第11章 分析句子结构

本章目标

- 如何使用形式化语法来描述句子集合的结构?
- 如何使用句法树来表示句子结构?
- 语法分析器如何分析一个句子并自动构建语法树?

内容

- 结构分析的歧义
- 文法的作用
- 上下文无关文法
- 上下文无关文法分析
- 依存关系和依存文法
- 文法开发

In [1]:

```
import nltk
```

结构分析的歧义

由词组成词组乃至句子时，由于其组成的词或词组间可能存在不同的语法或语义关系而出现的（潜在）歧义现象

- “VP＋的＋是＋NP”型歧义结构
  - “反对 | 的 | 是 | 少数人”
- “N1＋N2＋N3”型歧义结构
  - “北欧 | 语言 | 研究会 ”
- “ADJ.＋N1＋N2”型歧义结构
  - “小 | 学生 | 词典”
- “VP＋N1＋的＋N2”型歧义结构
  - “咬死了 | 猎人 | 的 | 狗 ”
- “VP＋ADJ.＋的＋N”型歧义结构
  - “喜欢 | 干净 | 的 | 小孩 ”
- “V＋N1＋N2”型歧义结构
  - “赠 | 意大利 | 图书”
- “数量结构＋NP1＋的＋NP2”型歧义结构
  - “三个 | 学校 | 的 | 实验员”
- 英语的例子
  - Fighting animals could be dangerous.
  - Visiting relatives can be tiresome.

花园幽径句

- 我是县长派来的
- 我是县长

- Put the frog on the napkin in the box
- Put the frog on the napkin

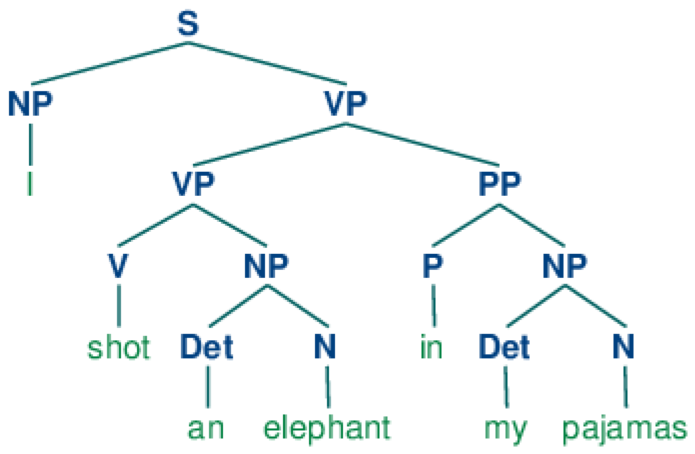
歧义和树结构

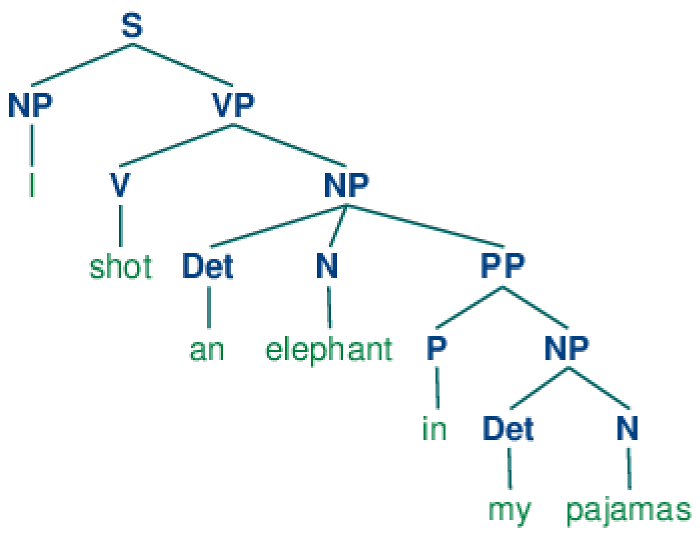
While hunting in Africa, I shot an elephant in my pajamas. How he got into my pajamas, I don't know.

In [2]:

```
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)
```

```
(S
  (NP I)
  (VP
    (VP (V shot) (NP (Det an) (N elephant)))
    (PP (P in) (NP (Det my) (N pajamas)))))
(S
  (NP I)
  (VP
    (V shot)
    (NP (Det an) (N elephant) (PP (P in) (NP (Det my) (N pajamas))))))
```





上下文无关文法

第一条产生式的左端是文法的开始符号，通常是S

In [3]:

```
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)
```

```
(S (NP Mary) (VP (V saw) (NP Bob)))
```

文法的作用

“产生式文法”形式化框架

- “语言”被认为是所有合乎文法的句子的集合
- 文法是一组形式化符号，可用于“产生”这个集合的成员

成分结构

- 在符合语法规则的句子中，词序列可以被更短的序列替代，而不会导致句子不符合语法规则
- 例如： The little bear saw the fine fat trout in the brook. He saw the fine fat trout in the brook.

词序列的替代

从最上面一排开始，用单个的词（如：it）替换词序列（如：the brook）

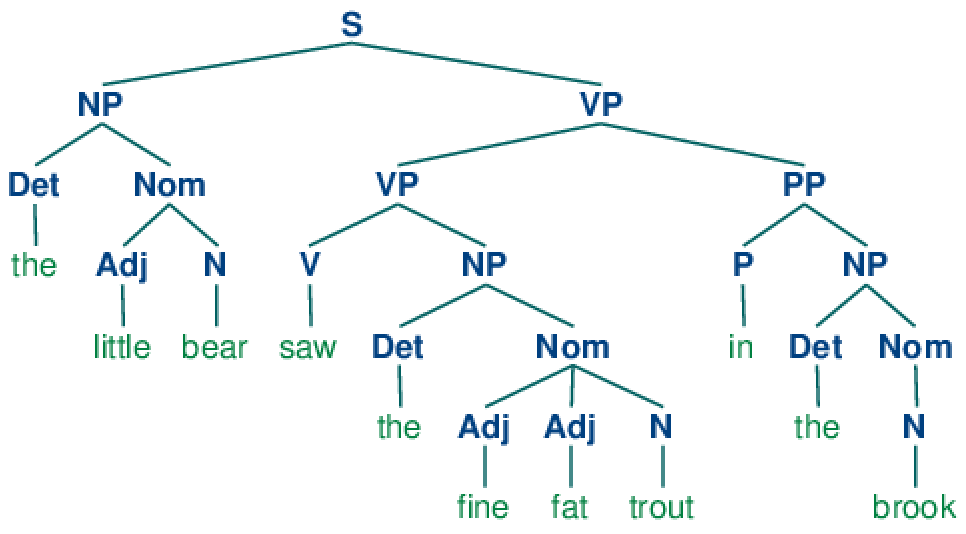
文法分类: 名词短语（NP），动词短语（VP），介词短语（PP）

Det the	Adj little	N bear	V saw	Det the	Adj fine	Adj fat	N trout	P in	Det the	N brook
Det the	Nom bear		V saw	Det the	Nom trout			P in	NP it	
NP He			V saw	NP it				PP there		
NP He			VP ran					PP there		
NP He			VP ran							

短语结构树

树的每个节点（包括词）被称为一个成分(constituent)

- 如S 的直接成分是NP 和VP



上下文无关文法

In [4]:

```
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"
  """)
```

In [5]:

```
sent = "Mary saw Bob".split()
rd_parser = nltk.RecursiveDescentParser(grammar1)
for tree in rd_parser.parse(sent):
    print(tree)
```

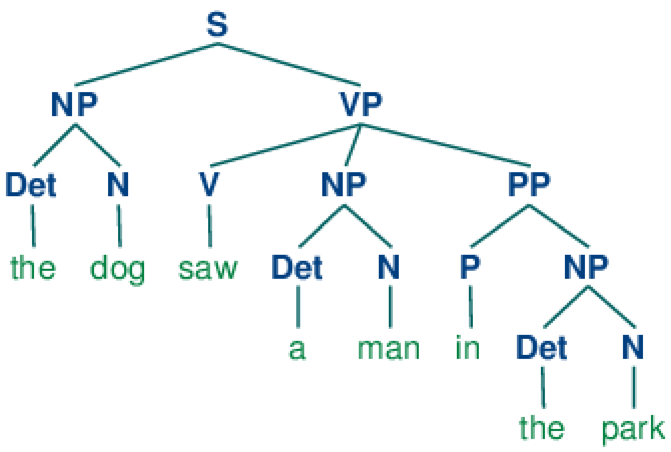
```
(S (NP Mary) (VP (V saw) (NP Bob)))
```

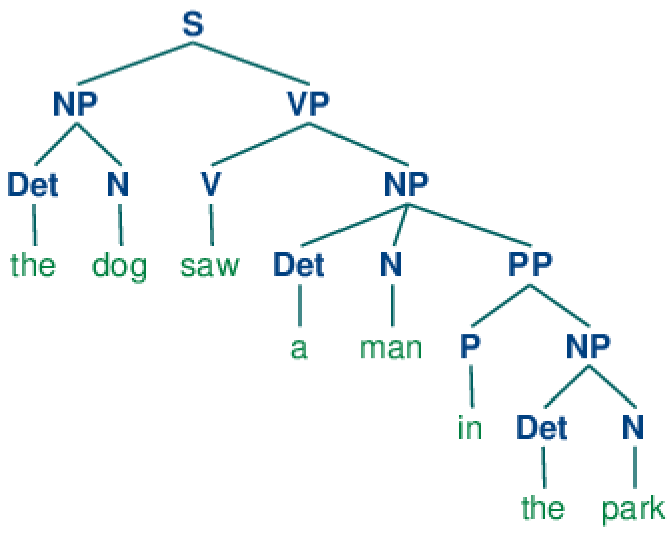
递归下降分析器演示

In [7]:

```
nltk.app.rdparser()
```

分析结果示例





句法结构中的递归

一个文法被认为是递归的，如果文法类型出现在产生式左侧也出现在右侧

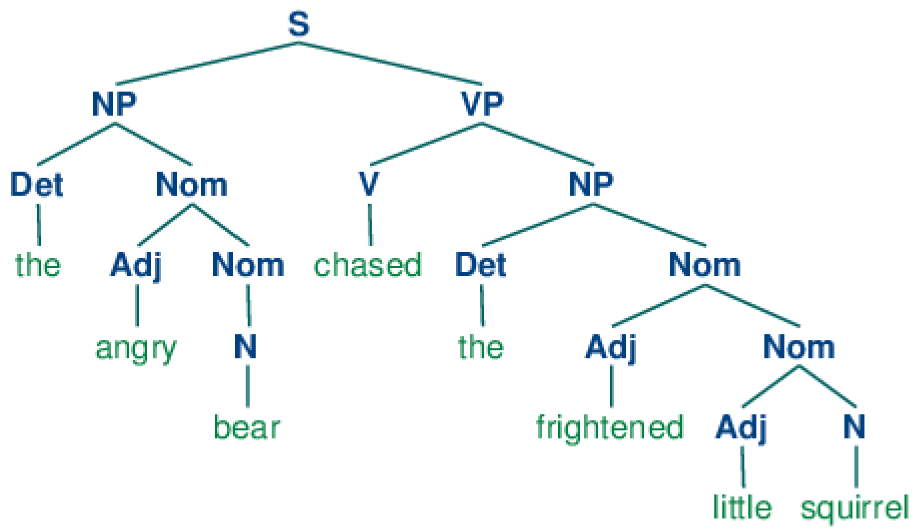
直接递归：Nom -> Adj Nom

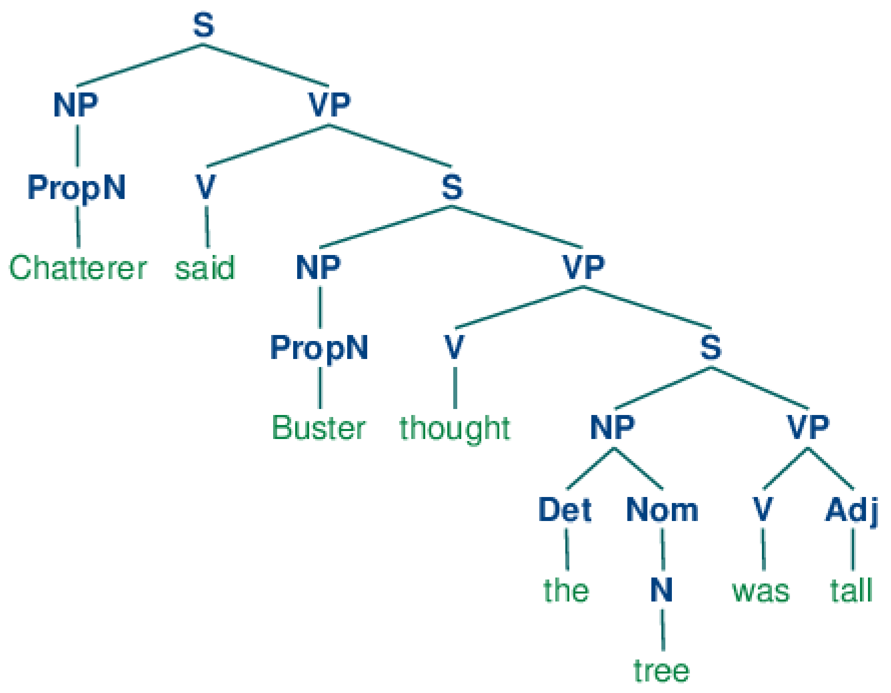
间接递归：S -> NP VP 与VP -> V S

In [6]:

```
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'
  """)
```

分析结果示例





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，并生成一个或多个符合文法的成分结构

文法是良构的声明规范，也是字符串

分析器是文法的解释程序

分析算法

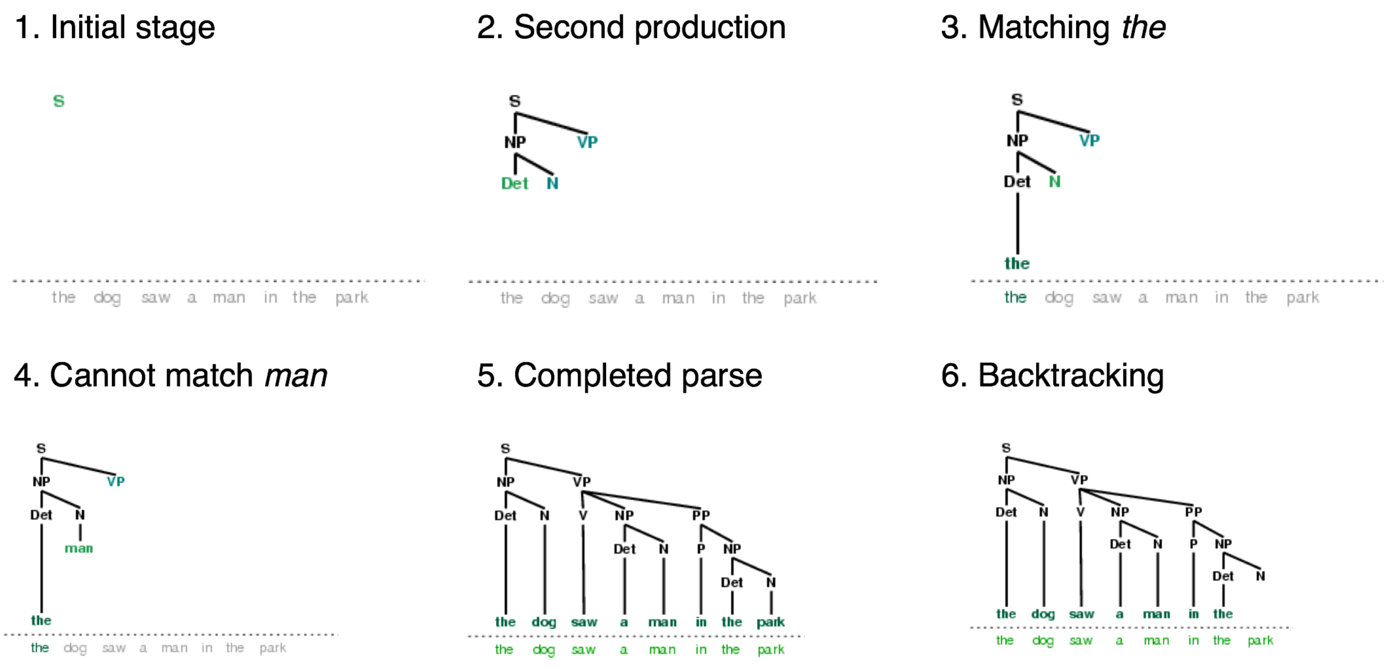
递归下降分析：自顶向下

递归下降分析器实现

缺点：1）左递归产生式，如：NP -> NP PP，会进入死循环

2）分析器浪费了很多时间处理不匹配输入句子的词和结构

3）回溯过程中会丢弃分析过的成分，它们在将来可能需要重建



递归下降分析器实现

In [7]:

```python
rd_parser = nltk.RecursiveDescentParser(grammar1,trace=2)
sent = 'Mary saw a dog'.split()
for tree in rd_parser.parse(sent):
    print(tree)
```

```
Parsing 'Mary saw a dog'
    [ * S ]
  E [ * NP VP ]
  E [ * 'John' VP ]
  E [ * 'Mary' VP ]
  M [ 'Mary' * VP ]
  E [ 'Mary' * V NP ]
  E [ 'Mary' * 'saw' NP ]
  M [ 'Mary' 'saw' * NP ]
  E [ 'Mary' 'saw' * 'John' ]
  E [ 'Mary' 'saw' * 'Mary' ]
  E [ 'Mary' 'saw' * 'Bob' ]
  E [ 'Mary' 'saw' * Det N ]
  E [ 'Mary' 'saw' * 'a' N ]
  M [ 'Mary' 'saw' 'a' * N ]
  E [ 'Mary' 'saw' 'a' * 'man' ]
  E [ 'Mary' 'saw' 'a' * 'dog' ]
  M [ 'Mary' 'saw' 'a' 'dog' ]
  + [ 'Mary' 'saw' 'a' 'dog' ]
(S (NP Mary) (VP (V saw) (NP (Det a) (N dog))))
  E [ 'Mary' 'saw' 'a' * 'cat' ]
  E [ 'Mary' 'saw' 'a' * 'telescope' ]
  E [ 'Mary' 'saw' 'a' * 'park' ]
  E [ 'Mary' 'saw' * 'an' N ]
  E [ 'Mary' 'saw' * 'the' N ]
  E [ 'Mary' 'saw' * 'my' N ]
  E [ 'Mary' 'saw' * Det N PP ]
  E [ 'Mary' 'saw' * 'a' N PP ]
  M [ 'Mary' 'saw' 'a' * N PP ]
  E [ 'Mary' 'saw' 'a' * 'man' PP ]
  E [ 'Mary' 'saw' 'a' * 'dog' PP ]
  M [ 'Mary' 'saw' 'a' 'dog' * PP ]
  E [ 'Mary' 'saw' 'a' 'dog' * P NP ]
  E [ 'Mary' 'saw' 'a' 'dog' * 'in' NP ]
  E [ 'Mary' 'saw' 'a' 'dog' * 'on' NP ]
  E [ 'Mary' 'saw' 'a' 'dog' * 'by' NP ]
  E [ 'Mary' 'saw' 'a' 'dog' * 'with' NP ]
  E [ 'Mary' 'saw' 'a' * 'cat' PP ]
  E [ 'Mary' 'saw' 'a' * 'telescope' PP ]
  E [ 'Mary' 'saw' 'a' * 'park' PP ]
  E [ 'Mary' 'saw' * 'an' N PP ]
  E [ 'Mary' 'saw' * 'the' N PP ]
  E [ 'Mary' 'saw' * 'my' N PP ]
  E [ 'Mary' * 'ate' NP ]
  E [ 'Mary' * 'walked' NP ]
  E [ 'Mary' * V NP PP ]
  E [ 'Mary' * 'saw' NP PP ]
  M [ 'Mary' 'saw' * NP PP ]
  E [ 'Mary' 'saw' * 'John' PP ]
  E [ 'Mary' 'saw' * 'Mary' PP ]
  E [ 'Mary' 'saw' * 'Bob' PP ]
  E [ 'Mary' 'saw' * Det N PP ]
  E [ 'Mary' 'saw' * 'a' N PP ]
  M [ 'Mary' 'saw' 'a' * N PP ]
```

```
E [ 'Mary' 'saw' 'a' * 'man' PP ]
E [ 'Mary' 'saw' 'a' * 'dog' PP ]
M [ 'Mary' 'saw' 'a' 'dog' * PP ]
E [ 'Mary' 'saw' 'a' 'dog' * P NP ]
E [ 'Mary' 'saw' 'a' 'dog' * 'in' NP ]
E [ 'Mary' 'saw' 'a' 'dog' * 'on' NP ]
E [ 'Mary' 'saw' 'a' 'dog' * 'by' NP ]
E [ 'Mary' 'saw' 'a' 'dog' * 'with' NP ]
E [ 'Mary' 'saw' 'a' * 'cat' PP ]
E [ 'Mary' 'saw' 'a' * 'telescope' PP ]
E [ 'Mary' 'saw' 'a' * 'park' PP ]
E [ 'Mary' 'saw' * 'an' N PP ]
E [ 'Mary' 'saw' * 'the' N PP ]
E [ 'Mary' 'saw' * 'my' N PP ]
E [ 'Mary' 'saw' * Det N PP PP ]
E [ 'Mary' 'saw' * 'a' N PP PP ]
M [ 'Mary' 'saw' 'a' * N PP PP ]
E [ 'Mary' 'saw' 'a' * 'man' PP PP ]
E [ 'Mary' 'saw' 'a' * 'dog' PP PP ]
M [ 'Mary' 'saw' 'a' 'dog' * PP PP ]
E [ 'Mary' 'saw' 'a' 'dog' * P NP PP ]
E [ 'Mary' 'saw' 'a' 'dog' * 'in' NP PP ]
E [ 'Mary' 'saw' 'a' 'dog' * 'on' NP PP ]
E [ 'Mary' 'saw' 'a' 'dog' * 'by' NP PP ]
E [ 'Mary' 'saw' 'a' 'dog' * 'with' NP PP ]
E [ 'Mary' 'saw' 'a' * 'cat' PP PP ]
E [ 'Mary' 'saw' 'a' * 'telescope' PP PP ]
E [ 'Mary' 'saw' 'a' * 'park' PP PP ]
E [ 'Mary' 'saw' * 'an' N PP PP ]
E [ 'Mary' 'saw' * 'the' N PP PP ]
E [ 'Mary' 'saw' * 'my' N PP PP ]
E [ 'Mary' * 'ate' NP PP ]
E [ 'Mary' * 'walked' NP PP ]
E [ * 'Bob' VP ]
E [ * Det N VP ]
E [ * 'a' N VP ]
E [ * 'an' N VP ]
E [ * 'the' N VP ]
E [ * 'my' N VP ]
E [ * Det N PP VP ]
E [ * 'a' N PP VP ]
E [ * 'an' N PP VP ]
E [ * 'the' N PP VP ]
E [ * 'my' N PP VP ]
```

递归下降分析器的缺点

- 左递归产生式，如：NP -> NP PP，会进入死循环
- 分析器浪费了很多时间处理不匹配输入句子的词和结构
- 回溯过程中会丢弃分析过的成分，它们在将来可能需要重建
  - 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

移进-归约分析

- 递归下降分析是一种自顶向下分析：在检查输入之前先使用文法预测输入将是什么！
- 移进-归约分析是一种自底向上分析：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

- 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，用左侧符号的替换它们，直到整个句子归约为一个S

移进-归约过程

1）移位操作(shift): 将下一个输入词移入堆栈

2）归约操作(reduce): 如果堆栈上的前n 项，匹配某个产生式的右侧的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

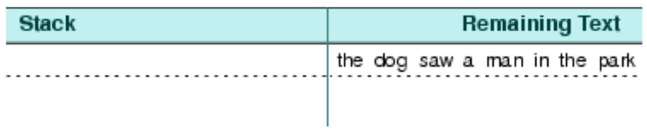
此操作只适用于堆栈的顶部

此操作必须在后面的项目被压入栈之前做

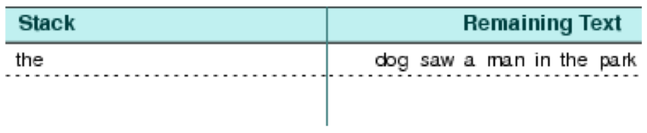
3）当所有的输入都使用过，堆栈中只剩余一个项目，也就是一棵分析树作为它的根的S 节点时，分析完成

移进-归约分析器的六个阶段

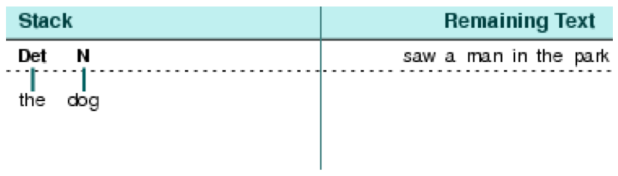
1. Initial state



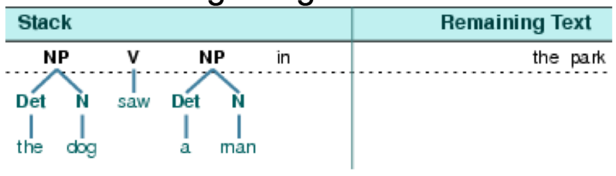
2. After one shift



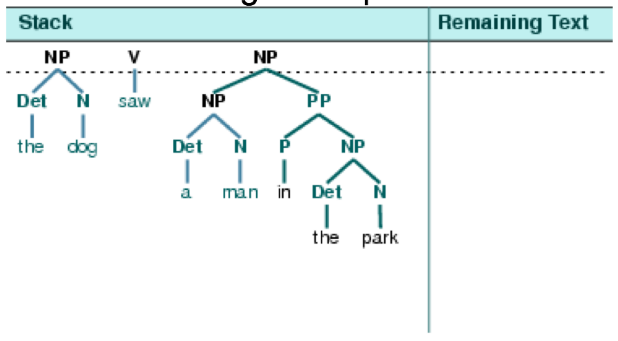
3. After reduce shift reduce



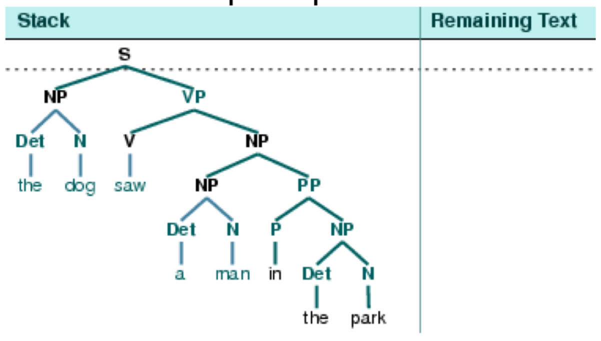
4. After recognizing the second NP



5. After building a complex NP



6. Built a complete parse tree



移进-归约分析器实现

In [8]:

```
sr_parser = nltk.ShiftReduceParser(grammar1,trace=2)
sent = 'Mary saw a dog'.split()
for tree in sr_parser.parse(sent):
    print(tree)
```

```
Parsing 'Mary saw a dog'
    [ * Mary saw a dog]
  S [ 'Mary' * saw a dog]
  R [ NP * saw a dog]
  S [ NP 'saw' * a dog]
  R [ NP V * a dog]
  S [ NP V 'a' * dog]
  R [ NP V Det * dog]
  S [ NP V Det 'dog' * ]
  R [ NP V Det N * ]
  R [ NP V NP * ]
  R [ NP VP * ]
  R [ S * ]
(S (NP Mary) (VP (V saw) (NP (Det a) (N dog))))
```

移进-归约分析器相比自顶向下分析器的优点：它们只建立与输入中的词对应的结构，且每个结构只建立一次

- 例如：NP(Det(the), N(man))只建立和压入栈一次，不管以后VP -> V NP PP 归约或者NP -> NP PP归约会不会用到

移进-归约分析器的冲突

- 当有多种归约可能时选择哪个归约
- 当移进和归约都可以时选择哪个动作
- 通过改进执行策略解决冲突
- 向前看：LR分析

In [9]:

```
nltk.app.srparser()
```

线图分析示例

In [9]:

```
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)
```

```
WFST 1    2    3    4    5    6    7
0    NP   .    .    .    .    .    .
1    .    V    .    .    .    .    .
2    .    .    Det  .    .    .    .
3    .    .    .    N    .    .    .
4    .    .    .    .    P    .    .
5    .    .    .    .    .    Det  .
6    .    .    .    .    .    .    N
```

In [10]:

```
wfst1 = complete_wfst(wfst0, tokens, groucho_grammar)
display(wfst1, tokens)
```

```
WFST 1    2    3    4    5    6    7
0    NP   .    .    S    .    .    S
1    .    V    .    VP   .    .    VP
2    .    .    Det  NP   .    .    .
3    .    .    .    N    .    .    .
4    .    .    .    .    P    .    PP
5    .    .    .    .    .    Det  NP
6    .    .    .    .    .    .    N
```

In [11]:

```
wfst1 = complete_wfst(wfst0, tokens, groucho_grammar, trace=True)
```

```
[2] Det [3]   N [4] ==> [2]  NP [4]
[5] Det [6]   N [7] ==> [5]  NP [7]
[1]   V [2]  NP [4] ==> [1]  VP [4]
[4]   P [5]  NP [7] ==> [4]  PP [7]
[0]  NP [1]  VP [4] ==> [0]   S [4]
[1]  VP [4]  PP [7] ==> [1]  VP [7]
[0]  NP [1]  VP [7] ==> [0]   S [7]
```

In [12]:

```
nltk.app.chartparser()
```

```
grammar= (
('    ', 'S -> NP VP,')
('    ', 'VP -> VP PP,')
('    ', 'VP -> V NP,')
('    ', 'VP -> V,')
('    ', 'NP -> Det N,')
('    ', 'NP -> NP PP,')
('    ', 'PP -> P NP,')
('    ', "NP -> 'John',")
('    ', "NP -> 'I',")
('    ', "Det -> 'the',")
('    ', "Det -> 'my',")
('    ', "Det -> 'a',")
('    ', "N -> 'dog',")
('    ', "N -> 'cookie',")
('    ', "N -> 'table',")
('    ', "N -> 'cake',")
('    ', "N -> 'fork',")
('    ', "V -> 'ate',")
('    ', "V -> 'saw',")
('    ', "P -> 'on',")
('    ', "P -> 'under',")
('    ', "P -> 'with',")
)
tokens = ['John', 'ate', 'the', 'cake', 'on', 'the', 'table']
Calling "ChartParserApp(grammar, tokens)"...
```

依存关系和依存文法

- 什么是依存关系和依存文法
- NLTK如何表示依存关系
- 如何确定中心词和从属词
- 动词与配价

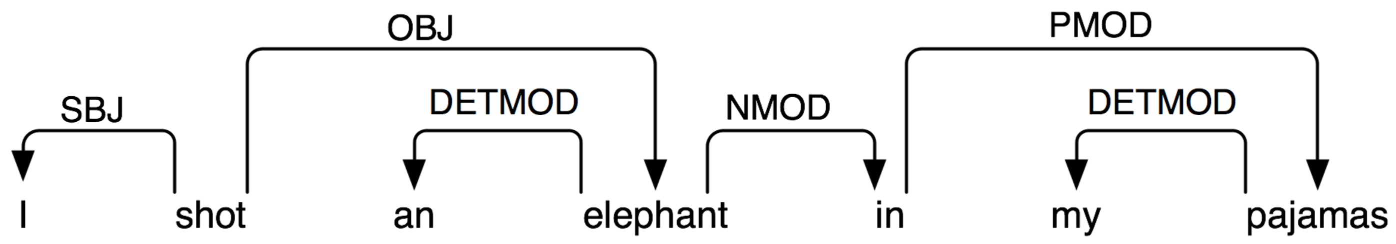
依存关系和依存文法

依存文法集中关注词与其他词之间的关系

依存关系是中心词与从属词之间的二元对称关系

- 句子的中心词通常是动词，所有其他词要么依赖于中心词，要么依赖路径与它连通

依存关系表示为带标签的有向图，其中节点是词语，弧表示依存关系，标签表示语法功能，箭头从中心词指向从属词



NLTK表示依存关系

只捕捉依存关系，不指定依存关系类型

投影式的依存关系：没有交叉边

In [13]:

```
groucho_dep_grammar = nltk.DependencyGrammar.fromstring("""
... 'shot' -> 'I' | 'elephant' | 'in'
... 'elephant' -> 'an' | 'in'
... 'in' -> 'pajamas'
... 'pajamas' -> 'my'
... """)
print(groucho_dep_grammar)
```

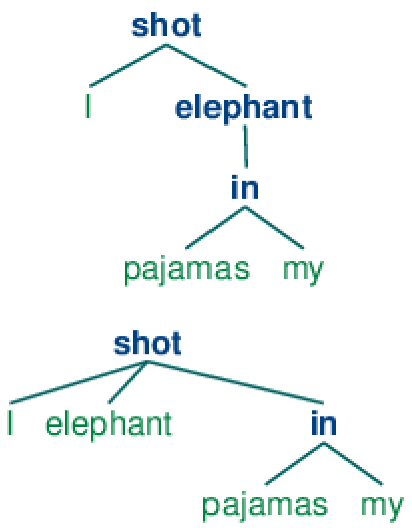
```
Dependency grammar with 7 productions
  'shot' -> 'I'
  'shot' -> 'elephant'
  'shot' -> 'in'
  'elephant' -> 'an'
  'elephant' -> 'in'
  'in' -> 'pajamas'
  'pajamas' -> 'my'
```

处理附着歧义

In [14]:

```
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)
```

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



如何确定中心词和从属词

- H 决定D的句法范畴；或者，D 的外部句法属性取决于H
- H 定义D 的语义范畴
- H 必须有而D 是可选的
- H 选择D 并且决定它是必须有的还是可选的
- D 的形态由H 决定（如agreement 或case government）

动词与配价

a. The squirrel was frightened.

b. Chatterer saw the bear.

c. Chatterer thought Buster was angry.

d. Joe put the fish on the log.

不符合语法规则的词序列

a. *The squirrel was Buster was angry.

b. *Chatterer saw frightened.

c. *Chatterer thought the bear.

d. *Joe put on the log.

文法开发

树库(TreeBank)与文法

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%样本

In [17]:

```
from nltk.corpus import treebank
t = treebank.parsed_sents('wsj_0001.mrg')[0]
print(t)
```

```
(S
  (NP-SBJ
    (NP (NNP Pierre) (NNP Vinken))
    (, ,)
    (ADJP (NP (CD 61) (NNS years)) (JJ old))
    (, ,))
  (VP
    (MD will)
    (VP
      (VB join)
      (NP (DT the) (NN board))
      (PP-CLR (IN as) (NP (DT a) (JJ nonexecutive) (NN director)))
      (NP-TMP (NNP Nov.) (CD 29))))
  (. .))
```

利用树库开发文法

使用过滤器找出带补语的动词

In [18]:

```
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)
```

In [19]:

```python
from nltk.corpus import treebank
[subtree for tree in treebank.parsed_sents()
 for subtree in tree.subtrees(filter)]
```

Out[19]:

```
[Tree('VP', [Tree('VBN', ['named']), Tree('S', [Tree('NP-SBJ', [Tree
('-NONE-', ['*-1'])]), Tree('NP-PRD', [Tree('NP', [Tree('DT', ['a']),
Tree('JJ', ['nonexecutive']), Tree('NN', ['director'])]), Tree('PP',
[Tree('IN', ['of']), Tree('NP', [Tree('DT', ['this']), Tree('JJ', ['B
ritish']), Tree('JJ', ['industrial']), Tree('NN', ['conglomerat
e'])])])])])]),
 Tree('VP', [Tree('VBD', ['said']), Tree(',', [',']), Tree('``', ['`
`']), Tree('S', [Tree('NP-SBJ', [Tree('DT', ['This'])]), Tree('VP',
[Tree('VBZ', ['is']), Tree('NP-PRD', [Tree('DT', ['an']), Tree('JJ',
['old']), Tree('NN', ['story'])])])])]),
 Tree('VP', [Tree('VBD', ['said']), Tree('S', [Tree('-NONE-', ['*T*-
1'])])]),
 Tree('VP', [Tree('VBN', ['expected']), Tree('S', [Tree('-NONE-',
['*?*'])])]),
 Tree('VP', [Tree('VBD', ['said']), Tree('S', [Tree('-NONE-', ['*T*-
1'])])]),
 Tree('VP', [Tree('VBZ', ['appears']), Tree('S', [Tree('NP-SBJ', [Tre
e('-NONE-', ['*-1'])]), Tree('VP', [Tree('TO', ['to']), Tree('VP', [T
```

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

In [20]:

```
from collections import defaultdict
entries = nltk.corpus.ppattach.attachments('training')
table = defaultdict(lambda: 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']))
```

```
1-to-4 N: ['added'] V: ['gained']
1-to-47 N: ['jumped'] V: ['added', 'rose']
1-to-point N: ['ended'] V: ['fell', 'rose']
3-to-17 N: ['lost'] V: ['lost']
500,000-in-fines N: ['paid'] V: ['paid']
6.9-on-scale N: ['registered'] V: ['registered']
access-to-AZT N: ['had'] V: ['had']
access-to-arena N: ['permits'] V: ['lack']
activity-in-part N: ['showed'] V: ['attributed']
agreement-in-principle N: ['reached'] V: ['reached']
agreement-with-Inc. N: ['announced', 'signed'] V: ['signed']
agreement-with-creditors N: ['reached'] V: ['nearing']
agreement-with-regulators N: ['presages', 'reach'] V: ['reach']
aid-to-Contras N: ['renewing'] V: ['renewing']
alliance-with-GM N: ['discussing', 'wrapping'] V: ['forge', 'have',
'negotiating']
approval-for-drug N: ['granted'] V: ['obtain']
attention-to-comments N: ['paid'] V: ['paid']
attention-to-concerns N: ['pay'] V: ['show']
attention-to-reports N: ['paid'] V: ['pay']
```

歧义的害处

In [21]:

```
grammar = nltk.CFG.fromstring("""
... S -> NP V NP
... NP -> NP Sbar
... Sbar -> NP V
... NP -> 'fish'
... V -> 'fish'
... """)
```

In [22]:

```
tokens = ["fish"] * 5
cp = nltk.ChartParser(grammar)
for tree in cp.parse(tokens):
    print(tree)
```

```
(S (NP fish) (V fish) (NP (NP fish) (Sbar (NP fish) (V fish))))
(S (NP (NP fish) (Sbar (NP fish) (V fish))) (V fish) (NP fish))
```

随着句子长度增加到（3，5，7，...），我们得到的分析树的数量是：1; 2; 5; 14; 42; 132; 429; 1,430; 4,862; 16,796; 58,786; 208,012; ….

加权文法

处理歧义是开发分析器的主要挑战，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

以动词give为例，它需要直接宾语和间接宾语，这些补语可以按任何顺序出现

example:宾州树库样本中give和gave的用法

In [23]:

```
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)
```

In [24]:

```
for tree in nltk.corpus.treebank.parsed_sents():
    for t in tree.subtrees(give):
        print_node(t, 72)
```

```
gave NP: the chefs / NP: a standing ovation
give NP: advertisers / NP: discounts for maintaining or increasing ad
sp...
give NP: it / PP-DTV: to the politicians
gave NP: them / NP: similar help
give NP: them / NP:
give NP: only French history questions / PP-DTV: to students in a Euro
pe...
give NP: federal judges / NP: a raise
give NP: consumers / NP: the straight scoop on the U.S. waste crisis
gave NP: Mitsui / NP: access to a high-tech medical product
give NP: Mitsubishi / NP: a window on the U.S. glass industry
give NP: much thought / PP-DTV: to the rates she was receiving , nor t
o ...
give NP: your Foster Savings Institution / NP: the gift of hope and fr
ee...
give NP: market operators / NP: the authority to suspend trading in fu
tu...
gave NP: quick approval / PP-DTV: to $ 3.18 billion in supplemental ap
pr...
give NP: the Transportation Department / NP: up to 50 days to review a
ny...
give NP: the president / NP: such power
give NP: me / NP: the heebie-jeebies
give NP: holders / NP: the right , but not the obligation , to buy a c
al...
gave NP: Mr. Thomas / NP: only a `` qualified '' rating , rather than
``...
give NP: the president / NP: line-item veto power
```

概率上下文无关文法

概率上下文无关文法是一种上下文无关文法，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分析树，并给每个分析树分配一个概率

所产生的分析树的概率仅仅是它用到的产生式的概率的乘积

In [25]:

```
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)
```

```
Grammar with 9 productions (start state = S)
    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]
```

In [26]:

```
viterbi_parser = nltk.ViterbiParser(grammar)
for tree in viterbi_parser.parse(['Jack', 'saw', 'telescopes']):
    print(tree)
```

```
(S (NP Jack) (VP (TV saw) (NP telescopes))) (p=0.064)
```

In [ ]:

```

```