Введение в ИИ на примере языка Prolog

Формальные грамматики на Prolog

https://github.com/Inscriptor/IntroductionToAI/tree/master/pdf

Федеральное государственное автономное образовательное учреждение высшего образования
«Новосибирский национальный исследовательский государственный университет»

14 октября 2019 г.

Понятие формальных грамматик

Понятие формальных грамматик

Интуитивное определение

Одной из первых и основных областей применения Пролога является компьютерная лингвистика — обработка естественных языков автоматически.
Формальная грамматика — это множество правил, которые определяют, какие фразы из заданного алфавита (лексикона) являются *синтаксически корректными*.
Пролог предоставляет средства для описания подобных правил, а, следовательно, для задания грамматик, что позволяет проверить любую фразу на корректность относительно заданной грамматики, а также по грамматике сгенерировать все возможные фразы, корректные относительно нее.
Все фразы, синтаксически корректные относительно грамматики, называются **языком, порождаемым грамматикой**.

Понятие формальных грамматик

Интуитивное определение

Контекстно-свободная грамматика является частным случаем формальной грамматики.

$S \rightarrow \text{nounPhrase verbPhrase}$
$\text{nounPhrase} \rightarrow \text{article noun}$
$\text{verbPhrase} \rightarrow \text{verbExpr nounPhrase}$
$\text{verbExpr} \rightarrow \text{modalVerb verb prep}$
$\text{article} \rightarrow a$
$\text{article} \rightarrow the$
$\text{noun} \rightarrow cat$
$\text{noun} \rightarrow king$
$\text{modalVerb} \rightarrow may$
$\text{verb} \rightarrow look$
$\text{prep} \rightarrow at$

Понятие формальных грамматик

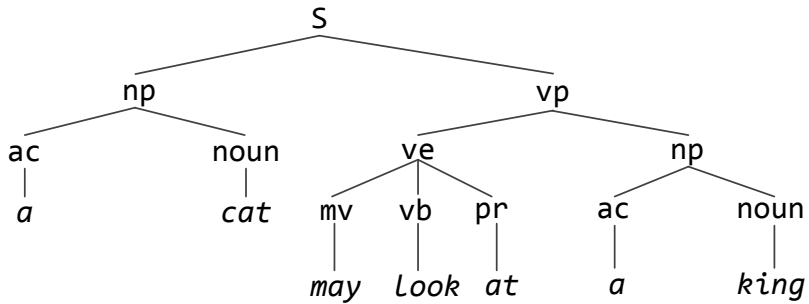
Дерево разбора

- Данная грамматика содержит 11 правил.
- Символ $\rightarrow$ означает переход в правиле: сущность из левой части правила можно разложить на составляющие таким образом, как указано в правой части.
- `S`, `nounPhrase`, `verbPhrase`, `article`, `noun`, `verbExpr`, `modalVerb`, `verb`, `prep` — нетерминальные символы (нетерминалы).
- *a*, *the*, *cat*, *king*, *may look at* — терминальные символы (терминалы). Множество терминалов также называют **алфавитом** или **лексиконом**.

Понятие формальных грамматик

Дерево разбора

Рассмотрим старую английскую поговорку *A cat may look at a king*.



Понятие формальных грамматик

Дерево разбора

Дерево разбора (parse tree) содержит информацию о синтаксической корректности и о структуре заданной фразы.

1. **Программа-распознаватель** (recognizer) по заданной строке сообщает, является ли эта строка синтаксически корректной относительно грамматики.
2. **Программа-парсер** сообщает, является ли фраза синтаксически корректной и строит parse tree.

Понятие формальных грамматик

Контекстно-свободные языки

- **Контекстно-свободным** называется язык, порождаемый контекстно-свободной грамматикой.
- Среди естественных языков контекстно-свободными являются, например, английский, фразцузский и немецкий языки.
- Многие языки программирования являются контекстно-свободными языками.
- Контекстно-свободные грамматики применяются при разработке компиляторов.

Реализация КСГ на Прологе

Реализация КСГ на Прологе

Recognizer

Реализуем грамматику, приведенную выше. Сначала опишем правила стандартными средствами Prolog.

```
sentense(S) :- nounPhrase(NP), verbPhrase(VP), append(NP,VP,S).
nounPhrase(NP) :- article(A), noun(N), append(A,N,NP).
verbPhrase(VP) :- verbExpr(VE), nounPhrase(NP), append(VE,NP,VP).
verbExpr(VE) :- modalVerb(MV),verb(V),prep(P),append([MV,V,P],VE).
article([A]) :- lexicon(''article'', A).
noun([N]) :- lexicon(''noun'',N).
modalVerb([MV]) :- lexicon(''modal verb'',MV).
verb([V]) :- lexicon(''verb'',V).
prep([P]) :- lexicon(''prep'',P).
```

Реализация КСГ на Прологе

Recognizer

Предикат `lexicon/2` используется для описания алфавита. Таким образом мы отделяем описание доступных нам лексем от описания грамматики. Это удобно.

```
lexicon(''article'',''a'').
lexicon(''article'',''the'').
lexicon(''noun'',''cat'').
lexicon(''noun'',''king'').
lexicon(''verb'',''look'').
lexicon(''modal verb'',''may'').
lexicon(''prep'',''at'').
```

Реализация КСГ на Прологе

Recognizer

Два предиката для удобства запуска программы. Первый распознает введенную строку, отвечая `true` или `false`, а второй генерирует все фразы языка, порождаемого грамматикой.

```
recognize :- write('Enter the phrase: '),
             current_input(In),
             read_string(In, '\n', '\r\t', _, Phrase),
             split_string(Phrase, ' ', '', ListPhrase),
             sentense(ListPhrase),!.
generate :- sentense(Phrase),
             atomics_to_string(Phrase,' ',String),
             write(String).
```

Реализация КСГ на Прологе

Recognizer

Данная программа не будет эффективной.

- Она сначала пытается угадать фразу, а затем сравнивает ее с заданной.
- Вопрос `sentense(['a','cat','may','look','at','a','king'])` заставит программу проверять все возможные фразы до тех пор, пока очередная не совпадет с нужной.
- Причина в том, что в предикаты `nounPhrase, verbPhrase` и прочие поступают неопределенные переменные, которые требуют означивания.
- Это можно исправить, заставив программу сначала разбить фразу на части, а затем проверять эти части на соответствие правилам грамматики.

Реализация КСГ на Прологе

Recognizer

В нашем случае это поможет при распознавании, но сломает генератор языка X_X
Кроме того, операция append очень дорогая и неэффективная.

```
sentense(S) :- append(NP,VP,S), nounPhrase(NP), verbPhrase(VP).
nounPhrase(NP) :- append(A,N,NP), article(A), noun(N).
verbPhrase(VP) :- append(VE,NP,VP), verbExpr(VE), nounPhrase(NP).
verbExpr(VE) :- append([MV,V,P],VE),
                modalVerb(MV),
                verb(V),
                prep(P).
```

Реализация КСГ на Прологе

Разностные списки

Разностный список `L` представляется в виде двух списков `A` и `B` таких, что `L = A \ B`. Первый список в паре содержит то, что надо оставить, включить в список `L`, а второй — то, что следует отбросить из того, что мы включили. Один и тот же список можно представить в виде разностного списка бесконечным числом способов.

```
[a, cat, may, look, at, a, king] []
[a, cat, may, look, at, a, king, wtf, omg] [wtf, omg]
```

Реализация КСГ на Прологе

Разностные списки

Так будет выглядеть описание грамматики с разностными списками вместо append.

```
sentense(S,D) :- nounPhrase(S,VP), verbPhrase(VP,D).
nounPhrase(NP,D) :- article(NP,N), noun(N,D).
verbPhrase(VP,D) :- verbExpr(VP,VE), nounPhrase(VE,D).
verbExpr(VE,D) :- modalVerb(VE,MV), verb(MV,V), prep(V,D).
article([A|D],D) :- lexicon(''article'',A).
noun([N|D],D) :- lexicon(''noun'',N).
modalVerb([MV|D],D) :- lexicon(''modal verb'',MV).
verb([V|D],D) :- lexicon(''verb'',V).
prep([P|D],D) :- lexicon(''prep'',P).
```

Реализация КСГ на Прологе

Специальный синтаксис

Теперь рассмотрим, какие специальные средства предоставляет Prolog для описания грамматик.

```
sentense --> nounPhrase, verbPhrase.
nounPhrase --> article, noun.
verbPhrase --> verbExpr, nounPhrase.
verbExpr --> modalVerb, verb, prep.
article --> [''a''].
article --> [''the''].
noun --> [''cat''].
noun --> [''king''].
modalVerb --> [''may''].
verb --> [''look''].
prep --> [''at''].
```

Рекурсивные правила

Рекурсивные правила

Создаем соединительное правило

Допустим, мы хотим добавить соединительные правила для генерации бесконечных последовательностей утверждений.

$$
\begin{aligned}
&\text{sentense} \rightarrow \text{sentense conjunction sentense} \\
&\text{conjunction} \rightarrow \text{and} \\
&\text{conjunction} \rightarrow \text{or} \\
&\text{conjunction} \rightarrow \text{but}
\end{aligned}
$$

Рекурсивные правила

Создаем соединительное правило

Нет ничего проще.

```
sentense --> sentense, conjunction, sentense.
conjunction --> [''and''].
conjunction --> [''or''].
conjunction --> [''but''].
```

Рекурсивные правила

Что может пойти не так?

- Если добавить рекурсивное правило в начало, то любой запрос на распознавание фразы приведет к бесконечному циклу и, как следствие, зависанию. Это произойдет потому, что в правиле первым стоит рекурсивный вызов. Любой запрос будет натыкаться на первое правило и бесконечно его применять.
- Если убрать рекурсивное правило в конец, то синтаксически правильные фразы будут распознаваться, но введение фразы, не являющейся верной в данной грамматике, опять-таки приведет к уходу в бесконечный цикл. На этот раз потому, что без возможности применить первое правило мы будем бесконечно пытаться применить второе.
- В случае обычной рекурсии такой эффект устраняется перестановкой рекурсивного вызова с первого места дальше. Но в случае грамматик последовательность предикатов в теле правила соответствует последовательности слов в синтаксически верных фразах, следовательно мы не можем менять предикаты местами.

Рекурсивные правила

Что может пойти не так?

Так что же делать? **Вводить дополнительные нетерминалы**.

```
plainSentense --> nounPhrase, verbPhrase.
sentense --> plainSentense.
sentense --> plainSentense, conjunction, sentense.
```

Рекурсивные правила

Формальные языки

Рассмотрим формальный язык $a^n b^n$. Все слова данного языка состоят из двух последовательностей из равного количества букв a и b, идущих друг за другом. Пустое слово также является словом данного языка.

Рекурсивные правила

Формальные языки

Рассмотрим формальный язык $a^n b^n$. Все слова данного языка состоят из двух последовательностей из равного количества букв a и b, идущих друг за другом. Пустое слово также является словом данного языка.

```
s -> [].
s -> [a],s,[b].
```

Упражнения

Упражнения

1. Реализуйте грамматику для языка $a^n b^n - \{\mathcal{E}\}$.
2. Реализуйте грамматику для языка $a^n b^{2m} c^{2m} d^n$.

Задачи для самостоятельной работы

Задачи для самостоятельной работы

Реализовать контекстно-свободную грамматику для проверки корректности S-выражений в языке Clojure. Будем рассматривать лишь его подмножество, исключив специфические компоненты.

Основа синтаксиса определяется следующим образом:

1. **Разделитель**. Пробел, табуляция, конец строки, запятая.
2. **Атом**.
   - 2.1 Число. Ограничимся целыми.
   - 2.2 Строка. Любая последовательность символов в двойных кавычках.
   - 2.3 Идентификатор. Либо последовательность букв, цифр и спецсимволов, начинающаяся с буквы, либо последовательность спецсимволов. Например `->` является валидным идентификатором. Можно ограничиться спецсимволами `+`, `-`, `>`, `<`, `=`.
   - 2.4 Ключевые слова. Ключевое слово — это идентификатор, предваряемый двоеточием. Например `:Num`, `:x`, `:Identifier`.
3. **S-выражение**.

Задачи для самостоятельной работы

S-выражение рекурсивно строится из атомов, разделителей и скобок.

- Любой атом является S-выражением.
- Последовательность S-выражений, разделенных разделителями и заключенная в круглые скобки, является S-выражением. **(S1 S2 S3,S4)**.
- Последовательность S-выражений, разделенных разделителями и заключенная в квадратные скобки, является S-выражением. **[S1 S2 S3 S4]**.
- Последовательность из четного числа S-выражений, разделенных разделителями и заключенная в фигурные скобки, является S-выражением. **{S1 S2 S3 S4}**.

Задачи для самостоятельной работы

Программа должна принимать S-выражение, записанное в виде строки, и отвечать `true` в случае, когда выражение корректно относительно синтаксиса языка Clojure, и `false` в противном случае.

Пример

s_expression(”(inc 1)“).
s_expression(”(+ [x y])“).
s_expression(”((lambda x (nth [(tail x) 0])) (list \”Some string\“))“).
s_expression(”{list (-> A (:List A)),nth (-> (cross (:List A) :Num) (:List A)), tail (-> (:List A) (:List A))}“).