

Тема 5 Анализ смысла предложений

Естественный язык, семантика и логика

Мы начали пытаться схватить значение (1a), переводя его в запрос на другом языке, SQL, который компьютер мог бы интерпретировать и выполнять. Но ответ на вопрос, был ли перевод правильным, так и не был дан. Отступив от запроса к базе данных, мы отметили, что смысл и, кажется, зависит от возможности указать, когда утверждения верны, а когда нет в той или иной ситуации. Вместо того, чтобы переводить предложение S с одного языка на другой, мы пытаемся сказать, о чем S , связывая его с ситуацией в мире. Давайте продолжим дальше. Представьте себе ситуацию s , в которой есть два объекта, Margrietje и ее любимая кукла, Brunoke. Кроме того, существует связь между двумя этими объектами, которую мы будем называть отношением *любви*. Если вы понимаете значение (3), то вы знаете, что оно истинно в ситуации s . В частности, вы это знаете, потому что вы знаете, что Margrietje относится к Margrietje, Brunoke относится к Brunoke, а houdt van относится

После того, как мы приняли понятие истины в некоторой ситуации, у нас есть мощный инструмент для рассуждений. В частности, мы можем рассмотреть наборы предложений и спросить, могут ли они быть истинными вместе в какой-то ситуации.

- (5) a. Sylvania is to the north of Freedonia.
 b. Freedonia is a republic.
- (6) a. The capital of Freedonia has a population of 9,000.
 b. No city in Freedonia has a population of 9,000.
- 7. a. Sylvania is to the north of Freedonia.
 b. Freedonia is to the north of Sylvania.

Мы выбрали предложения о вымышленных странах (из фильма 1933 года *Утиный суп* братьев Маркс), чтобы подчеркнуть, что ваша способность рассуждать об этих примерах не зависит от того, что является истинным или ложным в реальном мире. Если вы знаете значение слова по, а также знаете, что столица страны - это город в этой стране, то вы должны быть в состоянии сделать вывод о том, что два предложения в (6) несовместимы независимо от того, где находится Фридония или какова численность населения ее столицы. То есть, нет никакой возможности ситуации, в которой оба предложения могли бы быть правдой. Точно так же, если вы знаете, что отношение к северу от асимметрично, то вы должны быть в состоянии сделать вывод о том, что два предложения в (7) несовместимы.

Вообще говоря, подходы, основанные на логике, к семантике естественного языка сосредоточены на тех аспектах естественного языка, которые направляют наши суждения о совместимости и несовместимости. Синтаксис логического языка разработан так, чтобы сделать эти свойства формально выраженными. В результате определение свойств совместимости часто может быть сведено к символической манипуляции, то есть к задаче, которая может быть выполнена компьютером. Для реализации этого подхода мы сначала хотим разработать методику для представления возможной ситуации. Мы делаем это с помощью того, что логики называют моделью.

Модель для множества предложений W является формальным представлением ситуации, в которой все предложения в W являются истинными. Обычный способ представления моделей использует теорию множеств. Домен D дискурса (все объекты, которые нас интересуют в настоящее время) представляет собой совокупность индивидуальных объектов, а отношения рассматриваются как наборы, построенные из D . Давайте

посмотрим на конкретном примере. Наш домен D будет состоять из трех детей: Stefan, Klaus и Evi, представленных соответственно как s , k и e . Запишем это следующим образом: $D = \{s, k, e\}$. Выражение `boy` обозначает множество, состоящее из Stefan и Klaus, выражение `girls` обозначает множество, состоящее из Evi, а выражение `is running` обозначает множество, состоящее из Stefan и Evi. Рисунок [1.2](#) представляет собой графическое отображение модели.

Позже в этой главе мы будем использовать модели, чтобы оценить истинность или ложность английских предложений, и таким образом проиллюстрировать некоторые методы для представления значения. Однако прежде, чем остановиться на этом более подробно, давайте поместим обсуждение в более широкую перспективу и вернемся обратно к вопросу, который мы затронули в [5](#). Может ли компьютер понимать смысл предложения? И как мы могли бы сказать, что он это сделал? Это похоже на вопрос: "Может ли компьютер мыслить?" Алан Тьюринг предложил ответить на этот вопрос путем изучения способности компьютера вести имеющие смысл беседы с человеком (Turing, 1950). Предположим, что вы ведете две беседы в чате с человеком и компьютером, но вам не сказали, кто из них кто. Если после беседы с каждым из них вы не можете определить, кто из ваших партнеров компьютер, то компьютер успешно имитировал человека. Если компьютер успешно выдал себя за человека в этой "имитационной игре" (широко известной как "тест Тьюринга"), то согласно Тьюрингу мы должны быть готовы сказать, что компьютер может думать и может быть назван разумным. Так что Тьюринг обошел вопрос какого-либо исследования внутренних состояний компьютера, используя вместо этого его поведение как свидетельство интеллекта. По той же причине мы предположили, что для того, чтобы сказать, что компьютер понимает по-английски, он просто должен вести себя так, как будто бы это было так. Что важно здесь - так это не столько специфика имитационной игры Тьюринга, сколько предложение судить о способности понимать естественный язык по наблюдаемому поведению.

Логика высказываний

Логический язык разработан, чтобы сделать рассуждения формально явными. В результате он может схватывать аспекты естественного языка, которые определяют, является ли множество предложений совместимым. В рамках этого подхода мы должны разработать логические представления предложения ϕ , которые формально будут схватывать условия истинности ϕ . Начнем с простого примера:

(8) [Klaus chased Evi] and [Evi ran away].

Давайте заменим два простых предложения в (8) на ϕ [фи] и ψ [пси] соответственно и поставим $\&$ [амперсанд] для логического оператора, соответствующего английскому слову `and`: $\phi \& \psi$. Эта структура является логической формой [\(8\)](#).

Логика высказываний позволяет представить только те части языковой структуры, которые соответствуют конкретным связкам предложений. Мы только что рассмотрели `and`. Другими подобными связками являются `not`, `or` и `if ..., then` В формализации логики высказываний аналоги таких связей иногда называют логическими операторами. Базовые выражения логики высказываний суть символы высказываний, часто записываемые как P , Q , R и т.п. Существуют различные соглашения для представления логических операторов. Так как мы будем сосредоточены на способах изучения логики с помощью NLTK, мы будем придерживаться следующих ASCII-версий операторов:

```
>>> nltk.boolean_ops()
negation          -
conjunction        &
disjunction        |
implication        ->
equivalence        <->
```

Из символов высказываний и логических операторов мы можем построить бесконечное множество хорошо сформированных формул (или просто формул для краткости) логики высказываний. Во-первых, каждый символ высказываний является формулой. Тогда если ϕ является формулой, то и $\neg\phi$ является формулой. А если ϕ и ψ являются формулами, то и формулами являются: $(\phi \ \& \ \psi)$ $(\phi \ | \ \psi)$ $(\phi \rightarrow \psi)$ $(\phi \leftrightarrow \psi)$.

Таблица 2.1 характеризует условия истинности для формул, содержащих эти операторы. Как и прежде, мы используем ϕ и ψ в качестве переменных для предложений и сокращаем тогда и только тогда как iff.

Логический оператор	Условия истины		
Отрицание (это не тот случай, когда...)	$\neg\phi$ истинно в s	iff	ϕ ложно в S
конъюнкция (и)	$(\phi \ \& \ \psi)$ истинно в s	iff	ϕ истинно в s и ψ истинно в s
дизъюнкция (или)	$(\phi \ \ \psi)$ истинно в s	iff	ϕ истинно в s или ψ истинно в s
импликация (если ..., то ...)	$(\phi \rightarrow \psi)$ истинно в s	iff	ϕ ложно в s или ψ истинно в s
эквивалентность (тогда и только тогда)	$(\phi \leftrightarrow \psi)$ истинно в s	iff	ϕ и ψ оба являются истинными в s или ложными в s
Таблица 2.1: Условия истинности для логических операторов в логике высказываний.			

Эти правила, в общем, просты, хотя условия истинности для импликации исходят во многих случаях из наших обычных представлений об условных предложениях в английском языке. Формула вида $(P \rightarrow Q)$ ложна только тогда, когда P истинна, а Q ложно. Если P ложно (скажем, P соответствует Луна сделана из зеленого сыра) и Q истинно (скажем Q соответствует два плюс два равно четыре), то $P \rightarrow Q$ оказывается истинной.

Объект NLTK `Expression` может перерабатывать логические выражения в различные подклассы `Expression`:

```
>>> read_expr = nltk.sem.Expression.fromstring
>>> read_expr('-(P & Q)')
<NegatedExpression -(P & Q)>
>>> read_expr('P & Q')
<AndExpression (P & Q)>
>>> read_expr('P | (R -> Q)')
```

```
<OrExpression (P | (R -> Q))>
>>> read_expr('P <-> -- P')
<IffExpression (P <-> --P)>
```

С вычислительной точки зрения логики дают нам важный инструмент для умозаключений. Предположим, вы утверждаете, что Freedonia is not to the north of Sylvania, и вы указываете в качестве аргументов, что Sylvania is to the north of Freedonia. В этом случае, вы произвели аргумент. Предложение Sylvania is to the north of Freedonia является посылкой аргумента, в то время как Freedonia is not to the north of Sylvania является его выводом. Этап перехода от одной или нескольких посылок к выводу называется умозаключением. Неформально обычно записывают аргументы в формате, в котором выводу предшествует "следовательно" (therefore).

- (9) Sylvania is to the north of Freedonia.
Therefore, Freedonia is not to the north of Sylvania

Аргумент является действительным, если нет возможной ситуации, в которой все его предпосылки истинны, а его вывод неистинный.

Теперь справедливость (9) в решающей степени зависит от смысла фразы to the north of, в частности, от того факта, что она выражает асимметричное отношение:

- (10) if x to the north of y, тогда y is not to the north of x.

К сожалению, мы не можем выразить такие правила в логике высказываний: самые маленькие элементы, с которыми мы можем работать - атомарные высказывания, и мы не можем "заглянуть внутрь" них, чтобы сказать об отношениях между индивидуальными объектами x и y. Лучшее, что мы можем сделать в этом случае, - схватить частный случай асимметрии. Давайте использовать символ высказывания S_nF для Sylvania is to the north of Freedonia и F_nS для Freedonia is to the north of Sylvania. Чтобы выразить, что Freedonia is not to the north of Sylvania, мы напомним $\neg F_nS$. То есть, мы рассматриваем not как эквивалент фразы это не тот случай, когда... и переводим это как одноместный логический оператор -. Так что теперь мы можем записать импликацию в (10) как

- (11) $S_nF \rightarrow \neg F_nS$

Как насчет того, чтобы дать версию полного аргумента? Мы заменим первое предложение (9) на две формулы логики высказываний: S_nF и импликацию в (11), которая выражает (весьма слабо) наше фоновое знание о смысле выражения to the north of. Мы пишем $[A_1, \dots, A_n] / C$, чтобы представить аргумент, вывод которого C следует из посылок $[A_1, \dots, A_n]$. Это приводит к следующему выражению как представлению аргумента (9):

- (12) $[S_nF, S_nF \rightarrow \neg F_nS] / \neg F_nS$

Это действительный аргумент: если S_nF и $S_nF \rightarrow \neg F_nS$ оба являются истинными в ситуации s , то $\neg F_nS$ должно также быть истинно в s . Напротив, если F_nS было бы истиной, то это противоречило бы нашему пониманию того, что два объекта не могут быть оба к северу друг от друга в любой возможной ситуации. Эквивалентно, список $[S_nF, S_nF \rightarrow \neg F_nS, F_nS]$ несовместим - эти предложения не могут быть истинными вместе.

Аргументы могут быть проверены на "синтаксическую действительность" с помощью системы доказательств. Мы скажем об этом немного больше в 3. Логические

доказательства могут быть выполнены с помощью модуля NLTK *inference*, например, *через* интерфейс к стороннему пакету для доказательства теорем Prover9. Входы для механизма логического вывода сначала должны быть преобразованы в логические выражения.

```
>>> lp = nltk.sem.Expression.fromstring
>>> SnF = read_expr('SnF')
>>> NotFnS = read_expr('-FnS')
>>> R = read_expr('SnF -> -FnS')
>>> prover = nltk.Prover9()
>>> prover.prove(NotFnS, [SnF, R])
True
```

Вот еще один способ увидеть, почему следует этот вывод. Выражение $SnF \rightarrow -FnS$ семантически эквивалентно выражению $\neg SnF \vee -FnS$, где \vee является двухместным оператором, соответствующим или. В общем, $\phi \vee \psi$ истинно в ситуации s , если или ϕ истинно в s , или ψ истинно в s . Теперь предположим, что оба SnF и $\neg SnF \vee -FnS$ истинны в ситуации s . Если SnF истинно, то $\neg SnF$ не может быть также истинным; фундаментальное предположение классической логики заключается в том, что предложение не может быть одновременно истинным и ложным в одной ситуации. Следовательно, $-FnS$ должно быть истиной.

Напомним, что мы интерпретируем предложения логического языка, относящегося к модели, которая является очень упрощенной версией мира. Модель для логики высказываний должна присвоить значения Истина или Ложь для каждой возможной формулы. Мы делаем это индуктивно: сначала каждому символ высказываний присваивается значение, а затем мы вычисляем значение сложных формул, обращаясь к значениям логических операторов (т.е. [2.1](#)) и применяя их к значениям компонентов этих формул. Valuation - это соединение исходных выражений логики с их значениями (оценка). Вот пример:

```
>>> val = nltk.Valuation([('P', True), ('Q', True), ('R', False)])
```

Мы создаем Valuation с помощью списка пар, каждая из которых состоит из семантического символа и семантического значения. Полученный объект является по существу всего лишь словарем, который соединяет логические выражения (обрабатываемые как строки) и соответствующие значения.

```
>>> val['P']
True
```

Как мы увидим позже, наши модели должны быть несколько более сложными для того, чтобы обрабатывать более сложные логические формы, обсуждаемые в следующем разделе; до поры до времени просто игнорируйте параметры dom и g в следующих декларациях.

```
>>> dom = set()
>>> g = nltk.Assignment(dom)
```

Теперь давайте инициализируем модель m, которая использует val:

```
>>> m = nltk.Model(dom, val)
```

Каждый объект `Model` имеет метод `evaluate()`, который будет определять смысловое значение логических выражений, таких как формулы логики высказываний; конечно, эти значения зависят от начальных значений истинности, которые мы назначили символам высказываний, таким как `P`, `Q` и `R`.

```
>>> print(m.evaluate('(P & Q)', g))
True
>>> print(m.evaluate('-(P & Q)', g))
False
>>> print(m.evaluate('(P & R)', g))
False
>>> print(m.evaluate('(P | R)', g))
True
```

Замечание

Ваша очередь: Поэкспериментируйте с оценкой различных формул логики высказываний. Дает ли модель оценки, которые вы ожидали увидеть?

До сих пор мы переводили наши английские предложения на язык логики высказываний. Поскольку мы ограничены представлением атомарных предложений с помощью символов, таких как `P` и `Q`, мы не можем копаться в их внутренней структуре. Фактически мы говорим, что нет ничего, представляющего интерес с точки зрения логики в делении атомарных предложений на субъекты, объекты и предикаты. Однако это кажется ошибочным: если мы хотим формализовать аргументы, такие как [\(9\)](#), мы должны иметь возможность "заглянуть внутрь" исходных предложений. В результате мы выйдем за пределы логики высказываний к чему-то более выразительному, а именно логике первого порядка. Это то, к чему мы переходим в следующем разделе.

3 Логика первого порядка

В оставшейся части этой главы мы будем представлять смысл выражений естественного языка, переводя их на язык логики первого порядка. Не все из семантики естественного языка может быть выражено в логике первого порядка. Но это хороший выбор для вычислительной семантики, поскольку она достаточно выразительна, чтобы представлять собой хорошее решение, а с другой стороны, существуют отличные готовые системы для автоматизированных умозаключений в логике первого порядка.

Наш следующий шаг будет заключаться в том, чтобы описать, как формулы логики первого порядка строятся, а затем, как такие формулы могут быть оценены в модели.

3.1 Синтаксис

Логика первого порядка сохраняет все логические операторы логики высказываний. Но она добавляет некоторые важные новые механизмы. Начнем с того, что высказывания расчленяются на предикаты и аргументы, что делает нас на шаг ближе к структуре естественных языков. Стандартные правила построения для логики первого порядка признают такие термины, как отдельные переменные и константы, а также предикаты, которые принимают разное число аргументов. Например, `Angus walks` может быть

оформлена как `walk(angus)`, а `Angus sees Bertie` как `see(angus, bertie)`. Мы будем называть `walk` унарным предикатом, а `see` бинарным предикатом. Символы, используемые как предикаты не имеют внутреннего смысла, хотя это трудно запомнить. Возвращаясь к одному из наших предыдущих примеров, нет никакого логического различия между (13a) и (13b).

- (13) a. `love(margrietje, brunoke)`
 b. `houden_van(margrietje, brunoke)`

Сама по себе логика первого порядка не имеет ничего существенного сказать о лексической семантике - смысле отдельных слов - хотя некоторые теории лексической семантики могут быть закодированы в логику первого порядка. Является ли атомарная предикация, как `see(angus, bertie)`, истинной или ложной в некоторой ситуации, не вопрос логики, а зависит от конкретной оценки, которую мы выбрали для констант `see`, `angus` и `bertie`. По этой причине такие выражения называются нелогическими константами. В противоположность этому логические константы (такие как логические операторы) всегда получают одну ту же интерпретацию в каждой модели для логики первого порядка.

Следует отметить здесь, что один двоичный предикат имеет особый статус, а именно равенство, как в формулах, таких как `angus = aj`. Равенство рассматривается как логическая константа, так как для индивидуальных терминов `t1` и `t2`, формула `t1 = t2` истинна тогда и только тогда, когда `t1` и `t2` относятся к одному и тому же объекту.

Часто бывает полезно проверить синтаксическую структуру выражений логики первого порядка, обычный способ сделать это - назначить типы для выражений. Следуя традиции грамматики Montague, мы будем использовать два основных типа: `e` это тип объектов, а `t` - тип формул, то есть выражений, которые имеют значения истинности. С помощью этих двух основных типов, мы можем сформировать сложные типы для функциональных выражений. То есть для любых типов σ [сигма] и τ [тау] $\langle\sigma, \tau\rangle$ представляет собой сложный тип, соответствующий функциям из " σ вещей" в " τ вещи". Например, $\langle e, t \rangle$ является типом выражений из объектов в значения истинности, то есть одноместные предикаты. Логическое выражение может быть обработано с помощью проверки типа.

```
>>> read_expr = nltk.sem.Expression.fromstring
>>> expr = read_expr('walk(angus)', type_check=True)
>>> expr.argument
<ConstantExpression angus>
>>> expr.argument.type
e
>>> expr.function
<ConstantExpression walk>
>>> expr.function.type
<e, ?>
```

Почему мы видим $\langle e, ? \rangle$ в конце этого примера? Несмотря на то, что программа проверки типа попытается вывести как можно больше типов, в данном случае ей не удалось полностью определить тип `walk`, так как тип его результата неизвестен. Хотя мы предполагаем, что `walk` получит тип $\langle e, t \rangle$, в соответствии с тем, что программа проверки типа знает, в этом контексте оно может быть каким-либо другим типом, таким как $\langle e, e \rangle$ или $\langle e, \langle e, t \rangle \rangle$. Чтобы помочь программе проверки типа, необходимо указать подпись в виде словаря, который явно связывает типы с нелогическими константами:


```
>>> sig = {'walk': '<e, t>'}
>>> expr = read_expr('walk(angus)', signature=sig)
>>> expr.function.type
e
```

Бинарный предикат имеет тип $\langle e, \langle e, t \rangle \rangle$. Несмотря на то, что это тип чего-то, что сочетается в первую очередь с аргументом типа e , чтобы образовать унарный предикат, мы представляем бинарные предикаты как сочетающиеся непосредственно с их двумя аргументами. Например, предикат *see* в переводе *Angus sees Cyril* будет сочетаться со своими аргументами, чтобы дать результат *see(angus, cyril)*.

В логике первого порядка аргументы предикатов также могут быть индивидуальными переменными, такими как x , y и z . В NLTK мы принимаем соглашение, что переменные типа e - все в нижнем регистре. Индивидуальные переменные похожи на личные местоимения, как он, она и оно, в том, что мы должны знать о контексте использования, чтобы выяснить их денотат.

Один из способов интерпретации местоимения в (14) является указание на соответствующий индивидуальный объект в локальном контексте.

(14) He disappeared.

Другой способ заключается в том, чтобы предоставить текстового предшественника для местоимения он, например, произнеся (15a) перед (14). Здесь мы говорим, что *he* соотносится с именной фразой *Cyril*. В результате (14) семантически эквивалентно (15b).

- (15) a. Cyril is Angus's dog.
 b. Cyril disappeared.

Сравните с *he* в (16a). В этом случае оно связано с неопределенной NP *a dog*, и это отношение отличное от соотнесенности. Если заменить местоимение *he* на *a dog*, результат (16b) не будет семантически эквивалентным (16a).

- (16) a. Angus had a dog but he disappeared.
 b. Angus had a dog but a dog disappeared.

В соответствии с (17a) мы можем построить открытую формулу (17b) с двумя вхождениями переменной x . (Мы игнорируем время для упрощения изложения.)

- (17) a. He is a dog and he disappeared.
 b. $\text{dog}(x) \wedge \text{disappear}(x)$

Размещая квантификатор существования $\exists x$ ('для некоторого x ') перед (17b), мы можем связать эти переменные, как в (18a), что означает (18b) или, более идиоматически, (18c).

- (18) a. $\exists x.(\text{dog}(x) \wedge \text{disappear}(x))$
 b. At least one entity is a dog and disappeared.
 c. A dog disappeared.

NLTK представление (18a):

(19) `exists x.(dog(x) & disappear(x))`

В дополнение к квантификатору существования логика первого порядка дает нам универсальный квантификатор $\forall x$ ('для всех x '), проиллюстрированный в (20).

- (20) a. $\forall x.(\text{dog}(x) \rightarrow \text{disappear}(x))$
 b. Everything has the property that if it is a dog, it disappears.
 c. Every dog disappeared.

Синтаксис NLTK для (20a):

(21) `all x. (dog(x) -> disappear(x))`

Хотя (20a) является стандартным переводом (20c) на язык логики первого порядка, условия истинности необязательно такие, как вы ожидаете. Формула говорит, что если какое-то x - собака, то x исчезает, но она не говорит о том, что есть какие-то собаки. Таким образом, в ситуации, когда ни у кого нет собак, (20a) все равно будет истинным. (Помните, что $(P \rightarrow Q)$ истинно, когда P ложно.) Сейчас вы могли бы возразить, что `every dog disappeared` на самом деле предполагают наличие собак, и что логическая формализация просто ошибочна. Но можно найти и другие примеры, в которых отсутствует такое предположение. Например, мы могли бы объяснить, что значение выражения `Python astring.replace('ate', '8')` является результатом замены каждого вхождения 'ate' в `astring` на '8', хотя на самом деле в ней может не быть таких вхождений (3.2).

Мы видели множество примеров, когда переменные связаны квантификаторами. Что происходит в формулах, таких как эта?

`((exists x. dog(x)) -> bark(x))`

Сферой квантификатора `exists x` является `dog(x)`, поэтому появление x в `bark(x)` не ограничено. Следовательно, оно может стать связанным каким-либо другим квантификатором, например, `all x` в следующей формуле:

`all x. ((exists x. dog(x)) -> bark(x))`

Вообще вхождение переменной x в формуле ϕ является свободным в ϕ , если это вхождение не попадает в сферу действия `all x` или `some x` в ϕ . И, наоборот, если x свободен в формуле ϕ , тогда связан в `all x. ϕ` и `exists x. ϕ` . Если все вхождения переменных в формуле связаны, то формула называется закрытой.

Мы уже упоминали ранее, что объект `Expression` может обрабатывать строки и возвращать объекты класса `Expression`. Каждый экземпляр `expr` этого класса имеет метод `free()`, который возвращает набор переменных, которые свободны в `expr`.

```
>>> read_expr = nltk.sem.Expression.fromstring
>>> read_expr('dog(cyril)').free()
set()
>>> read_expr('dog(x)').free()
{Variable('x')}
>>> read_expr('own(angus, cyril)').free()
set()
>>> read_expr('exists x.dog(x)').free()
```

```

set()
>>> read_expr('((some x. walk(x)) -> sing(x)).free()
{Variable('x')}
>>> read_expr('exists x.own(y, x)).free()
{Variable('y')}

```

3.2 Доказательство теорем логики первого порядка

Вспомните ограничение на *to the north of which*, которое мы предложили ранее как (10):

(22) if *x* is to the north of *y* then *y* is not to the north of *x*.

Мы заметили, что логика высказываний не достаточно выразительна для представления обобщений о бинарных предикатах и в результате мы не правильно схватываем аргумент *Sylvania is to the north of Freedonia. Therefore, Freedonia is not to the north of Sylvania.*

Вы, без сомнения, уже догадались, что логика первого порядка, напротив, идеально подходит для формализации таких правил:

```

all x. all y.(north_of(x, y) -> -north_of(y, x))

```

Более того, мы можем сделать автоматический вывод, чтобы показать обоснованность аргумента.

Общим случаем в доказательстве теорем является определение того, может ли формула, которую мы хотим доказать (цель доказательства) быть получена с помощью конечной последовательности выводов из списка формул-предпосылок. Мы записываем это как $S \vdash g$, где S - (возможно, пустой) список предпосылок, а g - цель доказательства.

Проиллюстрируем это с помощью интерфейса NLTK к системе доказывания теорем Prover9. Сначала мы разбираем цель доказательства ❶ и две предпосылки ❷❸. Затем мы создаем экземпляр Prover9 ❹ и вызываем его метод `prove()` в отношении цели доказательства для данного списка предпосылок ❺.

```

>>> NotFnS = read_expr('-north_of(f, s)')
>>> SnF = read_expr('north_of(s, f)')
>>> R = read_expr('all x. all y. (north_of(x, y) -> -north_of(y, x))')
>>> prover = nltk.Prover9()
>>> prover.prove(NotFnS, [SnF, R])
True

```

К счастью, Prover9 соглашается с нами, что аргумент является действительным. И наоборот заключает, что не представляется возможным сделать вывод о `north_of(f, s)` из наших предпосылок:

```

>>> FnS = read_expr('north_of(f, s)')
>>> prover.prove(FnS, [SnF, R])
False

```

3.3 Резюме языка логики первого порядка

Мы воспользуемся этой возможностью, чтобы вновь сформулировать упомянутые нами ранее синтаксические правила для логики высказываний и добавить правила формирования для квантификаторов; вместе, они дают нам синтаксис логики первого порядка. Кроме того, мы сделаем явными типы участвующих выражений. Мы примем соглашение, что $\langle e^n, t \rangle$ является типом предиката, который сочетается с n аргументами типа e , чтобы получить выражение типа t . В этом случае мы говорим, что n является арностью (размерностью) предиката.

1. If P is a predicate of type $\langle e^n, t \rangle$, and $\alpha_1, \dots, \alpha_n$ are terms of type e , then $P(\alpha_1, \dots, \alpha_n)$ is of type t .
2. If α and β are both of type e , then $(\alpha = \beta)$ and $(\alpha \neq \beta)$ are of type t .
3. If ϕ is of type t , then so is $\neg \phi$.
4. If ϕ and ψ are of type t , then so are $(\phi \ \& \ \psi)$, $(\phi \mid \psi)$, $(\phi \rightarrow \psi)$ and $(\phi \leftrightarrow \psi)$.
5. If ϕ is of type t , and x is a variable of type e , then $\exists x.\phi$ and $\forall x.\phi$ are of type t .

Таблица 3.1 обобщает новые логические константы модуля `logic` и два метода `ExpressionS`.

Пример	Описание
<code>=</code>	равенство
<code>!=</code>	неравенство
<code>exists</code>	квантификатор существования
<code>all</code>	универсальный квантификатор
<code>e.free()</code>	показать свободные переменные e
<code>e.simplify()</code>	произвести β -сокращение на e
Таблица 3.1: Резюме новых логических отношений и операторов необходимых для логики первого порядка, наряду с двумя полезными методами класса <code>Expression</code> .	

3.4 Истинность в модели

Мы рассмотрели синтаксис логики первого порядка, а в 4 мы исследуем задачу перевода с английского языка на язык логики первого порядка. Тем не менее, как мы заявили в 1, это продвигает нас вперед, только если мы можем придать смысл предложениям логики первого порядка. Другими словами, мы должны придать семантику условной истинности логике первого порядка. С точки зрения вычислительной семантики, существуют очевидные ограничения того, как далеко можно продвинуть этот подход. Несмотря на то, что мы хотим говорить о предложениях, которые являются истинными или ложными в ситуациях, у нас есть для представления ситуаций в компьютере только средства в символической форме. Несмотря на это ограничение, все же возможно получить более

ясную картину семантики условной истинности посредством кодирования модели в NLTK.

Для данного языка логики первого порядка L модель M для L представляет собой пару $\langle D, Val \rangle$, где D представляет собой непустое множество, которое называется область (домен) модели, а Val - функция, которая называется функция оценки, которая присваивает значения из D выражениям L следующим образом:

1. Для каждой индивидуальной константы c в L , $Val(c)$ является элементом D .
2. Для каждого символа предиката P размерности $n \geq 0$, $Val(P)$ является функцией из D^n в $\{True, False\}$. (Если размерность P равно 0, то $Val(P)$ просто истинное значение, P рассматривается как символ высказывания.)

Согласно (ii), если P имеет размерность 2, тогда $Val(P)$ будет функцией f из пар элементов D в $\{True, False\}$. В моделях, которые мы будем строить в NLTK, мы будем принимать более удобную альтернативу, в которой $Val(P)$ представляет собой набор S пар, определенных следующим образом:

$$(23) \quad S = \{S \mid f(s) = True\}$$

Такая f называется характеристической функцией S (в соответствии с обсуждением в дополнительных материалах).

Отношения представлены семантически в NLTK стандартным способом теории множеств: как множества кортежей. Например, давайте предположим, что мы имеем область дискурса, состоящую из индивидуальных объектов Bertie, Olive и Cyril, где Bertie мальчик, Olive девочка, а Cyril собака. По мнемоническим причинам мы используем b , o и c в качестве соответствующих ярлыков в модели. Мы можем объявить область следующим образом:

```
>>> dom = {'b', 'o', 'c'}
```

Мы будем использовать утилитарную функцию `Valuation.fromstring()`, чтобы преобразовать список строк формы символ \Rightarrow значение в объект `Valuation`.

```
>>> v = """
... bertie => b
... olive => o
... cyril => c
... boy => {b}
... girl => {o}
... dog => {c}
... walk => {o, c}
... see => {(b, o), (c, b), (o, c)}
... """
>>> val = nltk.Valuation.fromstring(v)
>>> print(val)
{'bertie': 'b',
 'boy': (('b',)),
 'cyril': 'c',
 'dog': (('c',))}
```

```
'girl': {('o',)},
'olive': 'o',
'see': {('o', 'c'), ('c', 'b'), ('b', 'o')},
'walk': {('c',), ('o',)}
```

Таким образом, согласно этой оценке значение `see` - это множество кортежей, таких что Bertie sees ОливOlive, Cyril sees Bertie и Olive see Cyril.

Замечание

Ваша очередь: Нарисуй рисунок области m и множеств, соответствующих каждому из одноместных предикатов, по аналогии со схемой на Рисунке [1.2](#).

Вы возможно заметили, что наши унарные предикаты (т.е., `boy`, `girl`, `dog`) также оказываются наборами одиночных кортежей, а не просто наборами индивидуальных объектов. Это сделано для удобства и позволяет нам иметь однообразную обработку отношений любой размерности. Предикация вида $P(\tau_1, \dots, \tau_n)$, где P имеет размерность n , оказывается верной только в случае, если кортеж значений, соответствующих (τ_1, \dots, τ_n) принадлежит множеству кортежей в значении P .

```
>>> ('o', 'c') in val['see']
True
>>> ('b',) in val['boy']
True
```

3.5 Индивидуальные переменные и присвоения

В наших моделях соответствием контекста употребления является присвоение переменной. Это соединение индивидуальных переменных с объектами в области модели дискурса. Присвоения создаются с помощью конструктора `Assignment`, который также принимает область модели дискурса в качестве параметра. Мы не обязаны фактически вводить все привязки, но если мы это делаем, они даются в формате (переменная, значение) аналогичном тому, который мы видели ранее для оценок.

```
>>> g = nltk.Assignment(dom, [('x', 'o'), ('y', 'c')])
>>> g
{'y': 'c', 'x': 'o'}
```

Кроме того, есть формат `print()` для назначений, которые используют обозначение более близкое к тому, что часто встречается в учебниках логики:

```
>>> print(g)
g[c/y][o/x]
```

Давайте теперь посмотрим на то, как мы можем оценить атомарную формулу логики первого порядка. Сначала мы создаем модель, затем мы вызываем метод `evaluate()` для вычисления значения истинности.

```
>>> m = nltk.Model(dom, val)
>>> m.evaluate('see(olive, y)', g)
True
```

Что тут происходит? Мы оцениваем формулу, которая похожа на наш более ранний пример, `see(olive, cyril)`. Однако когда функция интерпретации встречает переменную `y`, вместо того, чтобы проверять ее значение в `val`, она запрашивает присвоение переменных `g` для получения значения:

```
>>> g['y']  
'c'
```

Так как мы уже знаем, что индивидуальные объекты `o` и `c` стоят в отношении `see`, значение Истина - это то, что мы ожидали. В этом случае, мы можем сказать, что присвоение `g` удовлетворяет формуле `see(olive, y)`. В противоположность этому следующая формула оценивается как Ложь относительно `g` - проверьте, что вы видите, почему это так.

```
>>> m.evaluate('see(y, x)', g)  
False
```

В нашем подходе (хотя и не в стандартной логике первого порядка) присвоения переменных являются частичными. Например, `g` ничего не говорит о каких-либо переменных, кроме `x` и `y`. Метод `purge()` очищает все связи присвоения.

```
>>> g.purge()  
>>> g  
{}
```

Если мы теперь попытаемся оценить формулу, такую как `see(olive, y)` относительно `g`, это все равно что пытаться интерпретировать предложение, содержащее `him`, когда мы не знаем, к чему оно (`him`) относится. В этом случае функция оценки не в состоянии установить значение истинности.

```
>>> m.evaluate('see(olive, y)', g)  
'Undefined'
```

Так как наши модели уже содержат правила интерпретации логических операторов, сколь угодно сложные формулы могут быть составлены и оценены.

```
>>> m.evaluate('see(bertie, olive) & boy(bertie) & -walk(bertie)', g)  
True
```

Общий процесс определения истинности или ложности формулы в модели называется проверкой модели.

3.6 Квантификация

Одним из важнейших прозрений современной логики является то, что понятие удовлетворения переменной можно использовать для интерпретации квантифицированных формул. Давайте используем [\(24\)](#) в качестве примера.

```
(24) exists x.(girl(x) & walk(x))
```

Когда это истинно? Давайте думать обо всех индивидуальных объектах в нашем домене, то есть в dom . Мы хотим проверить, обладает ли какой-либо из этих объектов свойством `girl` и `walking`. Другими словами, мы хотим знать, если есть некоторые u в dom , такие что $g[u/x]$ удовлетворяет открытой формуле (25).

(25) $\text{girl}(x) \ \& \ \text{walk}(x)$

Рассмотрите следующее:

```
>>> m.evaluate('exists x.(girl(x) & walk(x))', g)
True
```

`evaluate()` возвращает Истину здесь, потому что есть некоторые u в dom , такие что (25) удовлетворяется присвоением, которое связывает x с u . На самом деле, o является такой u :

```
>>> m.evaluate('girl(x) & walk(x)', g.add('x', 'o'))
True
```

Одним из полезных инструментов, предлагаемых NLTK, является метод `satisfiers()`. Он возвращает набор всех индивидуальных объектов, которые удовлетворяют открытой формуле. Параметрами метода являются разобранная формула, переменная и присвоение. Вот несколько примеров:

```
>>> fmla1 = read_expr('girl(x) | boy(x)')
>>> m.satisfiers(fmla1, 'x', g)
{'b', 'o'}
>>> fmla2 = read_expr('girl(x) -> walk(x)')
>>> m.satisfiers(fmla2, 'x', g)
{'c', 'b', 'o'}
>>> fmla3 = read_expr('walk(x) -> girl(x)')
>>> m.satisfiers(fmla3, 'x', g)
{'b', 'o'}
```

полезно подумать о том, почему `fmla2` и `fmla3` получить значения, которые они получают. Условия истинности для \rightarrow означают, что `fmla2` эквивалентно $\neg \text{girl}(x) \mid \text{walk}(x)$, что удовлетворяется чем-то, что либо не `girl`, либо не `walks`. Поскольку ни `b` (Bertie), ни `c` (Cyril) не `girl` в соответствии с моделью m , они оба удовлетворяют всей формуле. И, конечно, o удовлетворяет формуле, так как o удовлетворяет обоим членам дизъюнкции. Теперь, так как все члены области дискурса удовлетворяют `fmla2`, соответствующая универсально квантифицированная формула также истинна.

```
>>> m.evaluate('all x.(girl(x) -> walk(x))', g)
True
```

Другими словами, универсально квантифицированная формула $\forall x. \phi$ истинна в отношении g , только в случае если для любого u , ϕ истинно в отношении $g[u/x]$.

Замечание

Ваша очередь: Попробуйте выяснить, сначала с карандашом и бумагой, а затем с помощью `m.evaluate()`, значения истинности для `all x.(girl(x) & walk(x))` и `exists`

$x. (boy(x) \rightarrow walk(x))$. Убедитесь, что вы понимаете, почему они получают эти значения.

3.7 Неоднозначность сферы действия квантификатора

Что происходит, когда мы хотим дать формальное представление предложения с *двумя* квантификаторами, такого как следующее?

(26) Everybody admires someone.

Есть (по крайней мере) два способа выражения (26) в логике первого порядка:

- (27) a. $\text{all } x. (\text{person}(x) \rightarrow \text{exists } y. (\text{person}(y) \ \& \ \text{admire}(x, y)))$
 b. $\text{exists } y. (\text{person}(y) \ \& \ \text{all } x. (\text{person}(x) \rightarrow \text{admire}(x, y)))$

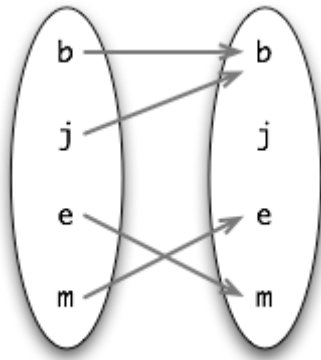
Можем ли мы использовать оба этих варианта? Ответ да, но они имеют разные значения. (27b) логически сильнее, чем (27a): он утверждает, что существует единственный человек, скажем, Брюс, который восхищает всех. С другой стороны, (27a) просто требует, чтобы для каждого человека x можно было найти человека y , которым x восхищается; но это может быть каждый раз другой человек y . Мы различаем (27a) и (27b) по сфере действия квантификаторов. В первом варианте \forall имеет более широкий охват, чем \exists , тогда как в (27b), соотношение сфер обратное. Так что теперь у нас есть два способа представления значения (26), и они оба вполне правомерны. Другими словами, мы утверждаем, что (26) *неоднозначно* относительно охвата квантификаторов, а формулы в (27) дают нам возможность сделать два прочтения явными. Тем не менее, мы заинтересованы не просто в связывании двух различных представлений с (26). Мы также хотим показать в деталях, как эти два представления приводят к различным условиям истинности в модели.

Для того, чтобы более внимательно изучить неоднозначность, давайте исправим нашу оценку следующим образом:

```
>>> v2 = ""
... bruce => b
... elspeth => e
... julia => j
... matthew => m
... person => {b, e, j, m}
... admire => {(j, b), (b, b), (m, e), (e, m)}
... ""
>>> val2 = nltk.Valuation.fromstring(v2)
```

Отношение `admire` можно визуализировать с помощью схемы отображения, показанной на рисунке (28).

(28)



На рисунке (28) стрелка между двумя индивидуальными объектами x и y означает, что x восхищается y . Таким образом, j и b оба восхищаются b (Брюс очень показной), а e восхищается m и m восхищается e . В этой модели формула (27a) истинна, а (27b) ложна. Одним из способов изучения этих результатов является метод `satisfiers()` объектов `Model`.

```
>>> dom2 = val2.domain
>>> m2 = nltk.Model(dom2, val2)
>>> g2 = nltk.Assignment(dom2)
>>> fmla4 = read_expr('(person(x) -> exists y.(person(y) & admire(x,
y)))')
>>> m2.satisfiers(fmla4, 'x', g2)
{'e', 'b', 'm', 'j'}
```

Это показывает, что `fmla4` содержит для каждого индивидуального объекта в домене. В противоположность этому, рассмотрим формулу `fmla5` ниже; она не имеет удовлетворяющих для переменной y .

```
>>> fmla5 = read_expr('(person(y) & all x.(person(x) -> admire(x, y)))')
>>> m2.satisfiers(fmla5, 'y', g2)
set()
```

То есть, нет ни одного человека, которым восхищаются все. Принимая другую открытую формулу, `fmla6`, мы можем убедиться в том, что есть человек, а именно Брюс, который вызывает восхищение как Юлии, так и Брюса.

```
>>> fmla6 = read_expr('(person(y) & all x.((x = bruce | x = julia) ->
admire(x, y)))')
>>> m2.satisfiers(fmla6, 'y', g2)
{'b'}
```

Замечание

Ваша очередь: Придумайте новую модель, основанную на `m2`, такую что (27a) оказывается ложным в вашей модели; аналогичным образом придумайте новую модель, такую что (27b) оказывается истинным.

3.8 Построение модели

До сих пор мы предполагали, что у нас уже есть модель, и хотели проверить истинность предложения в модели. В противоположность этому построение модели пытается создать новую модель для данного набора предложений. Если это удастся, тогда мы знаем, что множество совместимо, так как мы имеем доказательство существования модели.

Мы вызываем построитель моделей Mace4 путем создания экземпляра класса `Mace()` и вызова метода `build_model()` аналогично вызову доказателя теорем Prover9. Одним из вариантов является обработка нашего набора предложений в качестве посылок, оставив цель не определенной. Следующее взаимодействие показывает, почему `[a, c1]` и `[a, c2]` являются совместимыми списками, так как Mace4 удастся построить модель для каждого из них, в то время как `[c1, c2]` несовместим.

```
>>> a3 = read_expr('exists x. (man(x) & walks(x))')
>>> c1 = read_expr('mortal(socrates)')
>>> c2 = read_expr('-mortal(socrates)')
>>> mb = nltk.Mace(5)
>>> print(mb.build_model(None, [a3, c1]))
True
>>> print(mb.build_model(None, [a3, c2]))
True
>>> print(mb.build_model(None, [c1, c2]))
False
```

Мы также можем использовать построитель моделей в качестве дополнения к доказателю теорем. Давайте предположим, что мы пытаемся доказать $S \vdash g$, т.е. g логически выводимо из предположений $S = [s_1, s_2, \dots, s_n]$. Мы можем ввести эти же данные в Mace4, и он будет пытаться найти контрпример, то есть показать, что g не следует из S . Таким образом, для данного входа Mace4 попытается найти модель для множества S вместе с отрицанием g , то есть для списка $S' = [s_1, s_2, \dots, s_n, -g]$. Если g не следует из S , то Mace4 вполне может вернуть контрпример быстрее, чем Prover9 придет к выводу, что он не может найти требуемое доказательство. И наоборот, если g выводимо из S , Mace4 может длительное время безуспешно пытаться найти контрмодель и в конечном итоге сдаться.

Рассмотрим конкретный сценарий. Наши посылки - это список [There is a woman that every man loves, Adam is a man, Eve is a woman]. Наш вывод: Adam loves Eva. Может ли Mace4 найти модель, в которой посылки истинны, а заключение ложно? В следующем коде мы используем метод `MaceCommand()`, который позволит нам осмотреть модель, которая была построена.

```
>>> a4 = read_expr('exists y. (woman(y) & all x. (man(x) -> love(x,y)))')
>>> a5 = read_expr('man(adam)')
>>> a6 = read_expr('woman(eve)')
>>> g = read_expr('love(adam,eve)')
>>> mc = nltk.MaceCommand(g, assumptions=[a4, a5, a6])
>>> mc.build_model()
True
```

Так что ответ да: Mace4 нашел контрмодель, в которой есть какая-то женщина, кроме Евы, которую Адам любит. Но давайте подробнее рассмотрим модель Mace4, преобразованную в формат, который мы используем для оценок.

```
>>> print(mc.valuation)
```

```
{'C1': 'b',
  'adam': 'a',
  'eve': 'a',
  'love': {('a', 'b')},
  'man': {('a',)},
  'woman': {('a',), ('b',)}}
```

Общая форма этой оценки должна быть вам знакома: она содержит некоторые индивидуальные константы и предикаты, каждый из которых имеет надлежащий вид значения. Что может быть незнакомым, так это `C1`. Это "константа Сколема", которую наш построитель модели вводит в качестве представителя квантификатора существования. То есть, когда построитель модели столкнулся с `exists y` частью `a4`, он знал, что есть некоторый индивидуальный объект `b` в данной области, который удовлетворяет открытой формуле в теле `a4`. Однако он не знает, является ли `b` так же обозначением индивидуальной константы где-либо еще в его входе, так что он создает новое имя для `b` "на лету", а именно `C1`. Теперь, так как наши послышки ничего не говорят об индивидуальных константах `adam` и `eve`, построитель модели решил, что нет никаких оснований рассматривать их как обозначающие различные объекты, и они оба были сопоставлены с `a`. Кроме того, мы не указали, что `man` и `woman` обозначают непересекающиеся множества, поэтому построитель модели позволяет их денотациям перекрывать друг друга. Это иллюстрирует довольно резко неявное знание, которые мы привносим в интерпретацию нашего сценария, о котором построитель модели ничего не знает. Поэтому давайте добавим новое предположение, которое делает множества мужчин и женщин не пересекающимися. Построитель модели все равно производит контрмодель, но на этот раз она больше соответствует нашим интуициям о ситуации:

```
>>> a7 = read_expr('all x. (man(x) -> -woman(x))')
>>> g = read_expr('love(adam,eve)')
>>> mc = nltk.MaceCommand(g, assumptions=[a4, a5, a6, a7])
>>> mc.build_model()
True
>>> print(mc.valuation)
{'C1': 'c',
  'adam': 'a',
  'eve': 'b',
  'love': {('a', 'c')},
  'man': {('a',)},
  'woman': {('c',), ('b',)}}
```

По здравому размышлению мы можем увидеть, что в наших послылках нет ничего, что говорит, что Ева является единственной женщиной в области дискурса, поэтому контрмодель, на самом деле, является приемлемой. Если бы мы хотели исключить это, мы должны были бы добавить еще одно предположение, такое как `exists y. all x. (woman(x) -> (x = y))`, чтобы гарантировать, что есть только одна женщина в модели.

.

Резюме

- Логика первого порядка является подходящим языком для представления смысла естественного языка в вычислительной среде, поскольку эта логика является достаточно гибкой, чтобы представить множество полезных аспектов естественного осмысления, и существуют эффективные программы доказательства

теорем для рассуждений с использованием логики первого порядка. (Равно как существует множество явлений в семантике естественного языка, которые, как полагают, требуют более мощных логических механизмов.)

- Равно как мы можем переводить предложения естественного языка на логику первого порядка, мы можем устанавливать условия истинности этих предложений путем изучения моделей формул первого порядка.
- Для того чтобы построить представления смысла композиционно, мы дополняем логику первого порядка λ исчислением.
- β -сокращение в λ исчислении соответствует семантически применению функции к аргументу. Синтаксически оно включает в себя замену переменной, связанной с помощью λ в функциональном выражении, на выражение, которое предоставляет аргумент в применении функции.
- Одним из ключевых элементов создания модели заключается в построении оценки, которая назначает интерпретации нелогическим константам. Они интерпретируются либо как n -мерные предикаты, либо как индивидуальные константы.
- Открытое выражение представляет собой выражение, содержащее одну или несколько свободных переменных. Открытые выражения получают интерпретацию, только когда их свободные переменные получают значения из назначения переменных.
- Квантификаторы интерпретируются при создании для формулы $\phi[x]$ открытой в переменной x , множества индивидуальных объектов, которые делают $\phi[x]$ истинной, когда присвоение g присваивает их в качестве значения x . Квантификатор затем накладывает ограничения на это множество.
- Выражение является замкнутым, если оно не имеет свободных переменных; то есть все переменные связаны. Замкнутое предложение является истинным или ложным относительно всех присвоений переменных.
- Если две формулы отличаются только буквой переменной, связанной связующим оператором (т.е. λ или квантификатором), они называются α -эквивалентами. Результат переобозначения связанной переменной в формуле называется α -преобразованием.
- Для данной формулы с двумя вложенными квантификаторами Q_1 и Q_2 самый внешний квантификатор Q_1 , говорят, имеет широкую сферу (или сферу, включающую Q_2). Английские предложения часто неоднозначны относительно сферы действия квантификаторов, которые они содержат.
- Английские предложения могут быть ассоциированы с семантическим представлением путем использования sem в качестве свойства в грамматике на основе свойств. Значение sem сложных выражений, как правило, включает в себя функциональное применение значений sem компонентных выражений.