言都提供了标准的一些谓词来处理字符串。一个特别有用的谓词是atom_codes/2，它会把一个原子转换为一个字符串。下面的例子展示了atom_codes/2能够做些什么：

```
?- atom_codes(vicky, X).
X = [118, 105, 99, 107, 121]
true

?- atom_codes('Vicky', X).
X = [86, 105, 99, 107, 121]
true

?- atom_codes('Vicky Pollard', X).
X = [86, 105, 99, 107, 121, 32, 80, 111, 108 | ...]
true
```

还有另外一种使用，atom_codes/2的方式，通过字符串生成原子。假设你想要将一个原子abc变成另一个原子abcabc，那么你可以这么做：

```
?- atom_codes(abc, X), append(X, X, L), atom_codes(N, L).
X = [97, 98, 99]
L = [97, 98, 99, 97, 98, 99]
N = abcabc
```

最后一件需要知道的事情是，和atom_codes/2相关的另外一个内置谓词，是number_codes/2，这个谓词行为方式很前者很类似，但是正如名字所示，只能作用于数字。