ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Костюк Ю. Л.

ТЕОРИЯ АВТОМАТОВ И ФОРМАЛЬНЫХ ЯЗЫКОВ

Лекция 1

ФОРМАЛЬНЫЕ ЯЗЫКИ

$\sum$ – конечное множество из m символов (алфавит).

Последовательность символов – цепочка.

Пример: из символов $\{a, b\}$ можно получить цепочки: $\{a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb, \ldots\}$.

Обозначения цепочек – строчными греческими буквами: $\alpha, \beta, \gamma, \ldots$

Длина цепочки α – количество символов в ней.

Количество всех различных цепочек длины n из алфавита m символов: m^n.

Операция ***конкатенации*** соединяет две цепочки: $\alpha\beta$. Их длины суммируются: $|\alpha\beta| = |\alpha| + |\beta|$.

Цепочка длины 0 (пустая) обозначается буквой λ.

Количество всех различных цепочек длины не более чем n (включая λ) из алфавита m символов:

$$1 + m + m^2 + \ldots + m^n = (m^{n+1} - 1)/(m - 1).$$

Множество цепочек длины n обозначается как $\sum^n$

$\sum^+$ – множество всех цепочек длины от 1 до ∞:

$$\sum^+ = \sum \cup \sum^2 \cup \sum^3 \cup \ldots \cup \sum^n \cup \ldots$$

называется ***положительным транзитивным замыканием*** множества $\sum$.

$\sum^*$ – множество всех цепочек длины от 0 до ∞ называется ***транзитивным замыканием*** множества $\sum$:

$$\sum^* = \sum^+ \cup \{\lambda\}$$

Язык L – некоторое множество цепочек в алфавите $\sum$, которое есть подмножество $\sum^*$.

Порождающая грамматика

$$G(L) = \{\sum, N, S, P\}$$

$\sum$ – алфавит символов языка (терминальные символы, терминалы);

N – алфавит вспомогательных символов грамматики (грамматические понятия, нетерминальные символы, нетерминалы);

S – множество начальных нетерминалов (обычно 1);

P – множество порождающих правил вида

$$\alpha \rightarrow \beta,$$

где α, β – цепочки терминалов и нетерминалов, причем α содержит ***хотя бы 1 нетерминал***.

Знак $\rightarrow$ (порождает) означает ***замену*** (или ***подстановку***) подцепочки α на подцепочку β. α – левая часть, β – правая часть правила.

Процесс порождения

1. Одно из правил множества P, в котором левая часть – начальный нетерминал, (пусть это правило $S \rightarrow \beta_1$) начинает порождение: из цепочки S получается β_1.
2. Какая либо подцепочка α, входящая в β_1, заменяется в соответствии с правилом $\alpha \rightarrow \beta$, в результате вся цепочка β_1 превращается в β_2.

   Шаги 1 и 2 запишем так: $S \Rightarrow \beta_1 \Rightarrow \beta_2$.
3. Шаг 2 повторяется до тех пор, пока возможна какая либо из замен в цепочке β_2 в соответствии с правилами из множества P. В результате получится некоторая цепочка γ.

Если цепочка γ состоит только из терминалов (символов языка) или $\gamma = \lambda$, то процесс порождения успешно завершен. Иначе – тупик, безуспешное порождение. Если шаг 2 никак завершиться не может (процесс зациклил), то это также безуспешное порождение.

Процесс порождения

Всех вариантов порождения может быть очень много, часто бесконечно много, количество разных получающихся при этом терминальных цепочек также может быть бесконечно много.

В последнем случае язык, являющийся множеством всех различных получающихся при этом терминальных цепочек, ***бесконечен***.

Можно представить алгоритм, генерирующий ***все возможные варианты*** порождения для заданной грамматики. Но для бесконечного языка такой алгоритм будет работать бесконечно долго.

Пример порождения

Далее везде в примерах: заглавные буквы обозначают нетерминалы, строчные буквы – терминалы.

Грамматика: $G_1(L) = \{\sum, N, S, P\}$

$\sum = \{a, b\}, \quad N = \{S, T\}, \quad S = \{S\},$

$P = \{S \to aTa, S \to aS, aT \to Tb, aT \to T, T \to b\}.$

Варианты порождений:

- $S => aTa => aba$
- $S => aTa => Ta => ba$
- $S => aS => aaTa => aaba$
- $S => aS => aaTa => aTba => abba$
- $S => aS => aaTa => aTba => Tbba => bbba$
- . . .

Классификация порождающих грамматик по Хомскому

Класс 0: нет дополнительных ограничений.

Класс 1: контекстно-зависимые грамматики (***КЗ-грамматики***). Вид порождающих правил:

$$\omega_1 A \omega_2 \rightarrow \omega_1 \beta \omega_2$$

где A – нетерминал, $\beta \neq \lambda$, т.е. все правила КЗ-грамматики должны быть неукорачивающими.

Класс 2: контекстно-свободные грамматики (***КС-грамматики***). Вид порождающих правил:

$$A \rightarrow \beta$$

где A – нетерминал, β – любая цепочка (в т.ч. $\beta = \lambda$).

Класс 3: автоматные грамматики (***А-грамматики***). Вид порождающих правил:

$$A \rightarrow aB \quad \text{или } A \rightarrow a$$

где A, B – нетерминалы, a – терминал.

Пример грамматики, эквивалентной $G_1(L)$ (т.е. порождающей тот же самый язык):

Грамматика: $G_2(L) = \{\sum, N, S, P\}$

$\sum = \{a, b\}, \quad N = \{S, T\}, \quad S = \{S\},$

$P = \{S \to aS, S \to bT, T \to bT, T \to a\}.$

Варианты порождений:

- $S => bT => ba$
- $S => bT => bbT => bba$
- $S => aS => abT => aba$
- $S => aS => aaS => aabT => aaba$
- . . .

$G_1(L)$ – грамматика класса 0,

$G_2(L)$ – грамматика класса 3.

Один и тот же язык может порождаться многими различными грамматиками, в том числе грамматиками, принадлежащими различным классам по Хомскому.

Различные грамматики, порождающие один и тот же язык, называют эквивалентными.

Следствие: класс языка по Хомскому определяется классом наиболее простой из эквивалентных грамматик, порождающих язык.

Левое каноническое порождение: на каждом шаге в полученной к этому моменту цепочке символов заменяется самая левая подцепочка α (по правилу α→β) на подцепочку β.

Грамматика однозначная: если для любой полученной ею цепочки языка существует единственная последовательность применения правил при левом каноническом порождении.

***Грамматика неоднозначная*:** если существует хотя бы одна цепочка языка, которую можно получить более чем одним вариантом левого канонического порождения.

Следствие: язык является однозначным, если среди всех эквивалентных грамматик, порождающих этот язык, существует хотя бы одна однозначная грамматика.

Задача грамматического разбора

Задан язык (с помощью грамматики) и задана некоторая цепочка символов.

Задача распознавания принадлежности этой цепочки символов языку (ответ: «да» или «нет»).

Задача грамматического анализа: если ответ «да», то требуется восстановить всю последовательность порождающих правил, результатом применения которых явилась анализируемая цепочка.

По заданной грамматике нужно уметь строить ***автомат*** – ***алгоритм распознавания (алгоритм грамматического анализа)***

ЯЗЫКИ ПРОГРАММИРОВАНИЯ

Языки программирования ***низкого уровня***:
машинные языки, автокоды, ассемблеры, директивные языки – задаются А-грамматиками (3-го типа).

Языки программирования ***высокого уровня***:
почти все современные языки, начиная от Фортрана, (Паскаль, Си и др.) задаются КС-грамматиками (2-го типа), но отдельные слова (***лексемы***) в этих языках задаются А-грамматиками.

Трансляция – процесс преобразования программы на языке программирования в программу, которая может непосредственно исполняться на компьютере (в ***машинную программу*** или программу на некотором промежуточном языке).

Виды трансляторов:

- ***интерпретатор*** последовательно анализирует программу и сразу же исполняет её операторы;
- ***компилятор*** транслирует исходную программу в программу из машинных команд;
- ***компилятор-интерпретатор*** транслирует исходную программу в программу на внутреннем языке, которую затем исполняет.

Мера эффективности транслятора – во сколько раз замедляется исполнение транслируемой программы по сравнению с программой, транслируемой вручную.

Для интерпретатора **– замедление в сотни раз.**

Для компилятора **– в несколько раз.**

Для компилятора-интерпретатора **– в десятки раз.**

Сложность создания трансляторов:

- ***интерпретатор*** – наименьшая сложность;
- ***компилятор*** – наибольшая сложность;
- ***компилятор-интерпретатор*** – средняя сложность.

Фазы работы компилятора:

1) ***лексический анализ*** – распознавание в программе лексем – отдельных слов, описываемых А-грамматикой;
2) ***синтаксический анализ*** всей программы, описываемой КС-грамматикой и ***генерация*** программы на внутреннем языке;
3) ***оптимизация*** программы на внутреннем языке;
4) ***генерация*** машинных команд;
5) ***оптимизация*** программы на машинном языке.

Фазы работы компилятора-интерпретатора:

– вначале фазы 1 и 2, после чего исполнение (интерпретация) программы на внутреннем языке.