**NLP\_chapter8**

**第8章 分析句子结构**

**8.1 一些语法困境**

**语言数据和无限可能性**

"现代英语”并不等同于一个在过去 50 年中英文表达或写成的一切构成的巨大的语料库中的非常大的词序列的集合。说英语的人可以判断这些序列，并将拒绝其中一些不合文法的。

同样，组成一个新的句子，并让说话者认为它是非常好的英语是很容易的。例如:句子有一个有趣的属性:它们可以嵌入更大的句子中。

如果用符号 S 表示整个句子，可以看到像这样的模式:Andre said S and I think S。这些模板可以得到一个句子，并构建一个更大的句子。还有其他的可以使用的模板，如:S but S and S when S。稍微动点儿心思，就可以使用这些模板构建一些很长的句子。

语言提供给我们的结构，看上去可以无限扩展句子。这也是惊人的，我们能理解我们从来没有听说过的任意长度的句子:并不难假造一个全新的句子，一个在语言的历史上可能从来没有使用过的句子，而所有说这种语言的人都能理解它。

文法的目的是给出一个明确的语言描述。而我们思考文法的方式与我们认为这是一种语言紧密联系在一起。

在这一章中，将采取“生成文法”的形式化框架，其中一种“语言”被认为是所有合乎文法的句子的巨大集合，一个文法是一个形式化符号，可用于“产生“这个集合的成员。 文法使用 S → S and S 形式的递归产生式，

**普遍存在的歧义**

歧义的例子：

While hunting in Africa, I shot an elephant in my pajamas. How an elephant got into my pajamas I’ll never know.

短语 I shot an elephant in my pajamas 中存在歧义。

用下面的程序来分析句子：

groucho\_grammar = nltk.CFG.fromstring('''

S -> NP VP

PP -> P NP

NP -> Det N | Det N PP | 'I'

VP -> V NP | VP PP

Det -> 'an' | 'my'

N -> 'elephant' | 'pajamas'

V -> 'shot'

P -> 'in'

''')

sent = ['I', 'shot', 'an', 'elephant', 'in', 'my', 'pajamas']

parser = nltk.ChartParser(groucho\_grammar)

for tree in parser.parse(sent):

print(tree)

尽管相关的所有词的含义是没有歧义的，但是这个短语可以有两种理解方式，取决于介词短语 in my pajamas 是描述大象还是枪击事件。

**8.2 文法有什么用?**

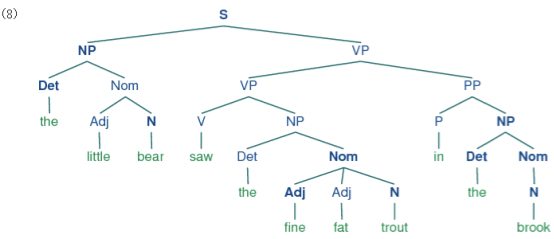
**超越 n-grams**

**并列结构**，两个短语通过并列连词如 and、but 或 or 连结在一起。下面是连词的语法功能的一个非正式(并且简单)的描述。

并列结构:如果 *v1* 和 *v2* 都是文法类型 X 的短语，那么 *v1 and v2* 也是 X 类型的短语。例如，两个NP(名词短语)连结在一起组成一个NP，其次，两个AP(形容词短语)连结在一起组成一个AP。而不能连结一个 NP 和一个 AP。

**成分结构**基于对词与其他词结合在一起形成单元的观察。一个词序列形成这样一个单元的证据是它是**可替代**的——也就是说，在一个符合语法规则的句子中的词序列可以被一个更 小的序列替代而不会导致句子不符合语法规则。事实上，形成一个单元的每个序列可以被一个单独的词替换，最终只有两个元素。

如果从最上面的行剥离出词汇，增加一个 S 节点，翻转图，最终得到一个标准的短语结构树。此树的每个节点(包括词)被称为一个**组成部分(成分，constituent)**。S 的**直接组成部分**是 NP 和 VP：



因为句子可以有任意的长度，所以短语结构树可以有任意深度。7.4 节中的级联块分析器只能产生有限深度的结构，使得分块方法在这里并不适用。

**8.3 上下文无关文法**

**一种简单的文法**

首先，看一个简单的上下文无关文法(context-free grammar，CFG)。按照惯例，第一条生产式的左端是文法的开始符号，通常是 S，所有符合语法规则的树都必须有这个符号作为它们的根标签。NLTK中，上下文无关文法定义在 nltk.grammar 模块。在下例中，定义了文法，并显示如何分析一个简单的符合文法的句子：

grammar1 = nltk.CFG.fromstring('''

S -> NP VP

VP -> V NP | V NP PP

PP -> P NP

V -> "saw" | "ate" | "walked"

NP -> "John" | "Mary" | "Bob" | Det N | Det N PP

Det -> "a" | "an" | "the" | "my"

N -> "man" | "dog" | "cat" | "telescope" | "park"

P -> "in" | "on" | "by" | "with"

''')

sent = "Mary saw Bob".split()

rd\_parser = nltk.RecursiveDescentParser(grammar1)

for tree in rd\_parser.parse(sent):

print(tree)

该例中的文法包含涉及各种句法类型的产生式，如下表所列出的：

符号 意思 例子S 句子 the man walked

NP 名词短语 a dog

VP 动词短语 saw a park

PP 介词短语 with a telescope

Det 限定词 the

N 名词 dog

V 动词 walked

P 介词 in

产生式如 VP -> V NP | V NP PP 右侧脱节了，中间显示|，这是两个产生式 VP -> V NP和VP -> V NP PP的缩写。

如果使用上例显示的文法分析句子 The dog saw a man in the park，会得到两棵树。由于这句话的两棵树都符合文法规则，这句话被称为结构上有歧义。这个歧义问题被称为介词短语附着歧义。这是一个附着歧义，因为 PP in the park 需要附着在树中两个位置中的一个:要么是 VP 的孩子要么是 NP 的孩子。当 PP 附着在 VP 上，合适的解释是:看到公园里发生的事情。然而，如果 PP 附着在 NP 上，意思就是:在公园里的一个人，看到(狗)的主语可能已经坐在公寓的阳台上俯瞰公园。

**写自己的文法**

可以在一个文本文件 mygrammar.cfg 中创建和编辑自己的语法(注意文件中不能包含’等符号，只需输入语法文本，可以加注释)，然后，可以加载它到 NLTK 中，并按如下方式用它进行分析:

grammar1 = nltk.data.load('file:mygrammar.cfg')

sent = "Mary saw Bob".split()

rd\_parser = nltk.RecursiveDescentParser(grammar1)

for tree in rd\_parser.parse(sent):

print(tree)

确保文件名后缀为.cfg，并且字符串'file:mygrammar.cfg '中间没有空格符。如果命令 print tree 没有产生任何输出，这可能是因为句子 sent 并不符合自定的文法。遇到这种情况可以将分析器的跟踪设置打开 :

rd\_parser = nltk.RecursiveDescentParser(grammar1, trace=2)

还可以查看当前使用的文法中的产生式，使用命令 :

for p in grammar1.productions():

print (p)

写 CFG 在 NLTK中分析时，不能将文法类型与词汇项目一起写在同一个产生式的右侧。因此，产生式PP -> 'of' NP是不允许的。另外，不得在产生式右侧仿制多个词的词汇项。因此，不能写成 NP -> 'New York'，而要写成类似 NP -> 'New\_York'这样的。

**句法结构中的递归**

如果文法类型出现在产生式左侧也出现在右侧，一个文法被认为是递归的，如下例所示。产生式 Nom -> Adj Nom(其中 Nom 是名词性的类别)包含 Nom 类型的直接递归， 而 S 上的间接递归来自于两个产生式的组合:S -> NP VP 与 VP -> V S：

grammar2 = nltk.CFG.fromstring('''

S -> NP VP

NP -> Det Nom | PropN

Nom -> Adj Nom | N

VP -> V Adj | V NP | V S | V NP PP

PP -> P NP

PropN -> 'Buster' | 'Chatterer' | 'Joe'

Det -> 'the' | 'a'

N -> 'bear' | 'squirrel' | 'tree' | 'fish' | 'log'

Adj -> 'angry' | 'frightened' | 'little' | 'tall'

V -> 'chased' | 'saw' | 'said' | 'thought' | 'was' | 'put'

P -> 'on'

''')

可以是嵌套的名词短语，也可以是嵌套的句子。

递归深度是没有限制的，但是，RecursiveDescentParser是无法处理形如 X -> X Y 的**左递归**产生式的。

**8.4 上下文无关文法分析**

**分析器**根据文法产生式处理输入的句子，并建立一个或多个符合文法的组成结构。文法是一个格式良好的声明规范——它实际上只是一个字符串，而不是程序。分析器是文法的解释程序。它搜索符合文法的所有树的空间找出一棵边缘有所需句子的树。

**递归下降分析**

一种最简单的分析器将一个文法作为如何将一个高层次的目标分解成几个低层次的子目标的规范来解释。顶层的目标是找到一个 S。S→NP VP 生产式允许分析器替换这个目标为两个子目标:找到一个 NP，然后找到一个 VP。每个这些子目标都可以再次被子目标的子目标替代，使用左侧有 NP 和 VP 的产生式。最终，这种扩张过程达到子目标，如:找到词 telescope。这样的子目标可以直接与输入序列比较，如果下一个单词匹配就取得成功。如果没有匹配，分析器必须备份，并尝试其它选择。

递归下降分析器在上述过程中建立分析树。带着最初的目标(找到一个 S)，创建 S 根节点。随着上述过程使用文法的产生式递归扩展，分析树不断向下延伸(故名为递归下降)。

在这个过程中，分析器往往被迫在多种可能的产生式中选择。当某一步不起作用时就回溯，按顺序尝试其他产生式，直到与输入句子的下一个词相匹配。如此循环直到发现一个完整的分析树。这是一个涵盖了整个句子的树， 没有任何悬着的边。发现了分析树后可以让分析器寻找其它额外的分析树。它会再次回溯和探索选择其它产生式，以免漏掉任何一个产生分析树的情况。

NLTK 提供了一个递归下降分析器: RecursiveDescentParser()，正是前面例子中所用到的，它接受一个可选的参数 trace。如果 trace 大于零， 则分析器将报告它解析一个文本的步骤。

递归下降分析有**三个主要的缺点**。首先，左递归产生式，如:NP -> NP PP，会进入死循环。第二，分析器浪费了很多时间处理不符合输入句子的词和结构。第三，回溯过程中可能会丢弃分析过的成分，它们将需要在之后再次重建。例如:从VP -> V NP上回溯将放 弃为NP 创建的子树。如果分析器之后处理 VP -> V NP PP，那么 NP 子树必须重新创建。

递归下降分析是一种**自上而下分析**。自上而下分析器在检查输入之前先使用文法预测输入将是什么!然而，由于输入对分析器一直是可用的，从一开始就考虑输入的句子会是更明智的做法。这种方法被称为**自下而上分析**。

**移进-归约分析**

一种简单的自下而上分析器是**移进-归约分析器**。与所有自下而上的分析器一样，移进-归约分析器尝试找到对应文法生产式右侧的词和短语的序列，用左侧的替换它们，直到整个句子归约为一个 S。

移位-规约分析器反复将下一个输入词推到堆栈，这是**移进**操作。如果堆栈上的前 n 项，匹配一些产生式的右侧的 n 个项目，那么就把它们弹出栈，并把产生式左边的项目压入栈。这种替换前 n 项为一项的操作就是**规约**操作。此操作只适用于堆栈的顶部;规约栈中的项目必须在后面的项目被压入栈之前做。当所有的输入都使用过，堆栈中只剩余一个项目，也就是一颗分析树作为它的根的 S 节点时，分析器完成。移位-规约分析器通过上述过程建立一颗分析树。每次弹出堆栈 n 个项目，它就将它们组合成部分的分析树，然后将这压回推栈。

NLTK 中提供了 ShiftReduceParser()，移进-归约分析器的一个简单的实现。这个分析器不执行任何回溯，所以它不能保证一定能找到一个文本的解析，即使确实存在一个这样的解析。此外，它最多只会找到一个解析，即使有多个解析存在。可以提供一个可选的 trace 参数，控制分析器报告它分析一个文本的步骤的繁琐程度(以跟踪模式运行，可以看到序列的移进和规约操作)：

sr\_parser = nltk.ShiftReduceParser(grammar1,trace=2)

sent = 'Mary saw a dog'.split()

for tree in sr\_parser.parse(sent):

print(tree)

移进-规约分析器可能会到达一个死胡同，而不能找到任何解析，即使输入的句子是符合语法的。这种情况发生时，没有剩余的输入，而堆栈包含不能被规约到一个 S 的项目。问题出现的原因是:较早前做出的选择不能被分析器撤销(虽然图形演示中用户可以撤消它们的选择)。分析器可以做两种选择 :(a)当有多种规约可能时选择哪个规约，(b)当移进和规约都可以时选择哪个动作。

移进-规约分析器可以改进执行策略来解决这些冲突。例如:它可以通过只有在不能规约时才移进，解决移进-规约冲突;它可以通过优先执行规约操作，解决规约-规约冲突;它可以从堆栈移除更多的项目。

移进-规约分析器相比递归下降分析器的好处是，它们只建立与输入中的词对应的结构。 此外，每个结构它们只建立一次。

**左角落分析器**

**左角落分析器**是一个带自下而上过滤的自上而下的分析器。不像普通的递归下降分析器，它不会陷入左递归产生式的陷阱。在开始工作之前，左角落分析器预处理上下文无关文法建立一个表，其中每行包含两个单元，第一个存放非终结符，第二个存放那个非终结符可能的左角落的集合。

分析器每次考虑产生式时，它会检查下一个输入词是否与左角落表格中至少一种非终结符的类别相容。

**符合语句规则的子串表**

为了弥补简单的分析器在完整性和效率上的限制，运用**动态规划**算法设计技术分析问题。动态规划存储中间结果，并在适当的时候重用它们，能显著提高效率。

这种技术可以应用到句法分析，使我们能够存储分析任务的部分解决方案，然后在必要的时候查找它们，直到达到最终解决方案。这种分析方法被称为**图表分析**。

动态规划使我们能够只建立一次 PP in my pajamas。第一次建立时就把它存入一个表格中，然后在需要作为对象 NP 或更高的 VP 的组成部分用到它时就查找表格。这个表格被称为**符合语法规则的子串表**或简称为 WFST。(术语“串”指一个句子中的连续的词序列。)

以“I shot an elephant in my pajamas”这个句子为例。

在 WFST 中，通过填充三角矩阵中的单元记录词的位置:纵轴表示一个子串的起始位置，而横轴表示结束位置(从而 shot将出现在坐标(1, 2)的单元中)。为了简化这个演示，假定每个词有一个独特的词汇类别，将在矩阵中存储词汇类别(不是词)。 所以单元(1, 2)将包含条目 V。更一般的，如果输入的字符串是 a1a2 ... an，文法中包含一个形为 A → ai 的产生式，那么把 A 添加到单元(i-1, i)。

对于 text 中的每个词，可以在设定的文法中查找它所属的类别：

text = ['I', 'shot', 'an', 'elephant', 'in', 'my', 'pajamas']

print(groucho\_grammar.productions(rhs=text[1]))

对于WFST，用 Python 中的链表的链表创建一个(n-1)×(n-1)的矩阵，在下例中的函数 init\_wfst()中用每个标识符的词汇类型初始化它。再定义一个实用的函数 display()来精美的输出WFST。正如预期的那样，V在(1, 2)单元中：

def init\_wfst(tokens,grammar):

numtokens = len(tokens)

wfst = [[None for i in range(numtokens+1)] for j in range(numtokens+1)]

for i in range(numtokens):

productions = grammar.productions(rhs=tokens[i])

wfst[i][i+1] = productions[0].lhs()

return wfst

def complete\_wfst(wfst,tokens,grammar,trace=False):

index = dict((p.rhs(),p.lhs()) for p in grammar.productions())

numtokens = len(tokens)

for span in range(2,numtokens+1):

for start in range(numtokens+1-span):

end = start + span

for mid in range(start+1,end):

nt1,nt2 = wfst[start][mid],wfst[mid][end]

if nt1 and nt2 and (nt1,nt2) in index:

wfst[start][end] = index[(nt1,nt2)]

if trace:

print("[%s] %3s [%s] %3s [%s] ==> [%s] %3s [%s]" % \

(start,nt1,mid,nt2,end,start,index[(nt1,nt2)],end))

return wfst

def display(wfst,tokens):

print('\nWFST '+' '.join(("%-4d"%i) for i in range(1,len(wfst))))

for i in range(len(wfst)-1):

print("%d " % i, end=" ")

for j in range(1,len(wfst)):

print("%-4s"%(wfst[i][j] or '.'),end=" ")

print()

tokens = 'I shot an elephant in my pajamas'.split()

wfst0 = init\_wfst(tokens,groucho\_grammar)

display(wfst0,tokens)

wfst1 = complete\_wfst(wfst0,tokens,groucho\_grammar)

display(wfst1,tokens)

回到表格表示，假设对于词 an 有 Det 在(2, 3)单元，对于词 elephant 有 N在(3, 4)单元，对于 an elephant 应该找到一个形如 NP → Det N 的产生式，放入(2, 4)。查询了文法，可以输入(0, 2)单元的 NP。

更一般的，如果有一个产生式 A → B C，并且在 (i, k) 中找到非终结符 B，在(k, j)中找到非终结符 C，可以在 (i, j)输入 A。下例中的程序使用此规则完成 WFST。通过调用函数complete\_wfst() 时设置 trace 为 True，显示 WFST 正在被创建的跟踪输出:

wfst1 = complete\_wfst(wfst0,tokens,groucho\_grammar,trace=True)

为了更简便地通过产生式的右侧检索产生式，为文法创建一个索引。这是空间-时间权衡的一个例子:对语法做反向查找，每次想要通过右侧查找产生式时不必遍历整个产生式链表。

可以得出结论:只要已经在(0, 7)单元构建了一个 S 节点，表明已经找到了一个涵盖了整个输入的句子，就为整个输入字符串找到了一个解析。

WFST 有几个缺点。首先，WFST 本身不是一个分析树，所以该技术严格地说是认识到一个句子被一个文法承认，而不是分析它。其次，它要求每个非词汇文法生产式是二元的。第三，作为一个自下而上的方法，它潜在的存在浪费，它会在不符合文法的地方提出成分。最后，WFST 并不能表示句子中的结构歧义(如两个动词短语的读取)。(2, 8)单元中的 VP 实际上被输入了两次，一次是读取 V NP，一次是读取 VP PP。这是不同的假设，第二个会覆盖第一个(虽然如此，这并不重要，因为左侧是相同的。)图表分析器使用稍微更丰富的数据结构和一些有趣的算法来解决这些问题

**8.5 依存关系和依存文法**

短语结构文法是关于词和词序列如何结合起来形成句子成分的。一个独特的和互补的方式，**依存文法**，集中关注的是词与其他词之间的关系。依存关系是一个**中心词**与它的**依赖**之间的二元对称关系。一个句子的中心词通常是动词，所有其他词要么依赖于中心词，要么依赖路径与它联通。

依存关系表示是一个加标签的有向图，其中节点是词汇项，加标签的弧表示依赖关系，从中心词到依赖。

依存关系图中的弧可以添加依赖与它的中心词之间的语法功能标签。例如: I 是shot(这是整个句子的中心词)的 SBJ(主语)，in 是一个 NMOD(elephant 的名词修饰语)。与短语结构文 法相比，依存文法可以作为一种依存关系直接用来表示语法功能。

下面是 NLTK 为依存文法编码的一种方式(左边为中心词，右边为它的依赖)——注意它只能捕捉依存关系信息，不能指定依存关系类型:

groucho\_dep\_grammar = nltk.DependencyGrammar.fromstring('''

'shot' -> 'I' | 'elephant' | 'in'

'elephant' -> 'an' | 'in'

'in' -> 'pajamas'

'pajamas' -> 'my'

''')

print(groucho\_dep\_grammar)

依存关系图是一个**投影**，当所有的词都按线性顺序书写 ，便可以在词上绘制而不会交叉。这等于是说一个词及其所有后代依赖(依赖及其依赖的依赖，等等)在句子中形成一个连续的词序列。可以使用投影依存关系分析器分析很多英语句子。下面的例子演示 groucho\_dep\_grammar 如何提供了一种替代的方法来捕捉之前在研究短语结构语法中遇到的附着歧义：

pdp = nltk.ProjectiveDependencyParser(groucho\_dep\_grammar)

sent = 'I shot an elephant in my pajamas'.split()

trees = pdp.parse(sent)

for tree in trees:

print(tree)

输出：

(shot I (elephant an (in (pajamas my))))

(shot I (elephant an) (in (pajamas my)))

这些括号括起来的依存关系结构也可以显示为树，依赖被作为它们的中心词的孩子。

在比英语具有更多灵活的词序的语言中，非投影的依存关系也更常见。

在一个成分 C 中决定哪个是中心词 H，哪个是依赖 D，已经提出了很多不同的标准。 最重要的有以下几种:

1. H 决定类型 C 的分布;或者，C 的外部句法属性取决于 H。

2. H 定义 C 的语义类型。

3. H 必须有而 D 是可选的。

4. H 选择 D 并且决定它是必须有的还是可选的。

5. D 的形态由 H 决定(如 agreement 或 case government)。

当在一个短语结构文法中说:一个 PP 的直接成分是 P 和 NP 时，这就隐含了中心词/依赖之间的区别。介词短语是一个短语，它的中心词是一个介词。此外，NP 是 P 的依赖。同样的区别在其它类型的短语结构文法中也存在。这里要注意的关键点是:虽然短语结构文法看上去似乎与依存关系文法非常不同，但是它们隐含着依存关系。虽然 CFG 不会直接捕获依存关系，最近的语言框架已越来越多地采用这两种方法的结合的形式。

**配价与词汇**

观察动词和它们的依赖。was 可以与跟在其后的形容词一起出现，saw 可以与跟在其后的 NP 一起出现，thought 可以与跟在其后的 S 一起出现，thought 可以与跟在其后的 NP 和 PP 一起出现。依赖 ADJ、NP、PP 和 S 通常被称为各自动词的**补语**，什么动词可以和什么补语一起出现具有很强的约束。

在依存文法的传统中，不同的动词被认为具有不同的**配价**。配价限制不仅适用于动词，也适用于其他类的中心词。

在 CFG 中，需要一些限制文法产生式的方式，使得扩展了 VP 后动词只与它们正确的补语一同出现。可以通过将动词划分成更多“子类别”做到这个，每个子类别与一组不同的补语关联。例如:**及物动词（TV, transitive verb）**，如:chased 和 saw，需要后面跟 NP 对象补语;它们是 NP 大类别的**子类别**。

动词子类别符号 含义 例子

IV Intransitive verb barked

TV Transitive verb saw a man

DatV Dative verb gave a dog to a man

SV Sentential verb said that a dog barked

配价是一个词项的属性。补语往往与修饰语对照，虽然两者都是依存关系的类别。

介词短语、形容词和副词通常充当修饰语。由于不同修饰语是可选的，经常可以进行迭代，不会像补语那样被中心词选择。

**扩大规模**

即使允许使用各种形式化的工具，它们可以提供文法产生式更简洁的表示，但保持对覆盖一种语言的主要成分所需要的众多产生式之间的复杂的相互作用的控制，仍然是极其困难的。换句话说，很难将文法模块化，每部分文法可以独立开发。反过来这意味着，在一个语言学家团队中分配编写文法的任务是很困难的。另一个困难是当文法扩展到包括更加广泛的成分时，适用于任何一个句子的分析的数量也相应增加。换句话说，歧义随着覆盖而增加。

尽管存在这些问题，一些大的合作项目在为几种语言开发基于规则的文法上已取得了积极的和令人印象深刻的结果。例如:词汇功能语法(Lexical Functional Grammar，LFG)Pargram 项目、中心词驱动短语结构文法(Head-Driven Phrase Structure Grammar，HPSG) LinGO 矩阵框架、词汇化树邻接文法 XTAG项目。

**8.6 文法开发**

**树库和文法**

corpus 模块定义了树库语料的阅读器，其中包含了宾州树库语料的10%的样本。

from nltk.corpus import treebank

t = treebank.parsed\_sents('wsj\_0001.mrg')[0]

print(t)

可以利用这些数据来帮助开发一个文法。下例中的程序使用一个简单的过滤器找出带句子补语的动词。假设已经有一个形如 VP -> Vs S 的产生式，这个信息使我们能够识别那些包括在Vs 的扩张中的特别的动词。搜索树库找出句子的补语：

def filter(tree):

child\_nodes = [child.label() for child in tree

if isinstance(child,nltk.Tree)]

return (tree.label() == 'VP')and('S' in child\_nodes)

print([subtree for tree in treebank.parsed\_sents()

for subtree in tree.subtrees(filter)][20])

PP 附着语料库，nltk.corpus.ppattach，是另一个有关特别动词配价的信息源。下例找出具有固定的介词和名词的介词短语对，其中介词短语附着到 VP 还是 NP，由选择的动词决定：

entries = nltk.corpus.ppattach.attachments('training')

table = nltk.defaultdict(lambda: nltk.defaultdict(set))

for entry in entries:

key = entry.noun1 + '-' + entry.prep + '-' + entry.noun2

table[key][entry.attachment].add(entry.verb)

for key in sorted(table):

if len(table[key])>1:

print(key,'N',sorted(table[key]['N']),'V:',sorted(table[key]['V']))

这个程序的输出行中有一行offer-from-group N: ['rejected'] V:['received']，这表示 received 期望一个单独的 PP 附着到 VP 而 rejected 不是的。和以前一样，可以使用此信息来帮助构建文法。

NLTK 语料库也收集了中央研究院树库语料，包括 10000 句已分析的句子，来自现代汉语中央研究院平衡语料库。加载并显示这个语料库中的一棵树：

nltk.corpus.sinica\_treebank.parsed\_sents()[3450].draw()

**有害的歧义**

随着文法覆盖范围的增加和输入句子长度的增长，分析树的数量也迅速增长。事实上，它以天文数字的速度增长。

没有实际的自然语言处理系统可以为一个句子构建数以百万计的树，并根据上下文选择一个合适的。很显然，人也不会这样做!

除了结构歧义之外，词汇歧义也同样大量存在。只要试图建立一个广泛覆盖的文法，就被迫使词汇条目对它们的词性高度含糊。事实上，所有的词都可以作为名字被提及:例如:he verb 'ate' is spelled with three letters;在讲话中，不需要使用引号。此外，大多数名词都可以动词化。因此，一个覆盖广泛的文法分析器将对歧义不堪重负。

这种歧义是不可避免的，导致在分析看似平淡无奇的句子时可怕的低效。概率分析提供了解决这些问题的方法，它以来自语料库的证据为基础对歧义句的解析进行排名。

**加权文法**

处理歧义是开发广泛覆盖的分析器的主要挑战。图表分析器提高了计算一个句子的多个分析的效率，但它们仍然因可能的分析的数量过多而不堪重负。加权文法和概率分析算法为这些问题提供了一个有效的解决方案。

符合语法的概念可能是有倾向性的。思考动词 give。这个动词既需要一个直接宾语(被给予的东西)也需要一个间接宾语(收件人)。这些补语可以按任何顺序出现，如(16)所示。在“介词格”的形式(16a)中，直接宾语先出现，然后是包含间接宾语单独介词短语。

(16) a. Kim gave a bone to the dog.

b. Kim gave the dog a bone.

在“双宾语”的形式(16b)中，间接宾语先出现，然后是直接宾语。在这种情况下，两种顺序都是可以接受的。然而，如果间接宾语是代词，人们强烈偏好双宾语结构:

(17) a. Kim gives the heebie-jeebies to me (prepositional dative).

b. Kim gives me the heebie-jeebies (double object).

使用宾州树库样本，可以检查包含 give 的所有介词格和双宾语结构的实例：

def give(t):

return t.label() == 'VP' and len(t)>2 and t[1].label() == 'NP'\

and (t[2].label() == 'PP-DTV' or t[2].label() == 'NP')\

and ('give' in t[0].leaves() or 'gave' in t[0].leaves())

def sent(t):

return ' '.join(token for token in t.leaves() if token[0] not in '\*-0')

def print\_node(t,width):

output = "%s %s: %s / %s: %s" %\

(sent(t[0]),t[1].label(),sent(t[1]),t[2].label(),sent(t[2]))

if len(output)>width:

output = output[:width]+'...'

print(output)

for tree in treebank.parsed\_sents():

for t in tree.subtrees(give):

print\_node(t,72)

可以观察到一种强烈的倾向就是最短的补语最先出现。然而，这并没有解释形如:give NP: federal judges / NP: a raise这样的句子，其中生命度起了重要作用。事实上，根据(Bresnan & Hay, 2008)的调查，存在大量的影响因素。这些偏好可以用加权文法来表示。

概率上下文无关文法(probabilistic context-free grammar，PCFG)是一种上下文无关文法，它的每一个产生式关联一个概率。它会产生与相应的上下文无关文法相同的文本解析，并给每个解析分配一个概率。PCFG 产生的一个解析的概率仅仅是它用到的产生式的概率的乘积。

最简单的方法定义一个 PCFG 是从一个加权产生式序列组成的特殊格式的字符串加载它，其中权值出现在括号里，如下例所示：

grammar = nltk.PCFG.fromstring('''

S -> NP VP [1.0]

VP -> TV NP [0.4]

VP -> IV [0.3]

VP -> DatV NP NP [0.3]

TV -> 'saw' [1.0]

IV -> 'ate' [1.0]

DatV -> 'gave' [1.0]

NP -> 'telescopes' [0.8]

NP -> 'Jack' [0.2]

''')

print(grammar)

有时可以很方便的将多个产生式组合成一行，如:VP -> TV NP [0.4] | IV [0.3]| DatV NP NP [0.3]。为了确保由文法生成的树能形成概率分布，PCFG 文法强加了约束: 产生式所有给定的左侧的概率之和必须为 1。parse()返回的分析树包含概率:

viterbi\_parser = nltk.ViterbiParser(grammar)

for tree in viterbi\_parser.parse(['Jack','saw','telescopes']):

print(tree)

这里不能直接用print函数输出，否则仅仅会返回内存区域。

现在，分析树被分配了概率，一个给定的句子可能有数量庞大的可能的解析就不再是问题。分析器将负责寻找最有可能的解析。