

Тема 4 Анализ структуры предложения

Простая грамматика

Давайте начнем с рассмотрения простой безконтекстной грамматики. По соглашению левая сторона первого продуцирования является начальным символом грамматики, как правило, *S*, и все хорошо сформированные деревья должны иметь этот символ в качестве корневой метки. В NLTK безконтекстные грамматики определены в модуле `nltk.grammar`.

Грамматика в 3.1 содержит продуцирования с участием различных синтаксических категорий.

Обозначения	Значение	Пример
S	предложение	the man walked
NP	именная фраза	a dog
VP	глагольная фраза	saw a park
PP	предложная фраза	with a telescope
Det	определитель	the
N	существительное (имя)	dog
V	глагол	walked
P	предлог	in

Продуцирование, как $VP \rightarrow V\ NP \mid V\ NP\ PP$, имеет дизъюнкцию на правой стороне, показнную \mid , и такая запись является аббревиатурой для двух продуцирований $VP \rightarrow V\ NP$ и $VP \rightarrow V\ NP\ PP$.

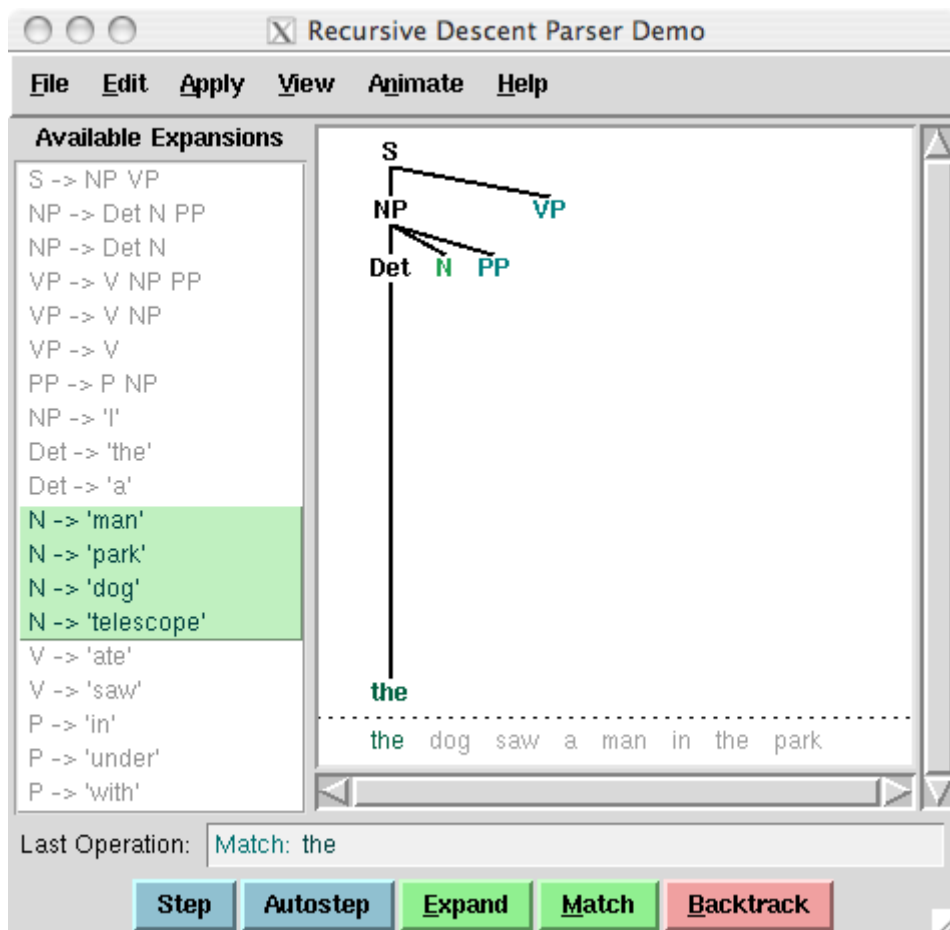


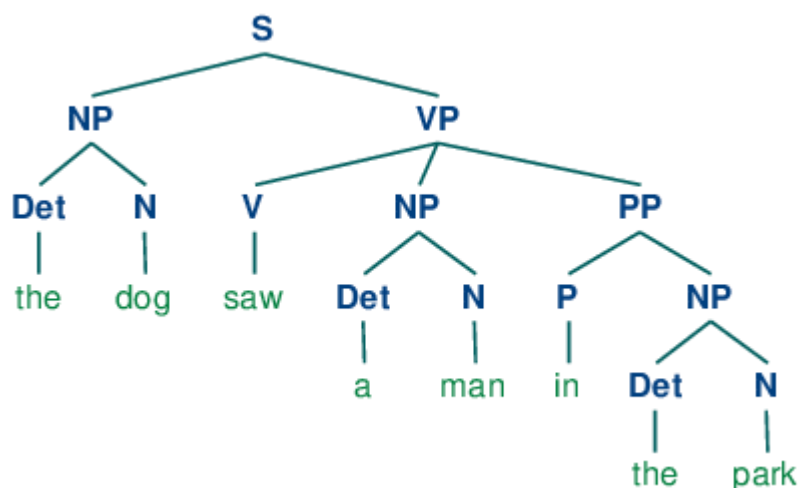
Рисунок 3.2: Recursive Descent Parser Demo: Этот инструмент позволяет следить за работой рекурсивно спускающегося синтаксического анализатора по мере того, как он создает дерево синтаксического разбора и наполняет его входными словами.

Замечание

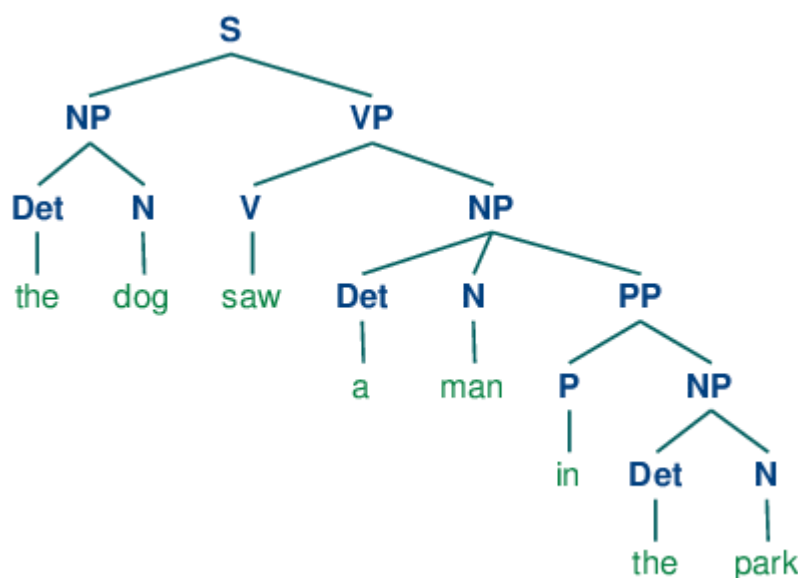
Ваша очередь: Попробуйте разработать простую грамматику самостоятельно с помощью приложения recursive descent parser, `nltk.app.rdparsers()`, показанного на Рисунке 3.2. Он поставляется с уже загруженным примером грамматики, но вы можете изменить его как вам будет угодно (с помощью меню `Edit`). Измените грамматику и предложение, которое должно быть проанализировано, и запустите анализатор с помощью кнопки *autostep*.

Если разобрать предложение *The dog saw a man in the park* с помощью грамматики, мы получим в конечном итоге два дерева похожих на те, которые мы видели для (3b):

(9) a.



b.



Так как наша грамматика допускает два дерева для этого предложения, предложение называется структурно неоднозначным. Данная неоднозначность называется неоднозначность приложения предложной фразы, как мы видели ранее в этой главе. Как вы, возможно, помните, это двусмысленность приложения, так как PP *in the park* необходимо приложить к одному из двух мест в дереве: либо как ребенок VP или как ребенок NP. Когда PP приложена к VP, предполагаемая интерпретация заключается в том, что событие видения произошло в парке. Однако если PP прилагается к NP, тогда в парке был человек, а агент видения (собака), возможно, сидела на балконе квартиры с видом на парк.

Написание ваших собственных грамматик

Если вы заинтересованы в экспериментировании с написанием безконтекстных грамматик (CFGs), вам будет удобно создавать и редактировать грамматики в текстовом файле, скажем, `mygrammar.cfg`. После этого вы можете загрузить ее в NLTK и осуществлять синтаксический разбор следующим образом:

```
>>> 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)
```

Убедитесь, что вы присоединили `.cfg` суффикс к имени файла и что в строке `'file:mygrammar.cfg'` нет пробелов. Если команда `print(tree)` не производит никакого вывода, это, вероятно, потому, что ваше предложение `sent` не допускается вашей грамматикой. В этом случае вызовите парсер с включенным отслеживанием: `rd_parser = NLTK.RecursiveDescentParser(grammar1, trace = 2)`. Вы также можете проверить, какие текущие продуцирования есть в грамматике с помощью команды `for p in grammar1.productions(): print(p)`.

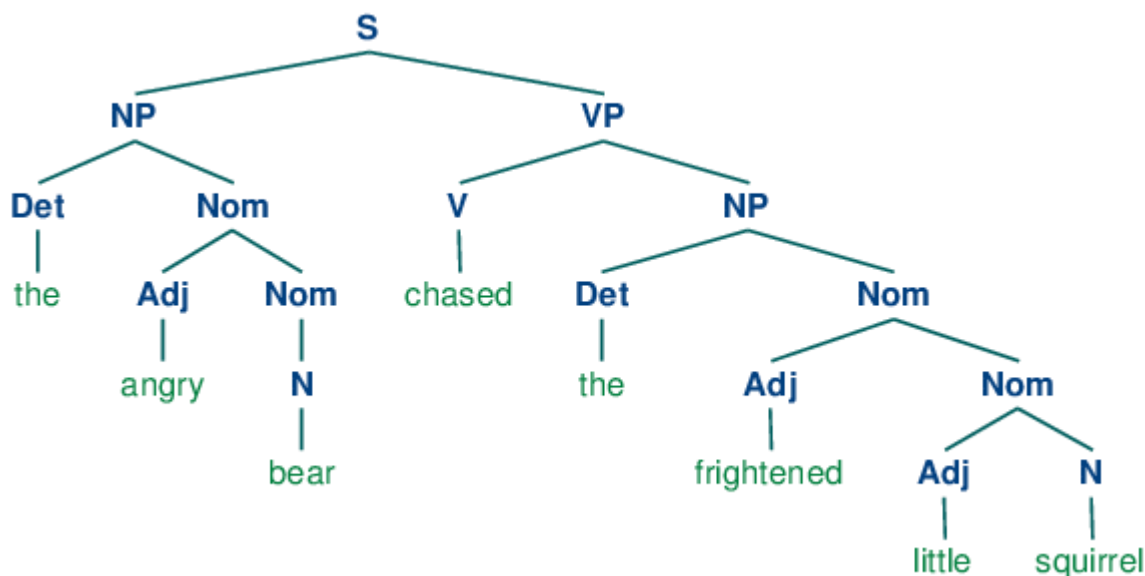
Когда вы пишете CFGs для разбора в NLTK, вы не можете комбинировать грамматические категории с лексическими единицами на правой стороне одного и того же продуцирования. Таким образом, продуцирование, такое как `PP -> 'of' NP` запрещено. Кроме того, не разрешается размещать лексические единицы, состоящие из нескольких слов, на правой части продуцирования. Поэтому вместо того, чтобы писать `NP -> 'New' 'York'`, вы должны прибегнуть к чему-то вроде `NP -> 'New_York'` вместо этого.

3.3 Рекурсия в синтаксической структуре

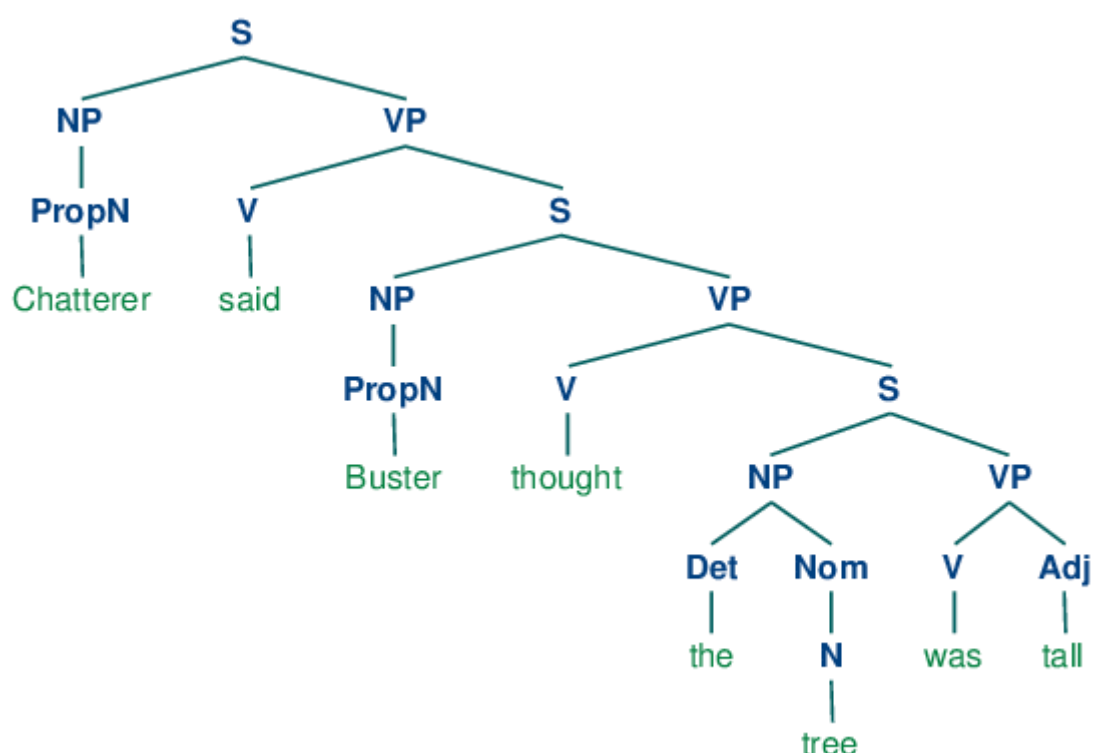
Грамматика называется рекурсивной, если категория, возникающая на левой стороне продуцирования, также появляется и на правой его части, как показано в 3.3. Продуцирование `Nom -> Adj Nom` (где `Nom` это категория номиналов) влечет прямую рекурсию по категории `Nom`, тогда как непрямая рекурсия по `S` возникает из сочетания двух продуцирований, а именно: `S -> NP VP` и `VP -> V S`.

Чтобы увидеть, как возникает рекурсия из этой грамматики, рассмотрим следующие деревья. (10a) влечет вложенные именные фразы, а (10b) содержит вложенные предложения.

(10)a.



b.



Мы проиллюстрировали только два уровня рекурсии здесь, но нет никакого верхнего предела глубины. Вы можете экспериментировать с разбором предложений, которые влекут более глубоко вложенные структуры. Обратите внимание, что `RecursiveDescentParser` не в состоянии справиться с левыми рекурсивными продуцированиями вида $X \rightarrow XY$; мы вернемся к этому в [4](#).

Синтаксический разбор с помощью безконтекстной грамматики

Анализатор обрабатывает входные предложения в соответствии с продуцированиями грамматики, а также создает одну или несколько составных структур, которые соответствуют грамматике. Грамматика является декларативной спецификацией хорошей оформленности - она, на самом деле, является только строкой, а не программой. Анализатор является процедурным толкованием грамматики. Он просматривает пространство деревьев, допускаемых грамматикой, чтобы найти то, которое может разместить требуемое предложение в своей кроне.

Анализатор позволяет оценить грамматику в отношении набора тестовых предложений, помогая лингвистам обнаружить ошибки в их грамматическом анализе. Анализатор может служить в качестве модели психолингвистической обработки, помогая объяснить трудности, которые люди испытывают при обработке некоторых синтаксических конструкций. Многие приложения естественного языка влекут синтаксический разбор на определенной стадии; например, можно было бы ожидать, что вопросы на естественном языке, задаваемые системе ответов на вопросы, пройдут синтаксический анализ в качестве первого шага обработки.

В этом разделе мы видим два простых алгоритма синтаксического анализа: метод, идущий сверху вниз, называемый рекурсивно спускающийся синтаксический анализ, и восходящий

метод, называемый синтаксическим анализом сокращения сдвига(?). Мы также видим некоторые более сложные алгоритмы: метод сверху вниз с фильтрацией снизу вверх под названием синтаксический анализ левого угла и метод динамического программирования, называемый синтаксическим анализом диаграмм.

Рекурсивно спускающийся синтаксический разбор

Простейший вид анализатора интерпретирует грамматику как спецификацию того, как разбить цель высокого уровня на несколько подзадач более низкого уровня. Цель верхнего уровня - найти S . Продукция $S \rightarrow NP VP$ позволяет анализатору заменить эту цель двумя подзадачами: найти NP , а затем найти VP . Каждая из этих подзадач может быть заменена в свою очередь подподцелями с помощью продукций, которые имеют на левой стороне NP и VP соответственно. В конце концов, этот процесс приводит к подзадачам, таким как: найти слово телескоп. Такие подзадачи можно сравнивать непосредственно с входной последовательностью, они являются успешными, если следующее слово найдено. Если совпадения нет анализатор должен вернуться наверх и попробовать другую альтернативу.

Во время этого процесса анализатору часто приходится выбирать между несколькими возможными продукциями. Например, при переходе от шага 3 к шагу 4, он пытается найти продукция с N на левой стороне. Первым из них является $N \rightarrow \text{человек}$. Когда это не срабатывает, он откатывается и пробует другие произведения N по порядку, пока не доберется до $N \rightarrow \text{собака}$, которое соответствует следующему слову во входном предложении. Намного позже, как показано на шаге 5, он находит полный разбор. Это дерево, которое охватывает все предложение без каких-либо оборванных краев. После того, как синтаксический разбор был найден, мы можем использовать анализатор для поиска дополнительных разборов. Опять же он будет возвращаться назад и исследовать другие варианты произведения на случай, если какой-нибудь из них приведет к разбору.

NLTK предоставляет метод рекурсивного спуска:

```
>>> rd_parser = nltk.RecursiveDescentParser(grammar1)
>>> sent = 'Mary saw a dog'.split()
>>> for tree in rd_parser.parse(sent):
...     print(tree)
(S (NP Mary) (VP (V saw) (NP (Det a) (N dog))))
```

Замечание

`RecursiveDescentParser()` принимает необязательный параметр `trace`. Если `trace` больше нуля, то анализатор будет сообщать те шаги, которые он предпринимает в процессе того, как он разбирает текст.

Синтаксический анализ с помощью рекурсивного спуска имеет три основных недостатка. Во-первых, левые рекурсивные произведения, как $NP \rightarrow NP PP$ отправляют его в бесконечный цикл. Во-вторых, анализатор тратит уйму времени, рассматривая слова и структуры, которые не соответствуют входному предложению. В-третьих, процесс отката может сбросить уже разобранные компоненты, которые необходимо будет восстановить позже. Например, откат с $VP \rightarrow V NP$ отбросит поддереву, созданное для NP . Если анализатор затем продолжает с $VP \rightarrow V NP PP$, то поддереву NP должно быть создано заново.

Синтаксический анализ с помощью рекурсивного спуска является видом синтаксического анализа сверху вниз. Нисходящие анализаторы используют грамматику, чтобы *предсказывать*, какой будет вход до проверки входа! Тем не менее, поскольку вход доступен для синтаксического анализатора все время, было бы более целесообразно рассмотреть входное предложение с самого начала. Такой подход называется синтаксический анализ снизу вверх, и мы увидим его пример в следующем разделе.

4.2 Синтаксический разбор помещения-сокращения

Простой вид синтаксического анализа снизу вверх является синтаксический анализатор помещения-сокращения. Как и все анализаторы снизу вверх, анализатор помещения-сокращения пытается найти последовательности слов и фраз, которые соответствуют *правой стороне* продуцирования грамматики, и заменить их левой стороной, пока вся фраза не свернется в S .

Анализатор неоднократно выталкивает следующее входное слово в стек; это операция называется помещением. Если верхние n элементов в стеке соответствуют n элементам на правой стороне какого-то продуцирования, то все они извлекаются из стека, а элемент на левой стороне продуцирования помещается в стек. Эта замена верхних n элементов одним элементом является операцией сокращения. Эта операция может быть применена только к верхней части стека; сокращение элементов, находящихся ниже в стеке, должно быть выполнено до помещения новых элементов в стек. Анализатор заканчивает работу, когда весь вход использован и остается только один элемент в стеке - дерево разбора с узлом S в качестве корневого элемента. Анализатор помещения-сокращения строит дерево разбора во время вышеописанного процесса. Каждый раз, когда она сокращает n элементов из стека, он объединяет их в частичное дерево разбора и помещает обратно в стек. Мы можем увидеть в действии алгоритм синтаксического анализа помещения-сокращения с помощью графической демонстрации `nltk.app.srparser()`.

NLTK предоставляет `ShiftReduceParser()`, простую реализацию анализатора помещения-сокращения. Этот анализатор не выполняет каких-либо откатов, так что нахождение разбора текста, даже если таковой существует, не гарантируется. Кроме того, он найдет не более одного разбора, даже если существует большее количество разборов. Мы можем предоставить дополнительный параметр `trace`, который управляет тем, как пространно анализатор сообщает о своих шагах, которые он предпринимает в процессе разбора текста:

```
>>> sr_parser = nltk.ShiftReduceParser(grammar1)
>>> sent = 'Mary saw a dog'.split()
>>> for tree in sr_parser.parse(sent):
...     print(tree)
(S (NP Mary) (VP (V saw) (NP (Det a) (N dog))))
```

Замечание

Ваша Очередь: Запустите вышеупомянутый анализатор в режиме трассировки, чтобы увидеть последовательность операций помещения и сокращения с помощью `sr_parse = NLTK.ShiftReduceParser(grammar1, trace = 2)`

Анализатор помещения-сокращения может зайти в тупик и не найти ни одного разбора, даже если входное предложение хорошо сформировано в соответствии с грамматикой. Когда это происходит, весь вход потреблен, а стек содержит элементы, которые не могут быть сокращены до S . Проблема возникает, потому что выбор, сделанный им ранее, не

может быть отменен (хотя пользователи графической демонстрации могут отменить свои решения). Есть два вида решений, которые будут приняты анализатором: (а) какое сокращение сделать, когда более чем одно возможно (б) следует ли поместить или сократить, когда оба действия возможны.

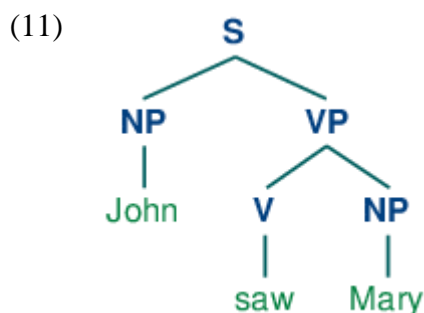
Анализатор помещения-сокращения может быть расширен для реализации политики разрешения подобных конфликтов. Например, такая политика может пытаться разрешить конфликты между помещением и сокращением, помещая только тогда, когда никакие сокращения невозможны, и она может пытаться решить конфликты между сокращениями выбором в пользу операции сокращения, которая удаляет самое большое число элементов из стека. (Обобщение анализатора помещения-сокращения, "смотрящий вперед LR-анализатор", обычно используется в компиляторах языков программирования.)

Преимущество анализаторов помещения-сокращения над рекурсивно спускающимися анализаторами заключается в том, что они строят только структуру, которая соответствует словам на входе. Кроме того, они строят каждую подструктуру только один раз, например $NP(Det(the), N(man))$ строится и помещается в стек только в один раз, независимо от того, будет ли он позже использоваться сокращением $VP \rightarrow V NP PP$ или сокращением $NP \rightarrow NP PP$.

Анализатор левого угла

Одна из проблем, связанных с рекурсивно спускающимся анализатором, заключается в том, что он уходит в бесконечный цикл, когда сталкивается с левосторонней рекурсией продуцирования. Это происходит потому, что он применяет грамматические продуцирования вслепую, без учета фактического входного предложения. Левоугольный анализатор представляет собой гибрид между подходами снизу вверх и сверху вниз, которые мы видели.

Грамматика `grammar1` позволяет нам производить следующий разбор `John saw Mary`:

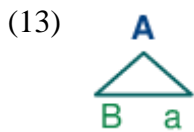


Напомним, что грамматика (определенная в [3.3](#)) имеет следующие продуцирования для расширения NP :

- (12)
- a. $NP \rightarrow Det N$
 - b. $NP \rightarrow Det N PP$
 - c. $NP \rightarrow "John" \mid "Mary" \mid "Bob"$

Предположим, мы просим вас сначала посмотреть на дерево (11), а затем решить, какую из NP подстановок (продуцирований) вы хотите, чтобы метод рекурсивного спуска применил первой - очевидно, (12с) является правильным выбором! Откуда вы знаете, что было бы бессмысленно применять сначала (12а) или (12б)? Потому что ни одна из этих двух

подстановок (продуцирований) не выведет последовательность, первое слово которой John. То есть, мы можем легко сказать, что в успешном разборе предложения John saw Mary анализатор должен расширить NP таким образом, чтобы NP давала последовательность John α . В более общем плане мы говорим, что категория В является левым углом дерева с корнем в А, если $A \Rightarrow *B \alpha$.



Левоугольный анализатор - это анализатор сверху вниз с фильтрацией снизу вверх. В отличие от обычного метода рекурсивного спуска, он не попадает в ловушку в левой рекурсии подстановки. Перед началом своей работы, левоугольный анализатор выполняет предварительную обработку контекстно свободной грамматики для построения таблицы, где каждая строка содержит две ячейки, первая содержит неконечную, а вторая содержит набор возможных левых углов неконечной (подстановки?). Таблица 4.1 иллюстрирует это для грамматики из grammar2.

Категория	Левые углы (предварительные конечные)
S	NP
NP	Det, PropN
VP	V
PP	P
Таблица 4.1:	
Левые углы в grammar2	

Каждый раз, когда подстановка рассматривается анализатором, он проверяет, что следующее входное слово совместимо, по меньшей мере, с одной из предварительных конечных категорий в левоугольной таблице.

Замечание

Чтобы помочь нам легко получать постановки по их правым частям, мы создаем индекс для грамматики. Это пример пространственно-временного компромисса: мы делаем обратный поиск по грамматике, вместо того, чтобы проверять весь список подстановок каждый раз, когда мы хотим найти по правой стороне.

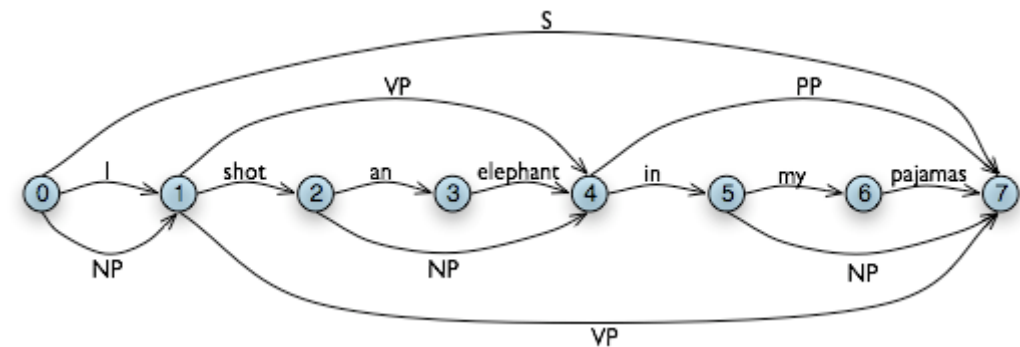


Рисунок 4.5: Графическая структура данных: неконечные представлены как дополнительные ребра на графике.

Обратите внимание на то, что мы не использовали каких-либо встроенных функций синтаксического анализа здесь. Мы реализовали полный примитивный графический анализатор с нуля!

WFST имеют несколько недостатков. Во-первых, как вы можете видеть, WFST не является сама по себе деревом разбора, поэтому метод, строго говоря, распознает, что предложение признается грамматикой, а не разбирает его. Во-вторых, она требует, чтобы каждая нелексическая подстановка грамматики была бинарной. Несмотря на то, что можно преобразовать произвольную CFG в эту форму, мы предпочли бы использовать подход без такого требования. В-третьих, как подход снизу вверх она потенциально расточительна, будучи в состоянии предложить конститuenty в тех местах, которые бы не были признаны грамматикой.

И, наконец, WFST не представляет структурную неоднозначность в предложении (то есть два прочтения глагольной фразы). VP в ячейке (1, 7) была фактически введена дважды, один раз для прочтения $V \quad NP$ и один раз для прочтения $VP \quad PP$. Это разные гипотезы, а вторая переписала первую (как оказалось, это не имело значения, так как левая часть была такой же.) Графические анализаторы используют немного более богатую структуру данных и некоторые интересные алгоритмы для решения этих проблем (обратитесь к разделу Дополнительные материалы в конце этой главы для получения подробностей).

Резюме

- Предложения имеют внутреннюю организацию, которая может быть представлена с помощью дерева. Основными свойствами конститuentной структуры являются: рекурсия, главы, дополнения и модификаторы.
- Грамматика представляет собой компактную характеристику потенциально бесконечного множества предложений; мы говорим, что дерево хорошо сформировано в соответствии с грамматикой, или что грамматика лицензирует (допускает) данное дерево.
- Грамматика - это формальная модель для описания того, может ли данной фразе быть присвоен отдельный конститuent или структура зависимости.
- Учитывая набор синтаксических категорий, безконтекстная грамматика использует набор подстановок, чтобы сказать, как фраза некоторой категории A может быть разобрана на последовательность меньших частей $\alpha_1 \dots \alpha_n$.
- Грамматика зависимости использует подстановки для спецификации зависимых данной лексической главы.
- Синтаксическая неоднозначность возникает, когда одно предложение имеет более одного синтаксического разбора (например, неоднозначность приложения предложной фразы).
- Анализатор представляет собой процедуру для нахождения одного или нескольких деревьев, соответствующих грамматически хорошо сформированным предложениям.
- Простой анализатор сверху вниз - рекурсивно спускающийся анализатор, который рекурсивно расширяет начальный символ (обычно s) с помощью подстановок грамматики и пытается найти соответствие входному предложению. Этот анализатор не может справиться с леворекурсивными подстановками (например, такими подстановками, как $NP \rightarrow NP \quad PP$). Он неэффективен в том, что слепо

расширяет категории без проверки, являются ли они совместимыми с входной строкой, и в том, что многократно расширяет одни и те же нетерминалы и в том, что отбрасывает результаты.

- Простой анализатор снизу вверх - анализатор помещения-смещения, который перемещает ввод в стек и пытается сопоставить элементы в верхней части стека с правой стороной грамматических подстановок. Этот анализатор не гарантирует нахождение правильного разбора для ввода, даже если таковой существует и строит подструктуру без проверки того, является ли она глобально совместимой с грамматикой.