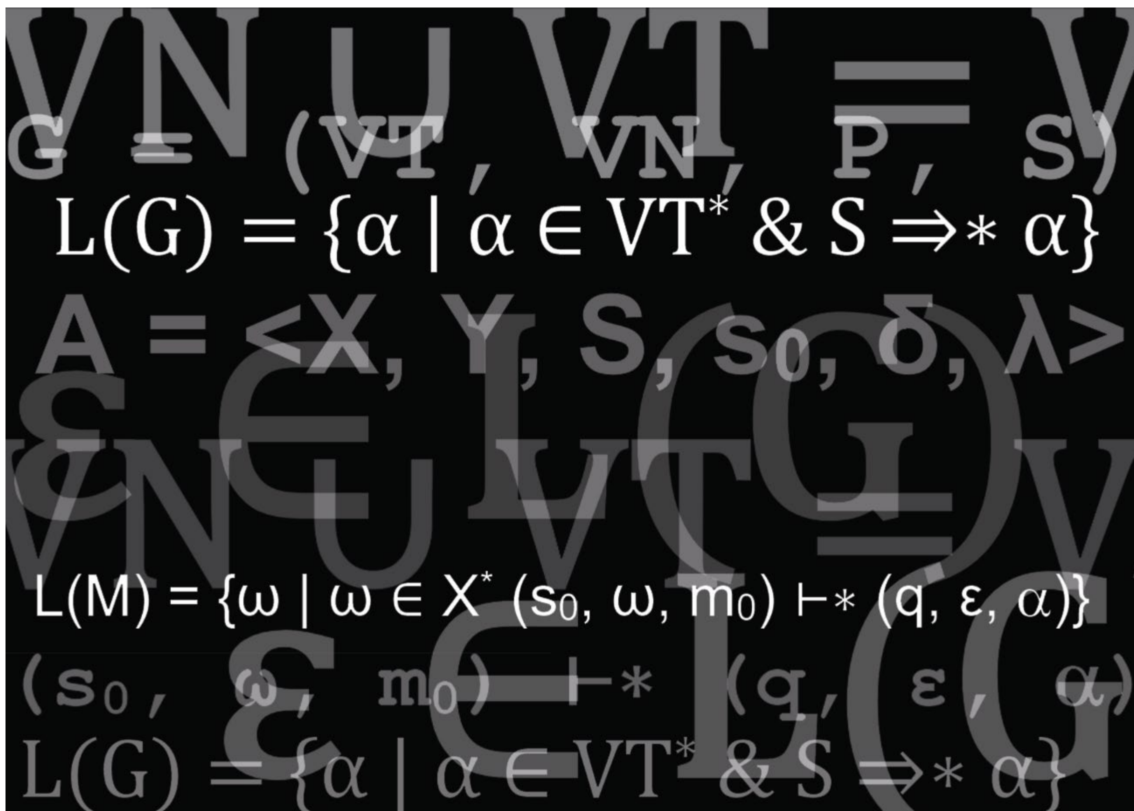
УНИВЕРСИТЕТ ИТМО

А. В. Лаздин

ФОРМАЛЬНЫЕ ЯЗЫКИ, ГРАММАТИКИ, АВТОМАТЫ



Санкт-Петербург
2019

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

УНИВЕРСИТЕТ ИТМО

А. В. Лаздин

ФОРМАЛЬНЫЕ ЯЗЫКИ, ГРАММАТИКИ, АВТОМАТЫ

УЧЕБНОЕ ПОСОБИЕ

РЕКОМЕНДОВАНО К ИСПОЛЬЗОВАНИЮ В УНИВЕРСИТЕТЕ ИТМО
по направлению подготовки 09.03.04 «Программная инженерия» и другим техническим направлениям в качестве учебного пособия для реализации основных профессиональных образовательных программ высшего образования бакалавриата

Санкт-Петербург
2019

УДК 004.4(519.766)

Лаздин А. В. Формальные языки, грамматики, автоматы – СПб: Университет ИТМО, 2019. – 99 с.

Рецензент: Поляков Владимир Иванович, кандидат технических наук, доцент, доцент (квалификационная категория "ординарный доцент") факультета программной инженерии и компьютерной техники, Университета ИТМО.

В пособии рассмотрены порождающие грамматики, их классификация, алгоритмы упрощения и их взаимного преобразования; определены конечные автоматы и МП-автоматы, рассмотрены вопросы построения распознавателей для регулярных и контекстно-свободных формальных языков; теоретический материал дополнен описанием алгоритмов на псевдокоде и примерами.

Учебное пособие предназначено для академического бакалавриата студентов обучающихся по направлению 09.03.04 «Программная инженерия» по программам «Программно-информационные системы», «Системное и прикладное программное обеспечение», «Нейротехнологии и программирование» в рамках дисциплин «Прикладная математика» и «Основы разработки компиляторов». Пособие может быть использовано студентами в качестве конспекта лекционного курса, и для подготовки к практическим и лабораторным работам.

Университет ИТМО – ведущий вуз России в области информационных и фотонных технологий, один из немногих российских вузов, получивших в 2009 году статус национального исследовательского университета. С 2013 года Университет ИТМО – участник программы повышения конкурентоспособности российских университетов среди ведущих мировых научно-образовательных центров, известной как проект «5 в 100». Цель Университета ИТМО – становление исследовательского университета мирового уровня, предпринимательского по типу, ориентированного на интернационализацию всех направлений деятельности.

© Университет ИТМО, 2019
© Лаздин А. В., 2019

Оглавление

Введение 5
1 Математические основы 7
1.1 Множества 7
1.2 Отношения 9
1.3 Функции и отображения 12
2 Языки и грамматики 13
2.1 Символы и цепочки 13
2.2 Порождающие грамматики 15
2.3 Классификация порождающих грамматик Хомского 20
2.4 Соотношения языков и грамматик 21
2.5 Распознаватели для формальных языков 22
Контрольные вопросы 26
3 Регулярные языки и конечные автоматы 27
3.1 Конечные автоматы 27
3.1.1 Способы задания конечного автомата 29
3.1.2 Автомат-распознаватель 31
3.1.3 Недетерминированный конечный автомат 33
3.1.4 От регулярной грамматики к НКА и обратно 34
3.2 Минимизация ДКА 35
3.3 Регулярные выражения 38
3.4 Построение НКА по регулярному выражению 40
3.5 Построение ДКА по НКА с ε-переходами 41
3.6 Лемма о разрастании для регулярных языков 45
3.7 Свойства регулярных языков 46
Контрольные вопросы 48
4 Контекстно-свободные языки и МП-автоматы 49
4.1 Синтаксические деревья 49
4.2 Неоднозначность грамматик 51

4.3 Автоматы с магазинной памятью ... 53
4.4 Упрощение КС-грамматик ... 59
4.4.1 Удаление непроизводящих символов ... 60
4.4.2 Удаление недостижимых символов ... 61
4.4.3 Удаление ε-правил ... 63
4.4.4 Удаление цепных правил. ... 66
4.4.5 Нормальная форма Хомского ... 69
4.4.6 Нормальная форма Грейбах ... 71
4.4.7 Устранение прямой левой рекурсии ... 72
4.4.8 Левая факторизация ... 73
4.5 Распознаватели для контекстно-свободных языков ... 74
4.5.1 Распознаватели с возвратами ... 75
4.5.2 Алгоритм Кока-Янгера-Касами ... 76
4.6 Нисходящие распознаватели КС-языков ... 80
4.6.1 Метод рекурсивного спуска ... 80
4.6.2 Применимость метода рекурсивного спуска ... 82
4.7 Восходящие распознаватели КС-языков ... 87
4.7.1 Грамматики простого предшествования ... 88
4.7.2 Построение отношений простого предшествования 90
4.7.3 Алгоритм сдвиг-свертка для грамматик простого предшествования ... 91
4.8 Лемма о разрастании КС-языков ... 94
4.9 Свойства КС-языков ... 95
Контрольные вопросы ... 96
Список литературы ... 98

Введение

Представленный в пособии материал представляет основы теории формальных языков и грамматик, которая является основополагающей для такой важной составляющей части компьютерных технологий, как конструирование компиляторов. Кроме того, рассматриваются некоторые аспекты теории автоматов, в частности автоматы-распознаватели.

Теория формальных языков берет свое начало от работ Н. Хомского, который в середине пятидесятых годов XX века заложил основы теоретической лингвистики. Предложенная им идея порождающих грамматик для естественных языков нашла необыкновенно плодотворное развитие в области формального определения синтаксиса языков программирования. Предложенная Дж. Бэкусом и П. Науром форма определения синтаксиса языков позволила дать формальное описание синтаксических конструкций языка программирования Алгол-60. Что в свою очередь позволило формализовать методы синтаксического анализа при проектировании компиляторов.

В настоящее время использование теории формальных языков и грамматик далеко вышло за границы области языков программирования. Широко используемые языки разметки документов и гипертекста, языки описания интерфейсов и языки спецификаций, языки описания аппаратуры, языки описания распределённых систем и коммуникационных протоколов требуют проектирования инструментов их анализа и, возможно, трансляции, в которых широко используются методы и алгоритмы из теории формальных языков и грамматик.

В учебном пособии дано формальное определение языка, на основе которого рассматриваются вопросы, связанные с построением и преобразованием порождающих грамматик и распознавателей для языков, порождаемых этими грамматиками. Для большинства алгоритмов, представленных в пособии, приводятся описание на псевдокоде, что, возможно, упростит их программную реализацию. Кроме того, все алгоритмы снабжены примерами, позволяющими наглядно продемонстрировать их работу.

Первый раздел знакомит с базовыми математическими понятиями, которые используются в учебном пособии. К ним относятся элементы теории множеств, связанные с ними понятия отношений, отображений и функций.

Во втором разделе рассматриваются основополагающие понятия курса: порождающие грамматики и формальные языки. Для порождающих

грамматик приведена классификация Хомского. Обсуждаются вопросы отношений между формальными грамматиками и языками. Показаны способы задания языков на примерах порождающих грамматик, распознавателей и множественно-теоретического подхода.

В третьем и четвертом разделах рассматриваются вопросы построения распознавателей для регулярных и контекстно-свободных грамматик соответственно Эти разделы построены по общей схеме: сначала рассматриваются математические основы проектирования распознавателей, которые рассматриваются как абстрактные машины, затем предлагаются способы реализации этих машин на уровне алгоритмов, которые могут быть программно реализованы. В конце каждого раздела приводятся способ определения типа языка (лемма о накачке) и свойства языков.

В третьем разделе приводятся минимально необходимые сведения из теории конечных автоматов: вводится понятие конечного автомата, автомата-распознавателя, рассмотрен вопрос минимизации детерминированных конечных автоматов. Приведены алгоритмы преобразования регулярных грамматик и регулярных выражений в конечные автоматы.

В четвертом разделе вводится понятие автомата с магазинной памятью как абстрактной машины для распознавания контекстно-свободных языков. Рассматриваются вопросы построения синтаксических деревьев и их применения для оценки неоднозначности контекстно-свободных языков. В разделе рассматриваются табличные методы распознавания, а также распознаватели, построенные на базе метода рекурсивного спуска (нисходящий разбор) и применения грамматик простого предшествования (восходящий разбор). Значительная часть раздела посвящена алгоритмам эквивалентных преобразований контекстно-свободных грамматик.

Разделы снабжены списком вопросов для самоконтроля.

Пособие может использоваться в качестве базового курса для дисциплин, связанных с проектированием языков программирования и описания данных, а также для построения компиляторов.

1 Математические основы

В этом разделе представлен минимальный набор математических понятий, в основном из дискретной математики, которые используются в изложении основных разделов пособия. Этот раздел не претендует на всеобъемлющую полноту, например, не затронута тема графов и деревьев.

1.1 Множества

Понятие «множество» относится к фундаментальным математическим понятиям. *Множество* — это определенная совокупность объектов. Объект, принадлежащий множеству, называется *элементом* этого множества. Ели x является элементом множества A, то это будет обозначаться: $x \in A$, или иначе: элемент x принадлежит множеству A. Если объект b не принадлежит множеству A, то это будет обозначаться: $b \notin A$. Множество натуральных чисел, меньших восьми, будет обозначаться: $\{1, 2, 3, 4, 5, 6, 7\}$. В этом случае $1 \in \{1, 2, 3, 4, 5, 6, 7\}$, а $9 \notin \{1, 2, 3, 4, 5, 6, 7\}$. Множество может быть элементом другого множества. Следовательно, если $A = \{1, 2, \{3, 4, 5\}, 6, 7\}$, то $2 \in A$, но $8 \notin A$. Заметим, что $5 \notin A$, но $\{3, 4, 5\} \in A$.

Пустое множество – это множество, которое не содержит ни одного элемента, обозначается: $\emptyset$. В этом смысле $\emptyset = \{\}$. Не следует путать пустое множество, и множество, единственным элементом которого является пустое множество: $\{\emptyset\}$.

Множество является *конечным*, если существует неотрицательное число n, равное количеству элементов этого множества. *Мощностью* конечного множества является количество его элементов, обозначается $|A|$.

$|\{1, 2, 3, 4, 5, 6, 7\}| = 7$; $|\emptyset| = 0$; $|\{\emptyset\}| = 1$.

В общем случае множество может содержать бесконечное количество элементов, например: множество натуральных чисел. Порядок элементов в множестве не важен. Кроме того, будем считать, что множества, не содержат одинаковых элементов.

Чтобы задать множество, необходимо указать элементы, которые ему принадлежат. Это можно сделать с помощью перечисления, или характеристического предиката, например:

$A_7 = \{1, 2, 3, 4, 5, 6, 7\}$.

$A_7 = \{x \mid x \in \mathbb{N} \ \& \ x<8\}$, где $\mathbb{N}$ – множество натуральных чисел. Кроме этого, множество можно задать порождающей процедурой [10].

Говорят, что множество A является подмножеством множества B, если каждый элемент множества A является элементом множества B. Это обозначается: $A \subset B$; если множество A не является подмножеством B, то это обозначается: $A \not\subset B$. Следовательно, $\{a, b, c\} \subset \{a, b, c, d, e, f\}$, но $\{a, b, g\} \not\subset \{a, b, c, d, e, f\}$.

Множество A эквивалентно множеству B, если $A \subset B$ и $B \subset A$; следовательно, два множества эквивалентны, если они содержат одинаковые элементы. Для любого множества A справедливо: $A \subset A$.

Операции над множествами.

Пусть A и B – множества, тогда для них определены следующие операции:

пересечение

$$A \cap B = \{x \mid x \in A \ \& \ x \in B\};$$

объединение

$$A \cup B = \{x \mid x \in A \vee x \in B\};$$

разность

$$A \setminus B = \{x \mid x \in A \ \& \ x \notin B\};$$

дополнение

$$\overline{A} = \{x \mid x \notin A\}.$$

Примечание: операция дополнения подразумевает, что задан некий универсум U, $\overline{A} = U \setminus B$, в противном случае операция дополнения не определена.

Если x – некоторый объект, причем $x \notin A$, то можно определить операцию добавления объекта x в множество A следующим образом: $A + x = \{a \mid a \in A \vee a = x\}$.

Если A – множество и $x \in A$, то можно определить операцию удаления элемента из множества следующим образом: $A - x = \{a \mid a \in A \ \& \ a \neq x\}$.

Если x и y – некоторые объекты, то (x, y) обозначает упорядоченную пару, причем в общем случае $(x, y) \neq (y, x)$. Пусть A и B – множества, тогда можно определить *прямое произведение* этих множеств следующим образом: $A \times B = \{(a, b) \mid a \in A \ \& \ b \in B\}$. Для множеств $A = \{1, 2\}$ и $B = \{a, b, c\}$, их произведение $A \times B = \{(1, a), (1, b), (1, c), (2, a), (2, b), (2, c)\}$.

Для любых конечных множеств $|A \times B| = |A| \times |B|$.

Степень множества A – это n-кратное прямое произведение множества на самого себя $A^1 = A$, $A^2 = A \times A$, … $A^n = A^{n-1} \times A$.

Множество всех подмножеств множества A называется *булеан* и обозначается 2^A. Для множества B из примера выше $2^B = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$. Если множество B конечно, то $|2^B| = 2^{|B|}$.

1.2 Отношения

Пусть A и B – два множества. *Бинарным отношением* из множества A в множество B называется множество R, которое является любым подмножеством A × B, иначе $R \subset A \times B$. Если $(a, b) \in R$, то это часто обозначают следующим образом: aRb. R принято называть *графиком* отношения, множество A – *областью отправления*, а множество B – *областью прибытия*. Существование отношения из множества A в множество B никак не влияет на существование или отсутствие бинарного отношения из множества B в множество A. В дальнейшем вместо формулировки «отношения из множества A в множество B», которая подчеркивает порядок элементов множеств в парах множества отношения, будет использоваться более простая формулировка: «отношение между A и B». Обычно, если это никак не влияет на понимание, под отношением будет пониматься бинарное отношение.

Если A = B, то говорят, что отношение задано на множестве A ($R \subset A \times A$).

Областью определения отношения R между множествами A и B являются те элементы множества A, которые присутствуют в хотя бы одной из пар отношения и составляют следующее множество: $\mathrm{Dom}\, R = \{a \in A \mid \exists\, b \in B\ aRb\}$.

Область значений отношения R можно определить следующим образом: $\mathrm{Im}\, R = \{b \in B \mid \exists\, a \in A\ aRb\}$.

Неформальное определение «все нечетные числа из множества A связаны отношением со всеми возможными буквами из множества B с нечетными номерами, а четные числа из множества A – с некоторыми буквами из множества B, имеющими четный номер» для множеств A = {1, 2, 3, 4} и B = {a, b, c, d}, можно, например, задать следующим образом: R = {(1, a), (1, c), (3, a), (3, c), (2, b), (4, d)}.

Пусть R есть отношение из множества A в множество B, *инверсия отношения* R, или *обратное отношение*, обозначаемое как R^{-1}, определяется следующим образом: $R^{-1} = \{(b, a) \mid (a, b) \in R\}$.

Отношение для конечного множества может быть задано с помощью матрицы. Если бинарное отношение задано на множестве A = {a_1, a_2, …, a_n}, то ему соответствует квадратная матрица C порядка n. В этой матрице элемент c_{ij}, стоящий на пересечении i-й строки и j-го столбца, определяется следующим образом:

$$c_{ij} = \begin{cases} 1, \text{если } a_i R a_j; \\ 0, \text{в противном случае.} \end{cases}$$

Такие матрицы можно рассматривать как *бинарные матрицы*, полагая, что значению 0 соответствует «ложь» (нет такого отношения), а значению 1 – «истина» (отношение существует).

Для матриц отношений могут быть введены операции сложения и умножения. Операции сложения для булевых матриц n-го порядка будет соответствовать логическая операция ИЛИ для соответствующих элементов матриц. Для D = B + C, элемент матрицы $d_{ij} \coloneqq b_{ij} \mid c_{ij}$.

Умножение булевых матриц выполняется по тем же правилам, что и для числовых матриц, но операция сложения заменяется операцией ИЛИ, а операция умножения – операцией И. Для $D = B \times C$ элемент $d_{ij} \coloneqq b_{i1} \& c_{1j} \mid b_{i2} \& c_{2j} \mid \ldots \mid b_{in} \& c_{nj}$.

Степень n отношения R, заданного на множестве A, обозначаемая R^n, определяется следующим образом:

aR^1b тогда и только тогда, когда aRb;

aR^nb для n > 1 тогда и только тогда, когда существует такое $c \in A$, что aRc и $cR^{n-1}b$.

Если отношение на множестве задано бинарной матрицей, то n-ю степень отношения можно получить в процессе n-1 умножения матрицы этого отношения на саму себя.

Отношение R и его степени, заданные бинарными матрицами.

R

	a_1	a_2	a_3	a_4	a_5
a_1	0	1	0	1	0
a_2	0	0	0	0	1
a_3	1	0	0	1	0
a_4	0	0	0	0	1
a_5	0	0	1	0	0

R^2

	a_1	a_2	a_3	a_4	a_5
a_1	0	0	0	0	1
a_2	0	0	1	0	0
a_3	0	1	0	1	1
a_4	0	0	1	0	0
a_5	1	0	0	1	0

R^5

	a_1	a_2	a_3	a_4	a_5
a_1	0	1	0	1	1
a_2	0	0	1	0	1
a_3	1	1	0	1	1
a_4	0	0	1	0	1
a_5	1	0	1	1	0

Рис. 1.1 Бинарное отношение и его степени.

Корректность полученных результатов предлагается проверить читателю самостоятельно.

Свойства отношений.

Пусть $R \subset A \times A$, тогда отношение R является:

рефлексивным, если $\forall a \in R$ (aRa);

симметричным, если $\forall a, b \in R$ (aRb) $\Rightarrow$ (bRa);

транзитивным, если $\forall a, b, c \in R$ (aRb) & (aRc) $\Rightarrow$ (aRc);

Отношение R на множестве A является отношением *эквивалентности*, если оно рефлексивно, симметрично, и транзитивно.

Замыкание отношения R относительно свойства P называется такое множество R^C, что:

- $R \subset R^C$;
- R^C обладает свойством P;
- R^C является подмножеством любого другого отношения, содержащего R и обладающего свойством P.

Для любого отношения R отношение R^+ называется транзитивным замыканием отношения. $R^+ = R^1 \cup R^2 \cup R^3 \dots \cup R^n \cup \dots$

Определим R^0 как единичное отношение, aR^0b тогда и только тогда, когда a=b. Единичное отношение представляется бинарной матрицей, на главной диагонали которой расположены единицы. Других единичных значений в этой матрице нет.

Рефлексивное транзитивное замыкание R^* определяется следующим образом: $R^* = R^0 \cup R^+$.

Транзитивное замыкание отношения может быть построено с помощью алгоритма Воршалла (Stephen Warshall) [5].

Алгоритм Воршалла.

вход: матрица бинарного отношения A[n, n]

выход: матрица транзитивного замыкания отношения Res[n, n]

```
temp = A
for (int k = 0; k < size; k++)
    for (int i = 0; i < size; i++)
        for (int j = 0; j < size; j++)
            temp[i,j]= temp[i,j]|temp[i,k]&temp.[k,j]
    Res = temp
```

Рефлексивное транзитивное замыкание может быть получено на основе алгоритма Воршалла путем прибавляется единичной матрицы к матрице транзитивного замыкания отношения.

1.3 Функции и отображения

Пусть A и B – два множества. Закон f, в соответствии с которым каждому элементу $a \in A$ поставлен в соответствие единственный элемент $b \in B$, называется отображением из A в B, или функцией, заданной на A со значениями в B. Отображение обозначается $f : A \to B$. Если $f : A \to B$ – функция и $(a, b) \in f$, мы говорим, что $b = f(a)$. Множество A называется областью определения отображения f; множество B – областью значений отображения f. Элемент $a \in A$ называют аргументом или независимой переменной, а элемент $b \in B$ – значением, или зависимой переменной.

Пусть $f : A \to B$ отображение, тогда это отображение, или функция, f называется:

- *инъективной*, если для любых двух значений $a_1 \in A$ и $a_2 \in A$, $b_1 = f(a_1)$ и $b_2 = f(a_2)$, и при этом $b_1 \neq b_2$;
- *сюръективной*, если $\forall b \in B\ (\exists a \in A\ (b = f(a)))$;
- *биективной*, если она инъективна и сюръективна.

Если $f : A \to B$ – биективна, или взаимно-однозначна, то f^{-1} – также взаимно-однозначная функция.

Если $f : A \to B$ – биективна и $A = B$, то $f : A \to A$ является отображением множества A на себя, или тождественным отображением.

2 Языки и грамматики

Как правило, изучение иностранного языка начинается с алфавита. Это нужно для того, чтобы научится читать и писать слова изучаемого языка. Но состоит ли язык из слов? На это можно дать совершенно определенный ответ – нет. Из слов составляют словари, но словарный запас языком не является. Язык состоит из предложений.

Рассмотрим предложение *«Мама мыла раму»*. Данное предложение является однозначным, так как не возникает вопроса кто выполняет действие, что это за действие, продолжается или закончено ли оно.

Не все предложения являются однозначными, например, предложение: *«День сменит ночь»* не является однозначным. Не вполне понятно, что придет на смену чего. Это пример неоднозначного предложения.

Не всякая последовательность слов из словаря языка может быть предложением языка. Очевидно, что *«Стандарт отглагольного много город»* не является предложением русского языка, так как не соответствует синтаксическим правилам.

Предложение *«Делопроизводство трепещет синусоидами»*, судя по всему, бессмысленно, хотя не нарушает ни одного правила синтаксиса русского языка.

Можно сделать, пока неформальное, определение понятия *язык* – это подмножество множества всех возможных комбинаций слов из словаря, если мы говорим о русском языке, то подразумевается орфографический словарь русского языка.

В этом разделе будет предложена формализация таких понятий, как алфавит, грамматика, предложение, язык, синтаксис, и других.

2.1 Символы и цепочки

Конечное непустое множество символов задает *алфавит* Σ. Обычно для символов алфавита задается отношение следования, однако в рамках данного пособие это несущественно.

Всякая конечная последовательность символов из заданного алфавита называется *цепочкой*. В качестве символов алфавита будут использоваться прописные латинские буквы, а для именования цепочек – буквы греческого алфавита: α, β, γ, δ. Например, для алфавита Σ = {a, b, c} цепочками являются aba, bbbaaa, abc. В дальнейшем запись

вида $\alpha = aabbcc$ будет обозначать, что цепочка, названная α, определяется значением aabbcc.

Для цепочек вводится операция *конкатенации*. Конкатенацией двух цепочек α и β будет цепочка, полученная в результате добавления символов из цепочки β справа от символов, входящих в цепочку α. Так, например, если α = abc и β = cba, то результатом конкатенации этих цепочек, обозначаемой αβ, будет цепочка αβ = abccba.

Символом ε будем обозначать *пустую цепочку*, т. е. цепочку, не содержащую ни одного символа. Очевидно, что $\alpha = \varepsilon\alpha = \alpha\varepsilon = \varepsilon\alpha\varepsilon$.

Степень цепочки определяется следующим образом: пусть а – цепочка, тогда $\alpha^0 = \varepsilon$ – пустая цепочка, $\alpha^1 = \alpha$, $\alpha^5 = \alpha\alpha\alpha\alpha\alpha$. Индуктивно степень цепочки может быть определена следующим образом: $\alpha^0 = \varepsilon$; $\alpha^n = \alpha^{n-1}\alpha$.

Любая цепочка последовательных символов в α называется подцепочкой α. Если $\alpha = \beta\gamma$, то подцепочки β и γ являются префиксом и суффиксом цепочки α, соответственно. Так, например, для цепочки α = abbac, множество {ε, a, ab, abb abba, abbac} задает все возможные префиксы цепочки α, а множество {c, ac, bbac} – некоторые из её суффиксов.

Длина цепочки, обозначается $|\alpha|$, есть количество символов в этой цепочке. Для α = ababab $|\alpha| = 6$.

Введем понятие *степень алфавита*. Пусть Σ – алфавит, причем Σ = {k, l, m}. Тогда нулевая степень алфавита – это множество, включающее единственный элемент – пустую цепочку $\Sigma^0 = \{\varepsilon\}$; первая степень алфавита – это множество, включающее все возможные цепочки длиной в один символ: $\Sigma^1 = \{k, l, m\}$; вторая степень алфавита – множество всех цепочек состоящих из двух символов: $\Sigma^2 = \{kk, kl, km, lk, ll, lm, mk, ml, mm\}$; очевидно, что $\Sigma^n = \Sigma^{n-1} \times \Sigma$ есть множество всех возможных цепочек из символов алфавита Σ длиной n символов.

Усеченной итерацией алфавита, обозначается Σ^+, является объединений всех натуральных степеней этого алфавита: $\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \ldots \cup \Sigma^n \cup \ldots$

Итерация алфавита, обозначается Σ^* и определяется $\Sigma^* = \Sigma^0 \cup \Sigma^+$, определяет бесконечное множество всех возможных цепочек, которые могут быто получены из символов этого алфавита, включая и пустую цепочку ε.

2.2 Порождающие грамматики

Одним из главных средств задания языка является грамматика. Правило подстановки (*правило грамматики*) – это упорядоченная пара (α, β), которая записывается следующим образом: α ::= β, где α – непустая конечная цепочка, β – конечная цепочка, возможно пустая, а символ ::= обозначает «есть по определению». Цепочку α называют левой частью правила, а β – правой частью.

Неформально грамматикой G(S) можно назвать конечное непустое множество правил, где S – символ, который должен встретиться в левой части хотя бы одного правила. Все символы, которые встречаются в левых и правых частях правил, образуют словарь грамматики V.

Пример 2.1.

Рассмотрим следующую грамматику, справа в скобках указан номер правила:

<целое_число> ::=<знак><число> (1)
<целое_число> ::=<число> (2)
<число> ::= <цифра> (3)
<число> ::=<цифра><число> (4)
<цифра> ::= 1 (5)
<цифра> ::= 2 (6)
<цифра> ::= 3 (7)
<цифра> ::= 4 (8)
<цифра> ::= 5 (9)
<цифра> ::= 6 (10)
<цифра> ::= 7 (11)
<цифра> ::= 8 (12)
<цифра> ::= 9 (13)
<цифра> ::= 0 (14)
<знак> ::= – (15)
<знак> ::= + (16)

Будем считать, что слова, расположенные в угловых скобках, рассматриваются как один символ, тогда V = {<целое_число>, <число>, <знак>, <цифра>, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, -,+}. Начальным символом S для этой грамматики является <целое_число>.

С помощью этой грамматики можно получать строки символов, представляющие собой десятичные целые числа, возможно со знаком.

Символы грамматики, заключённые в угловые скобки, являются вспомогательными и называются *нетерминальными* символами. Остальные символы словаря V являются *терминальными*. Множество нетерминальных символов грамматики обозначается VN; множество терминальных символов грамматики обозначается VT. Очевидно, что $VN \cup VT = V$, при этом $VN \cap VT = \emptyset$.

Правило грамматики можно рассматривать как операцию над цепочками. Применение правила грамматики заменяет подцепочку в исходной цепочке, совпадающую с левой частью одного из правил грамматики, на правую часть этого правила. Никакие другие, кроме заданных правилами грамматики, действия над цепочкой недопустимы.

На рис 2.1 рассмотрен пример преобразования цепочек для грамматики из примера 2.1.

Исходная цепочка	Полученная цепочка	№ правила
<целое_число>	**<знак><число>**	1
<знак>**<число>**	<знак>**<цифра><число>**	4
<знак>**<цифра>**<число>	<знак>**5**<число>	9
<знак>5**<число>**	<знак>5**<цифра>**	3
<знак>5<цифра>	**+**5<цифра>	16
+5**<цифра>**	+5**3**	7

Рис. 2.1 Преобразование цепочек для заданной грамматики.

Может возникнуть вопрос: почему в цепочке <знак><цифра><число> было применено правило <цифра> ::= 5 (9), а не правило <знак> ::= + (16). Если правила грамматики задают конечное множество операций над цепочками, и никакие действия, кроме заданных правилами, над цепочками выполнить нельзя, то порядок применения правил никак не детерминирован. Если к цепочке символов можно применить несколько альтернативных правил, то правило выбирается исходя из, например, соображений удобства объяснения. Для некоторых случаев существует ограничение на выбор нетерминального символа в цепочке, который будет раскрываться правой частью правила, см. ниже левосторонний и правосторонний вывод.

Порождающей грамматикой называется четверка, или кортеж

$G = (VT, VN, P, S)$, где

VT – множество терминальных символов грамматики;

VN – множество нетерминальных символов грамматики;

P – непустое конечное множество правил грамматики;

S – начальный(стартовый) символ грамматики, $S \in VN$.

В дальнейшем в качестве нетерминальных символов грамматик будут использоваться заглавные латинские буквы, а в качестве терминальных – строчные буквы, обычно из начала латинского алфавита.

Вместо символа ::= будет использоваться символ →, кроме того, если существует несколько правил с одинаковыми левыми частями, то их альтернативные правые части будут объединяться с помощью символа | (читается как «или»), например, правила $\alpha \to \beta_1$, $\alpha \to \beta_2$, $\alpha \to \beta_3$, … $\alpha \to \beta_n$ сокращенно могут быть записаны следующим образом: $\alpha \to \beta_1 \mid \beta_2 \mid \beta_3 \ldots \mid \beta_n$. В подобных случаях будем говорить, что левой части правила соответствует несколько альтернатив.

Грамматика из примера 2.1, в соответствии с вышесказанным, может быть представлена следующим образом:

<целое_число> → <знак><число> | <число>

<число> → <цифра> | <цифра><число>

<цифра> → 1 | 2 | 3| 4 | 5 | 6 | 7 | 8 | 9 | 0

<знак> → – | +

Гораздо большая компактность определения порождающей грамматики очевидна.

Пусть G – грамматика, тогда говорят, что цепочка α *непосредственно порождает* цепочку β, обозначается $\alpha \Rightarrow \beta$, если для некоторых цепочек $\alpha = \xi_1\gamma\xi_2$ и $\beta = \xi_1\delta\xi_2$, причем $\xi_1,\xi_2 \in (VT \cup VN)^*$, а $\gamma \to \delta$ – правило из множества P грамматики G. Для любого правила грамматики $\gamma \to \delta$ справедливо $\gamma \Rightarrow \delta$.

Говорят, что цепочка α *порождает* цепочку β, если существует конечное количество непосредственных порождений от α к β $\alpha \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \ldots \gamma_n \Rightarrow \beta$. Эта последовательность обозначается $\alpha \Rightarrow^+ \beta$, имеет длину n, причем $n>0$, и называется *выводом* для цепочки β.

Если в процессе вывода цепочки всякий раз раскрывается самый левый нетерминальный символ, это называется *левосторонним*

выводом цепочки; если всякий раз раскрывается самый правый нетерминальный символ, то это *правосторонний* вывод.

Символ ⇒ задает отношение между цепочками, которое мы назвали «непосредственно порождает», тогда отношение $\Rightarrow+$ есть не что иное, как транзитивное замыкание этого отношения.

Для грамматики из примера 2.1

- непосредственный вывод: <знак>5<число> ⇒ +5<число>;
- порождение цепочки: <знак><число> ⇒+ +5<цифра>.

Говорят, что цепочка β *выводима* из цепочки α, обозначается $\alpha \Rightarrow* \beta$, если $\alpha \Rightarrow \beta$ или $\alpha \Rightarrow+ \beta$.

Сентенциальной формой грамматики называется цепочка α такая, что $S \Rightarrow* \alpha$. Иначе говоря α выводима из начального символа грамматики. Цепочка <знак>5<цифра> является сентенциальной формой для грамматики из примера 2.1, т. к. <целое_число> ⇒* <знак>5<цифра>, а цепочка <цифра>+3 не является сентенциальной формой для этой грамматики, поскольку эта цепочка не выводима из начального символа грамматики.

Предложение – это сентенциальная форма, состоящая только из терминальных символов, если $S \Rightarrow* \alpha$, и $\alpha \in VT^*$, то α – предложение языка. Структура предложения задается грамматикой.

Язык, порождаемый грамматикой G, есть подмножество всех цепочек из VT^* и определяется следующим образом:

$L(G) = \{\alpha \in VT^* \mid S \Rightarrow* \alpha\}$.

Несколько различных грамматик могут порождать один и тот же язык. Тогда говорят об эквивалентности этих грамматик. Грамматики эквивалентны, если они порождают один и тот же язык.

На рис. 2.2 приведены три примера грамматик, для каждой из которых определяется язык, порождаемый этой грамматикой.

G1(S)	P = {S → aB B → cD D → k }	VT = {a, c, k} VN = {S, B, D}	L(G1) = {ack}
G2(S)	P = {S → aB B → cB | k }	VT = {a, c, k} VN = {S, B}	L(G2) = $\{ac^nk\}$ n = 0,1,2,3...
G3(S)	P = {S → aB B → cB}	VT = {a, c} VN = {S, B}	L(G3) = ∅

Рис. 2.2 Примеры порождающих грамматик.

Грамматика G_1 порождает язык, состоящий из единственного предложения. Грамматика G_2 порождает бесконечный язык. Важно отметить то, что только грамматики с *рекурсивными правилами* могут порождать бесконечные языки. Рекурсивным называется правило, в котором один и тот же нетерминальный символ встречается и в левой, и в правой части. Допускается косвенная рекурсия.

Грамматика G_3 порождает пустой язык, так как в соответствии со вторым правилом ($B \to cB$) ни одно предложение получить невозможно, или иначе грамматика не порождает ни одной терминальной цепочки.

В силу того, что язык является множеством, можно использовать способы определения множеств для задания языков, указывая сведения о цепочках символов, $L = \{\alpha \mid \text{информация об } \alpha\}$.

Например, $L_1 = \{\alpha \mid \alpha$ содержит одинаковое число нулей и единиц$\}$. Вместо имени цепочки можно указать выражение, зависящее от параметров, и описывать цепочки языка, уточняя эти параметры.

$L_2 = \{1^n0^n \mid n \geq 1\}$ — Язык L_2 есть множество цепочек, в которых n единиц предшествуют n нулям, причем n больше или равно 1, $L_2 = \{10, 1100, 111000, \ldots\}$

$L_3 = \{0^m1^n \mid n \geq m \geq 0\}$ — Язык L_3 состоит из цепочек, в которых нули предшествуют единицам, причем число единиц не меньше числа нулей.

Язык L_2 также может быть определен с помощью порождающей грамматики $G = (VT = \{0,1\}, VN = \{S\}, P = \{S \to 1S0 \mid 10\}, S)$.

Если алфавит состоит из единственного символа, например $\Sigma = \{a\}$, то возможны следующие определения языков:

$L_4 = \{ \alpha \mid \alpha = a^+\}$ — Язык L_3 состоит из цепочек, содержащих одну или больше букв a;

$L_5 = \{ \alpha \mid \alpha = a^*\}$ — Язык L_5 состоит из цепочек, содержащих одну или больше букв a, либо пустую цепочку.

Обозначим через α^R : реверс цепочки α, тогда можно определить следующий язык для алфавита T

$L_6 = \{ \alpha\alpha^R \mid \alpha = VT^+\}$ — Язык L_6 задает палиндромы из словаря VT.

Подобные множественно-теоретические способы определения языка весьма наглядны, однако построение порождающей грамматики по такому представлению языка является, в общем случае,

нетривиальной задачей. Читателю предлагается построить порождающую грамматику для языка L_3.

2.3 Классификация порождающих грамматик Хомского

Н. Хомский (Avram Noam Chomsky) определил четыре основных класса языков в терминах порождающих грамматик: G = (VT, VN, P, S), различие четырех типов грамматик заключается в виде правил подстановки, допустимых в множестве P.

Говорят, что G — это грамматика *типа 0* или неограниченная грамматика, если для этой грамматики правила имеют следующий вид: $\alpha \to \beta$, где $\alpha \in V^+$ и $\beta \in V^*$.

Иначе говоря, левая часть правила является конечной цепочкой из усеченной итерации словаря грамматики, а правая часть принадлежит итерации словаря грамматики V, т. е. может быть пустой.

Грамматики *типа 1*, называемые также *неукорачивающимися* или *контекстно-зависимыми* грамматиками.

Если для всех правил грамматики выполняется следующее соглашение: $\alpha \to \beta$, где $\alpha, \beta \in V^+$ и $1 \leq |\alpha| \leq |\beta|$, то такие грамматики называются неукорачивающимися.

Если для всех правил $\alpha \to \beta \in P$, где $\alpha = \xi_1 N \xi_2$, $\beta = \xi_1 \gamma \xi_2$; $N \in VN^+$, $\gamma \in V^+$; $\xi_1, \xi_2 \in (VT \cup VN)^*$, то такая грамматика называется *контекстно-зависимой*. Термин «контекстно-зависимая» отражает тот факт, что нетерминальный символ N можно заменить на γ только в контексте $\xi_1 \ldots \xi_2$.

Грамматика *типа 2*, или *контекстно-свободная* (КС) – это грамматика все правила которой имеют следующий вид: $A \to \alpha$, где $A \in VN$, $\alpha \in V^+$. Если $\alpha \in V^*$, то такая грамматика является укорачивающейся КС грамматикой.

Грамматика *типа 3*, также называемая автоматной или *регулярной* грамматикой, в зависимости от типа правил может быть праволинейной или леволинейной. Грамматика называется праволинейной, если каждое правило грамматики имеет вид: $A \to t$, или $A \to tB$, где $A, B \in VN$; $t \in VT$. Грамматика называется леволинейной, если каждое правило грамматики имеет вид: $A \to t$, или $A \to Bt$, где $A, B \in VN$; $t \in VT$.

Для заданной грамматики G = (VT, VN, P, S) если правило имеет вид $A \to \varepsilon$, где $A \in VN$, то его принято называть ε-правилом (эпсилон-правило).

Для $i = 0, 1, 2, 3$ грамматика $G = (VT, VN, P, S)$ называется *расширенной грамматикой типа i*, если в множестве P существует хотя бы одно правило вида $A \to \varepsilon$.

Для $i = 0, 1, 2, 3$ грамматика $G = (VT, VN, P, S)$ называется *S-расширенной грамматикой типа i*, если в множестве P существует хотя бы одно правило вида $S \to \varepsilon$.

Язык L(G) включает ε тогда и только тогда, когда $S \Rightarrow* \varepsilon$. Например грамматика G = ({a, b}, {S}, {S → aSb | ε}, S) порождает язык L(G) = {ε, ab, aabb, aaabbb, …}.

2.4 Соотношения языков и грамматик

Соотношения между типами грамматик [9]:

- любая регулярная грамматика является КС-грамматикой;
- любая регулярная грамматика является УКС-грамматикой;
- любая КС-грамматика является КЗ-грамматикой;
- любая КС-грамматика является неукорачивающей грамматикой;
- любая КЗ-грамматика является грамматикой типа 0.
- любая неукорачивающая грамматика является грамматикой типа 0.

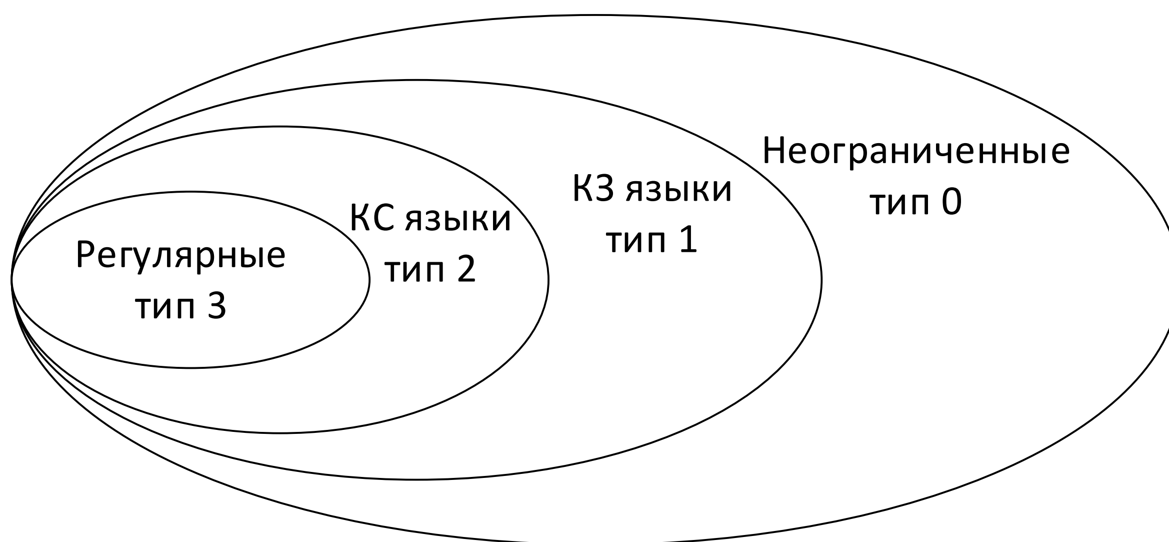


Рис. 2.3. Соотношение между языками в иерархии Хомского.

Язык L(G) называется языком типа k, если он может быть определен грамматикой типа k, причем k – максимально возможный номер типа грамматики. Любой регулярный язык является подмножеством КС языков, но существуют КС языки, которые не являются регулярными. Любой КС язык является КЗ языком, но

существуют КЗ языки, которые не являются КС. Любой КЗ язык является языком типа 0, или иначе неограниченным.

Словари грамматики	Правила грамматики	Порождаемый язык и тип грамматики.
VN = {S, A, C} VN = {a, b, c}	P = { S → aAbc | ε A → aAbC | ε Cb → bC Cc → cc}	L = { $a^n b^n c^n$ | n ≥ 0} Грамматика типа 0.
VN = {S, B, C} VT = {a, b,c}	P = { S → aSBC | aBC CB → BC aB → ab bB → bb bC → bc cC → cc}	L = {$a^n b^n c^n$ | n > 0} Грамматика типа 1.
VN = {S, A, B} VT = {0, 1}	P = { S → 0A | 1B A → 0 | 0S | 1AA B → 1 | 1S | 0BB}	L = {ω | ω ∈ $\{0, 1\}^+$ и число нулей равно числу единиц. } Грамматика типа 2.
VN = {S, A, B} VT = {0, 1}	P = { S → 0A | 1B | 0 | 1 A → 0A | 1B | 0 | 1 B → 0A | 0}	L = {ω | ω ∈ $\{0, 1\}^+$ и не содержит двух рядом стоящих единиц} Грамматика типа 3.

Рис. 2.4. Примеры грамматик и языков в иерархии Хомского.

Грамматика из примера 2.1 является КС грамматикой, при этом она порождает регулярный язык целых чисел со знаком.

Читателю предлагается самостоятельно доказать совпадают ли типы порождающих грамматик на рис. 2.4. с типами порожденных ими языков.

2.5 Распознаватели для формальных языков

Пусть G = (VT, VN, P, S) – грамматика, а α – цепочка терминальных символов ($\alpha \in VT^+$), тогда есть задача определения: принадлежит ли цепочка α языку, порождаемому грамматикой G.

Распознаватель для языка – это алгоритм или программа, позволяющая решить эту задачу. На вход распознавателя поступает цепочка символов, результатом работы является ответ «Да», если цепочка принадлежит языку, порождённому некоторой грамматикой (*допуск цепочки*), или ответ «Нет» (*цепочка отвергнута*).

Схематично и очень условно распознаватель можно представить так, как это показано на рис. 2.5. Распознаватель состоит из следующих частей:

- ленты, содержащей цепочку входных символов, предназначенную для процесса распознавания, и считывающей головки, обозревающей очередной символ в этой цепочке;
- устройства управления (УУ), которое координирует работу распознавателя, имеет конечное множество состояний и ограниченную память для хранения текущего состояния и некоторой служебной информации;
- внешней (рабочей) памяти (ВП), которая хранит промежуточные данные, полученные в процессе работы распознавателя, и в отличие от памяти УУ может иметь неограниченный объем.

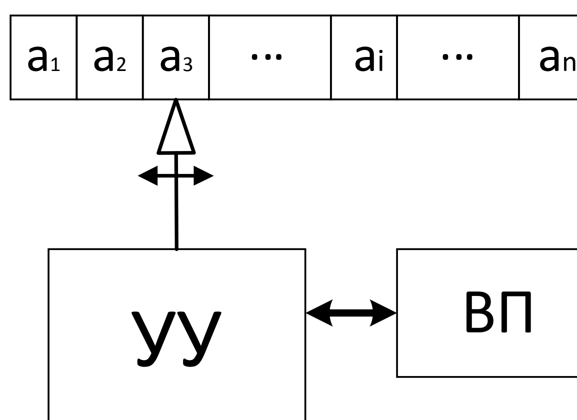


Рис. 2.5. Структурная схема распознавателя.

Распознаватель получает на вход цепочку из символов алфавита распознавателя. Алфавит распознавателя конечен. Он включает в себя все допустимые символы входных цепочек (терминальные символы грамматики), а также некоторый дополнительный алфавит символов, которые могут обрабатываться УУ и храниться в рабочей памяти распознавателя.

В процессе своей работы распознаватель может выполнять элементарные операции, такие как чтение очередного символа из входной цепочки, сдвиг входной цепочки на заданное количество символов (вправо или влево), доступ к рабочей памяти для чтения или записи информации, преобразование информации в рабочей памяти, изменение состояния УУ. То, какие конкретно операции должны выполняться в процессе работы распознавателя, определяется в УУ.

Распознаватель является синхронным устройством, т. е. на один шаг работы алгоритма, который он реализует отводится один такт. Вся работа распознавателя состоит из *последовательности тактов*. В начале каждого такта состояние распознавателя определяется его конфигурацией. В процессе работы конфигурация распознавателя меняется.

Конфигурация распознавателя определяется следующими параметрами:

- содержимое входной цепочки символов и положение считывающей головки в ней;
- состояние УУ;
- содержимое внешней памяти.

Для распознавателя задается определенная конфигурация, которая считается *начальной*. В начальной конфигурации считывающая головка обозревает первый символ входной цепочки, УУ находится в заранее определенном исходном положении, а внешняя память либо пуста, либо содержит строго определенную информацию.

Кроме начальной конфигурации, для распознавателя задается одна или несколько конечных конфигураций. В конечной конфигурации считывающая головка, как правило, находится за концом входной цепочки. Для демонстрации этого обычно вводят специальный символ, называемый концевой маркер, обозначающий конец входной цепочки, например #, который расположен правее самого правого символа входной цепочки.

Распознаватель *допускает* входную цепочку символов α, если, начав работу из начальной конфигурации, он через конечную последовательность шагов переходит в одну из конечных конфигураций. В противном случае цепочка *отвергается* распознавателем.

Язык, допускаемый распознавателем, — это множество всех цепочек, которые допускает распознаватель. В этом смысле можно говорить о том, что распознаватель является способом определения языка, альтернативным порождающим грамматикам, и может быть назван распознающей грамматикой. Тем не менее порождающая грамматика намного более наглядно демонстрирует свойства языка, в отличие от черного ящика (распознаватель), допускающего или отвергающего цепочки символов.

По видам считывающего устройства распознаватели могут быть *двунаправленные* и *однонаправленные*. Первые допускают возврат считывающей головки.

По видам устройства управления распознаватели бывают детерминированные и недетерминированные. Распознаватель называется *детерминированным* в том случае, если для каждой допустимой конфигурации распознавателя, которая возникла на некотором шаге его работы, существует единственно возможная конфигурация, в которую распознаватель перейдет на следующем шаге работы. В противном случае распознаватель называется *недетерминированным*.

По видам внешней памяти распознаватели бывают следующих типов:

- без внешней памяти;
- с ограниченной внешней памятью;
- с неограниченной внешней памятью.

Соотношение между классами грамматик для иерархии Хомского и классами распознавателей [12] представлено на рис. 2.6.

Уровень в иерархии	Класс грамматик	Класс распознавателя
0	Неограниченные	Машина Тьюринга
1	Контекстно-зависимые	Линейно-ограниченный автомат
2	Контекстно-свободные	Автомат с магазинной памятью
3	Регулярные	Конечный автомат

Рис. 2.6. Классы грамматик и абстрактных машин.

Грамматики и распознаватели — два независимых метода, которые могут быть использованы для определения языка. Третьим способом задания языка является множественно-теоретический способ из п. 2.2.

Порождающая грамматика более информативна, так как позволяет сделать выводы о порождающем языке, хотя бы в общем виде, например рис 2.2 – см. множественно-теоретический подход. Распознаватель представляет собой черный ящик, относительно которого известно только множество входных символов, и понять по его реакции на допуск или недопуск цепочек из множества входных символов, какой язык он допускает – задача нетривиальная, особенно для бесконечных языков.

Разбор (допуск цепочки терминальных символов) предполагает построение распознавателя по заданной грамматике, которая порождает язык, допускаемый сконструированным распознавателем.

Контрольные вопросы

1. Дайте определение усеченной итерации алфавита.
2. Из чего состоит правило грамматики.
3. Дайте определение порождающей грамматики.
4. Чем отличаются терминальные и нетерминальные символы?
5. Перечислите способы задания формальных языков.
6. Что лежит в основе классификации грамматик по Хомскому?
7. Как соотносятся типы порождающих грамматик и языков?
8. Дайте определение распознавателю для формальных языков.
9. Что понимается под конфигурацией распознавателя?
10. Что понимается под «рекурсивным правилом» грамматики?

3 Регулярные языки и конечные автоматы

Регулярные языки относятся к хорошо изученному классу языков, для которых существует мощный математический аппарат. Кроме того, регулярные языки нашли широкое применения в конструировании лексических анализаторов для компиляторов языков программирования [1, 3, 13].

В данном разделе будут обсуждаться вопросы задания регулярных языков с помощью регулярных грамматик, конечных автоматов и регулярных множеств, а также вопросы построения распознавателей для регулярных языков. Регулярные грамматики обсуждались в п. 2.3.

3.1 Конечные автоматы

Абстрактный дискретный исполнитель, преобразующий последовательность входных символов в последовательность выходных символов так, что значение символа на выходе исполнителя зависит не только от значения входного символа, но и от предыдущей последовательности входных символов (предыстории), принято называть *автоматом*. Для хранения текущей предыстории используется внутренняя память автомата, причем каждой предыстории соответствует определенное состояние памяти.

Если количество предысторий бесконечно, то отсутствует возможность программной или аппаратной реализации такого автомата.

На бесконечном счетном множестве предысторий можно ввести отношение эквивалентности. Будем считать, что любые две предыстории эквивалентны, если для каждой из них любая последовательность входных символов будет приводить к одинаковой последовательности выходных символов. Если количество эквивалентных предысторий конечно, то соответствующий им автомат будем называть *конечным автоматом*.

Обозначим через X алфавит входных символов таких, что $x_i \in X$, а через Y – алфавит выходных символов $y_i \in Y$. В общем случае алфавиты входных и выходных символов автомата могут совпадать.

Каждому подмножеству эквивалентных предысторий в множестве предысторий соответствует уникальное *состояние автомата*. Набор всех возможных состояний конечного автомата называется *множеством состояний* S. Будем считать, что процесс

преобразования входных символов в выходные, а именно: поступление входного символа, переход автомата из одного состояния в другое и выдача выходного символа, происходит не непрерывно, а мгновенно в некоторые моменты времени, называемые *тактами*.

Состояние $s_i \in S$ конечного автомата определяется состоянием автомата на предыдущем такте и значением входного символа на данном такте. Конечный автомат хранит текущее состояние во внутренней памяти. Будем считать, что в каждый момент времени t = 1, 2, 3,… на вход автомата поступает символ из множества X, а на выходе формируется символ из выходного множества Y. Тогда входной символ в момент времени t обозначим как x(t), а выходной символ y(t), а в момент времени t+1 как x(t+1) и y(t+1), соответственно. Состояние автомата в момент времени t обозначим как s(t). Структурная схема конечного автомата (КА) с одним входом и одним выходом представлена на рис. 3.1.

Рис. 3.1. Структурная схема абстрактного конечного автомата

Конечный автомат формально определяется как шестерка, или кортеж, следующим образом:

$A = (X, Y, S, s_0, \delta, \lambda)$, где:

X – конечное непустое множество входных сигналов (входной алфавит);

Y – конечное непустое множество выходных сигналов (выходной алфавит);

S – конечное непустое множество состояний;

s_0 – начальное состояние автомата, $s_0 \in S$;

$\delta: S \times X \to S$ – функция переходов;

$\lambda: S \times X \to Y$ – функция выходов.

Автомат называется *детерминированным*, если для любого состояния s_i и входного символа x_k переход $\delta(s_i, x_k) = s_j$ является единственным. Описанный КА является *преобразователем*, так как он по входной цепочке символов входного алфавита строит выходную цепочку из символов выходного алфавита. Все состояния КА преобразователя семантически одинаковы.

3.1.1 Способы задания конечного автомата

Для иллюстрации способов задания (представления) конечных автоматов рассмотрим следующий пример.

Пример 3.1.

Пусть символам A, B, C, D соответствуют неравномерные двоичные коды 0, 10, 110 и 111, соответственно. Необходимо построить конечный автомат, принимающий последовательность входных кодов и преобразующий их в соответствующие символы. Для удобства будем считать, что ошибок во входной последовательности нет. Неформально такой автомат можно определить следующим образом.

Для этого автомата X = {0, 1}, Y = {A, B, C, D, -}. У автомата будет три состояния: начальное, одно состояние для определения ведущей единицы, с которой начинается код символа B, и одно для распознавания последующей единицы для кодов символов C и D.

Допустим, что автомат находится в начальном состоянии и входным символом является «1». Принять решение о выходном символе можно будет только после поступления следующего входного символа. Тогда на некоторых тактах своей работы автомат будет генерировать выходной символ – (минус), который интерпретируется как служебный. После того, как автомат определил очередной выходной символа по входной последовательности символов, он должен переходить в начальное состояние. Задать КА можно с помощью диаграмм перехода или с помощью таблиц функций переходов и выходов.

Диаграмма переходов автомата является направленным помеченным графом, число вершин которого совпадает с количеством состояний автомата. Специально выделяется начальная вершина, см. вершину S_1 на рис. 3.2.

Если из состояния s_i существует переход в состояние s_j по входному символу x_k, при этом выходным символом является y_m, то этому будет соответствовать направленная дуга из s_i в s_j, помеченная x_k/y_m. Если переход из s_i в s_j возможен по различным символам, то вместо различных дуг обычно изображают одну, помеченную разными входными и выходными символами, перечисленными через запятую. На рис 3.2 показана диаграмма переходов для автомата, преобразующего входную последовательность двоичных кодов в буквы из примера 3.1.

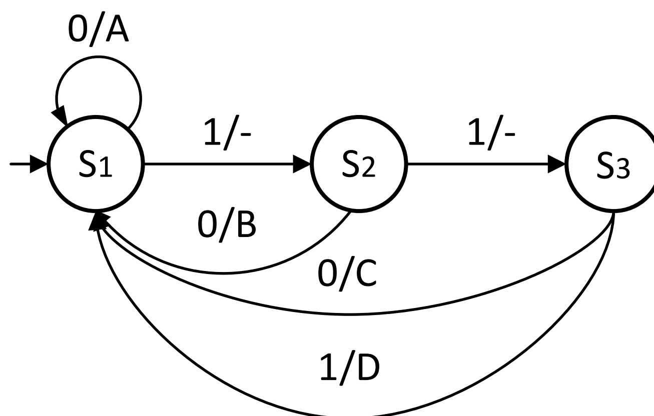


Рис. 3.2 Диаграмма переходов конечного автомата из примера 3.1.

Альтернативным способом задания конечного автомата являются *таблицы функций переходов и выходов*, показанных на рис. 3.3. В соответствии с определением из п. 3.1. данный КА является автоматом-преобразователем.

δ	0	1
S_1	S_1	S_2
S_2	S_1	S_3
S_3	S_1	S_1

a) Функция переходов

λ	0	1
S_1	A	-
S_2	B	-
S_3	C	D

b) Функция выходов

Рис. 3.3. Таблицы функций переходов и выходов для автомата из примера 3.1.

3.1.2 Автомат-распознаватель

Конечный автомат-распознаватель формально определяется как пятерка, или кортеж, следующим образом:

$A = (X, S, s_0, \delta, F)$, где:

X – конечное непустое множество входных сигналов (входной алфавит);

S – конечное непустое множество состояний;

s_0 – начальное состояние автомата, $s_0 \in S$;

$\delta: S \times X \to S$ – функция переходов;

F – множество допускающих (заключительных) состояний, $F \subseteq S$.

Определенный таким образом конечный автомат является детерминированным (ДКА). В дальнейшем изложении под конечным автоматом-распознавателем будет пониматься именно ДКА распознаватель, если это не будет оговорен иначе. В общем случае $s_0 \in F$.

Пусть G = (VT, VN, P, S) – грамматика, а α – цепочка терминальных символов ($\alpha \in VT^+$), тогда ДКА должен определить принадлежит ли α языку L(G). Начиная с первого символа цепочки α, ДКА читает символы входной цепочки и переходит из состояния в состояние в соответствии с функцией переходов. Если после прочтения всей цепочки ДКА оказался в одном из допускающих состояний, то цепочка α допущена, и $\alpha \in L(G)$. Таким образом, для определения принадлежности цепочки символов языку нет необходимости формирования выходных символов. Поэтому у автомата-распознавателя отсутствует функция выходов, а сами автоматы-распознаватели называют автоматами без выходов. Если для текущего состояния s и текущего символа t из входной цепочки не существует δ(s, t), или после прочтения всей цепочки ДКА не перешел в одно из заключительных состояний, то цепочка α отвергается ДКА.

Под конфигурацией ДКА понимается двойка вида (s, ω), где s – текущее состояние автомата, цепочка ω – необработанная часть входной ленты, включая самый левый символ, находящийся под головкой чтения.

Начальная конфигурация ДКА определяется двойкой (s_0, ω). На каждом такте работы ДКА в соответствии с функцией переходов осуществляется смена конфигурации. Введем отношение непосредственной смены конфигураций, обозначаемое ⊢, пусть

$\delta(s_i, t) = s_j$, причем s_i, $s_j \in S$, $t \in X$, тогда для всех цепочек $\omega \in X^*$ справедливо следующее отношение: $(s_i, t\omega,) \vdash (s_j, \omega)$.

Под этим понимается следующее: если ДКА находится в состоянии s_i, причем t – это текущий символ на входной ленте, то осуществляется передвижение считывающей головки к следующему символу входной ленты, и ДКА переходит в состояние s_j.

Конечной конфигурацией ДКА является двойка (s, ε) где $s \in F$. Обозначим $\vdash *$ последовательность из ноля или более переходов между состояниями ДКА.

Говорят, что ДКА допускает цепочку входящих символов α, если существует путь по конфигурациям $(s_0, \alpha) \vdash * (s, \varepsilon)$, для некоторых $s \in F$. Множество, возможно бесконечное, допустимых цепочек является языком, допускаемым ДКА.

Два ДКА являются эквивалентными тогда и только тогда, когда они допускают один и тот же язык.

Пример 3.2. Пусть задан язык $L_1 = \{\alpha \mid \alpha = \{a, b\}^+$ и α включает подцепочку $bb\}$. На рисунке 3.4 показана диаграмма переходов ДКА, допускающего язык L_1.

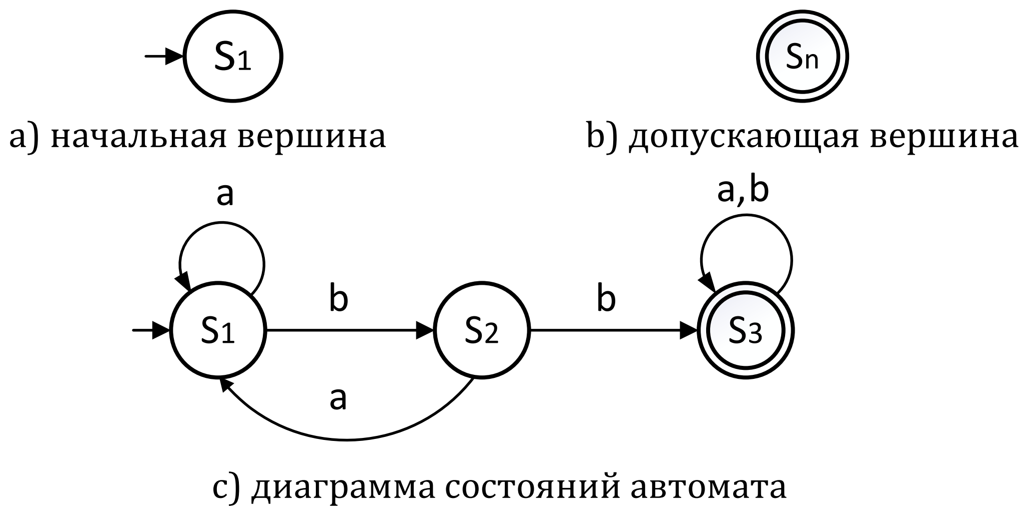


Рис. 3.4. Диаграмма переходов ДКА допускающий язык L_1

3.1.3 Недетерминированный конечный автомат

Недетерминированный конечный автомат формально определяется как пятерка, или кортеж, следующим образом:

$A = (X, S, s_0, \delta, F)$, где:

X – конечное непустое множество входных сигналов (входной алфавит);

S – конечное непустое множество состояний;

s_0 – начальное состояние автомата, $s_0 \in S$;

$\delta: S \times X \rightarrow 2^S$ – функция переходов;

F – множество допускающих (заключительных) состояний, $F \subseteq S$.

В отличие от ДКА, функция переходов НКА является отображением пар (состояние, входной символ) из $S \times X$ в множество всех подмножеств множества состояний НКА. Для диаграмм состояний это соответствует нескольким альтернативным переходам из одного состояния в несколько по одному и тому же входному символу. Понятие конфигурации и языка, допускаемого НКА, совпадает с соответствующими определениями для ДКА с необходимостью учета специфики функции перехода для НКА.

Соотношение между ДКА и НКА может быть парадоксально определено следующим образом: любой детерминированный КА является недетерминированным КА. Это можно легко доказать, исходя из того, что функция переходов ДКА осуществляет переход из текущего состояния по входному символу в одно состояние, а функция переходов НКА – в ноль, одно или несколько состояний.

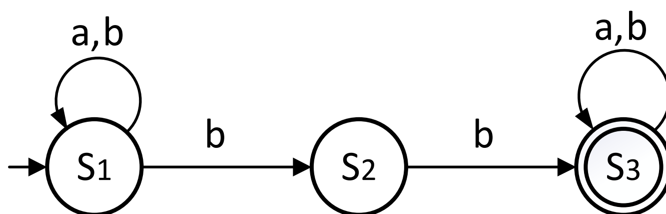


Рис. 3.5. НКА допускающий язык L1

На рисунке 3.5 показан НКА, допускающий язык из примера 3.2.

Если функцию переходов НКА изменить следующим образом: $\delta: S \times (X \cup \{\varepsilon\}) \rightarrow 2^S$, то будет получен НКА, допускающий *ε-переходы*. ε - переходу соответствует переход между состояниями НКА без чтения входного символа; ε-переходы часто используют для построения сложных (составных) НКА из простых, как это будет показано в 3.5.

Из теоремы Рабина-Скотта [15] следует, что класс языков, определяемых недетерминированными конечными автоматами, совпадает с классом языков, определяемых детерминированными конечными автоматами. Это означает, что для любого НКА всегда можно построить ДКА, допускающий тот же язык, причем каждое состояние ДКА будет подмножеством 2^S, где S – множество состояний НКА, а значит, что в худшем случае ДКА, построенный по заданному НКА, будет содержать 2^S состояний.

3.1.4 От регулярной грамматики к НКА и обратно

Пусть G = (VT, VN, P, S) – праволинейная регулярная грамматика, тогда НКА, допускающий тот же язык, который порождает данная грамматика, может быть построен с помощью следующего алгоритма.

Алгоритм 3.1 Построение НКА по праволинейной РГ.
Вход G = (VT, VN, P, S)
Выход NFA = A = (X, S, s_0, δ, F)

Шаг 1 S = VN; X = VT; F = ∅; δ = ∅;
Шаг 2 **foreach** p in P **{**
  if (p = A → aB **)** добавить B в δ(A, a)
  if (p = A → a**) {**
    создать новое состояние s; S = S ∪ {s}; F = F ∪ {s};
    добавить s в δ(A, a)**}**
  if (p = S → ε**)**
    F = F ∪ S
}

Пусть G = (VT, VN, P, S) – леволинейная регулярная грамматика, тогда НКА, допускающий тот же язык, который порождает данная грамматика, может быть построен с помощью алгоритма 3.2.

Полученный в результате работы этого алгоритма размеченный мультиграф представляет искомый недетерминированный конечный автомат. Этот НКА может иметь эквивалентные состояния. Удалению эквивалентных состояний соответствует процесс минимизации НКА.

Алгоритм 3.2 Построение НКА по леволинейной РГ.
Вход G = (VT, VN, P, S)
Выход NFA = A = (X, S, s_0, δ, F)

Шаг 1.	S = VN; X = VT; F = ∅; δ = ∅;
Шаг 2.	Единственным конечным состоянием недетерминированного конечного автомата NFA является состояние S
Шаг 3.	Начальное состояние NFA равно S, если S → ε ∈ P, иначе вводится новое состояние s_0.
Шаг 4.	Для каждого правила из P вида A → a вводится ребро, помеченное a, из вершины s_0 к вершине A.
Шаг 5.	Для каждого правила из P вида A → Ba вводится ребро, помеченное a, из вершины B к вершине A.

Ниже представлен алгоритм, формирующий порождающую регулярную грамматику по заданному НКА.

Алгоритм 3.3 Построение регулярной грамматики по НКА
Вход NFA = (X, S, s_0, δ, F)
Выход G = (VT, VN, P, s_0)

Шаг 1 P = ∅;
Шаг 2

```
for всех состояний A и B из S, и всех a из X {
    P.Add( A → aB); VT.Add(a); VN.Add(A); VT.Add(B)
    if( B ∈ F ) P.Add( A → a)
}
```

Полученная в результате этого алгоритма грамматика может иметь бесполезные символы. Вопрос удаления бесполезных символов рассматривается в п. 4.4.

3.2 Минимизация ДКА

Введем понятие расширенной функции переходов для ДКА. Пусть A = (X, S, s_0, δ, F) – ДКА с функцией переходов δ: S × X → S, тогда расширенной функцией переходов будет δ*: S × X* → S, эта функция определена на множестве входных цепочек, возможно, включая и пустую, а не на множестве входных символов:

- δ*(s, ε) = s;
- δ*(s, aα) = δ*(δ (s, a), α).

Состояние $s \in S$ называется *достижимым* тогда и только тогда, когда $(\exists\ \alpha \in X^*)\ \delta^*(s_0, \alpha) = s$, существует хотя бы одна цепочка из X^*, такая, что из начальной конфигурации есть путь в состояние s. В противном случае состояние называется *недостижимым*.

Пусть $A = (X, S, s_0, \delta, F)$ – ДКА, тогда два состояния s_i и s_j $(s_i, s_j \in S)$ называются *эквивалентными*, если $\delta^*(s_i, \alpha) \in F$ и $\delta^*(s_j, \alpha) \in F$ для всех $\alpha \in X^*$.

Два состояния ДКА s_1 и s_2 будем называть k-эквивалентными, если $(\forall\alpha \in X^*, |\ \alpha\ | \leq k)\ \delta^*(s_1, \alpha) = \delta^*(s_2, \alpha)$.

Недостижимые и эквивалентные состояния могут быть получены в процессе формальных действий, связанных с конструированием КА, к ним относятся построение КА по регулярному выражению, прямое произведение автоматов [6], и др.

ДКА без недостижимых и эквивалентных состояний является *минимальным ДКА*.

Алгоритм 3.4 Минимизация ДКА (теорема об эквивалентности) Вход $DFA = (X, S, s_0, \delta, F)$ Выход $DFA = (X, S', s'_0, \delta', F')$	
Шаг 1.	Все состояния из S делятся на два класса эквивалентности. В первый включаются все допускающие состояния F, во второй все не допускающие S - F. Обозначим это разбиение P_0. k = 0;
Шаг 2.	k += 1; На очередном шаге построения разбиения P_k в классы эквивалентности включить те состояния, которые по одинаковым входным цепочкам длины k переходят в k-1 эквивалентные состояния. $P_k = \{r_i\ (k) : \{s_{ij} \in S: \forall\ x \in X\ \delta(s_{ij}, x) \subseteq r_j\ (n-1)\}\ \forall\ i, j \in \mathbb{N}\ \}$
Шаг 3.	Если $P_k \neq P_{k-1}$, то перейти к шагу 2.
Шаг 4.	Объединить полученные на k-м шаге классы эквивалентности в состояния минимизированного автомата. Определить эквивалентный минимальный ДКА в новых обозначениях $Min_{DFA} = (X, S', s'_0, \delta', F')$.

Рассмотрим минимизацию ДКА (рис. 3.6), допускающего язык, $L_2 = \{\ \alpha\ |\ \alpha = (a(ab)^m b)^k\ \ |\ m, k \geq 0\}$.

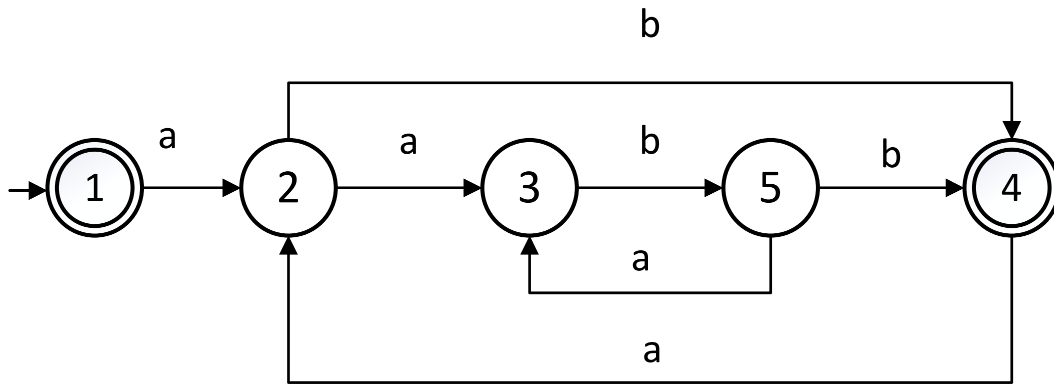


Рис. 3.6 Диаграмма переходов ДКА

Начальное разбиение P_0 = {A_0 = <2, 3, 5>, B_0 = <1, 4>}. Разбиение P_1 включает в себя те классы эквивалентных состояний, которые невозможно отличить при подаче на вход ДКА цепочек длиной в один символ. Состояния 1 и 4 остаются в одном классе эквивалентности, а класс эквивалентности A_0 разделяется на два класса: для состояний 2 и 5 и для состояния 3. P_1 = {A_1 = <2, 5>, B_1= <3>, C_1 = <1, 4>}

На следующем шаге разбиение P_2 = {A_2 = <2, 5>, B_2= <3>, C_2 = <1, 4>}, P_1 совпадает с разбиением P_2, и это значит, что получен минимальный ДКА. Процесс построения разбиений показан на рис. 3.7, а минимальный ДКА – на рис. 3.8.

			P_0		P_1		P_2	
δ	a	b	a	b	a	b	a	b
1	2	–	A_0	–	A_1	–	A_2	–
2	3	4	A_0	B_0	B_1	C_1	B_2	C_2
3	–	5	–	A_0	–	A_1	–	A_2
4	2	–	A_0	–	A_1	–	A_2	–
5	3	4	A_0	B_0	B_1	C_1	B_2	C_2

Рис 3.7 Разбиения классов эквивалентных состояний

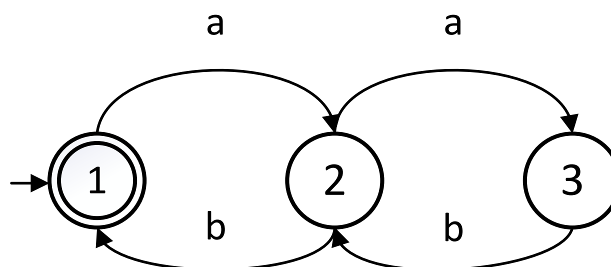


Рис. 3.8 Диаграмма переходов минимального ДКА

3.3 Регулярные выражения

Регулярное множество является регулярным языком. Язык – это множество цепочек символов из некоторого словаря. Для порождающих грамматик таким словарем является словарь терминальных символов. Пусть T – словарь терминальных символов. Тогда регулярное множество (регулярный язык) может быть задано следующим образом:

1. $\emptyset$ (пустое множество) – регулярное множество для T;
2. {ε} – регулярное множество для T (ε– пустая цепочка);
3. {a} – регулярное множество для T, причем $a \in T$;
4. если P и Q – регулярные множества, то регулярными являются и множества
   a. $P \cup Q$ (объединение),
   b. PQ (конкатенация),
   c. P^* (итерация);
5. ничто другое не является регулярным множеством для T.

Регулярное множество может быть либо пусто, либо содержать множество всевозможных комбинаций символов словаря T, возможно, включая и пустую цепочку.

Регулярное выражение – это способ определения регулярного языка, альтернативный способу, использующему порождающие грамматики, при этом используется только словарь терминальных символов. В этом случае словарь нетерминальных символов и множество правил порождающей грамматики не используются, как, собственно, и сама порождающая грамматика.

Регулярное выражение это:

1. $\emptyset$ – регулярное выражение, обозначающее множество $\emptyset$;
2. ε – регулярное выражение, обозначающее множество {ε};
3. a – регулярное выражение, обозначающее множество {a};
4. если p и q – регулярные выражения, соответствующие регулярным множествам P и Q соответственно, то
   a. p|q – регулярное выражение, обозначающее регулярное множество $P \cup Q$;
   b. pq – регулярное выражение, обозначающее регулярное множество $PQ = \{xy \mid x \in P, y \in Q\}$;
   c. p^* – регулярное выражение, обозначающее регулярное множество P^* (замыкание Клини);

d. (p) – регулярное выражение, регулярное множество (p), т. е. регулярное выражение допустимо заключать в скобки без изменения определяемого регулярного множества.

Операции итерации, конкатенации и объединения имеют приоритеты, причем приоритет итерации высший, а объединения – низший. Обычно скобки будут опускаться везде, где их отсутствие не влияет на определение регулярного множества. Регулярное выражение ((a)(b*))|(c) может быть записано следующим образом: ab*|c.

Регулярное выражение (abc) задает регулярное множество из единственного элемента {abc}, а регулярное выражение (a | b | c) задает регулярное множество из трех элементов {a, b, c}. Итерация регулярного выражения задает бесконечное регулярное множество, например $a^* = \{\varepsilon, a, aa, aaa, aaaa, aaaaa, \ldots\}$.

В данном пособии в основном будут использоваться базовые операции для объединения, конкатенации и замыкания, предложенные Стивеном Клини (Stephen Cole Kleene) для регулярных выражений, однако для удобства работы с ними будут также использоваться символы + или ?.

Символ + используется для определения регулярного выражения, повторяющегося один или более раз. В этом смысле $p^+ = pp^*$. Символ ? используется для указания того, что регулярное выражение встречается ноль или один раз, тогда $p? = \varepsilon|p$.

Читатель может проверить, что ДКА на рис. 3.4 допускает язык, заданный регулярным выражением (a|b)*bb(a|b)*.

С. Клини была доказана взаимосвязь между регулярными языками и конечными автоматами.

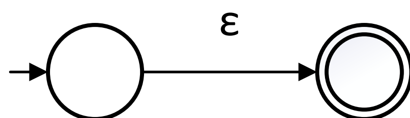
Теорема Клини. ДКА с входным алфавитом X допускает язык L тогда и только тогда, когда L является регулярным множеством для алфавита X. Эта фундаментальная теорема для теории автоматов. [16]

Таким образом регулярные грамматики, конечные автоматы и регулярные выражения являются равномощными средствами определения регулярных языков, и значит, что существуют любые их взаимные преобразования на уровне алгоритмов [15, стр. 109].

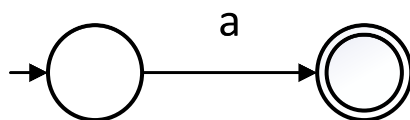
3.4 Построение НКА по регулярному выражению

Из теоремы Клини следует эквивалентность регулярных выражений и конечных автоматов. Ниже будет рассмотрен алгоритм Мак-Нотона – Ямады – Томпсона (McNaughton – Yamada – Thompson), позволяющий преобразовать произвольное регулярное выражение в НКА с ε-переходами. В ходе работы алгоритма исходное регулярное выражение разбивается на подвыражения, для каждого из которых строится соответствующий ему НКА, после чего полученные НКА объединяются. Подробнее алгоритм определен в [1, 15].

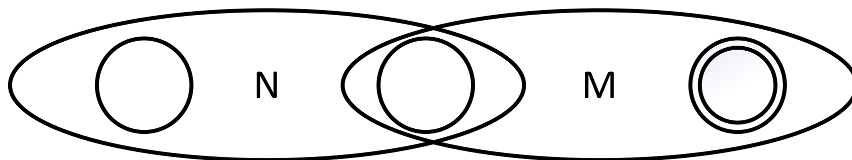
На рис. 3.9 показаны базисные правила для обработки подвыражений.



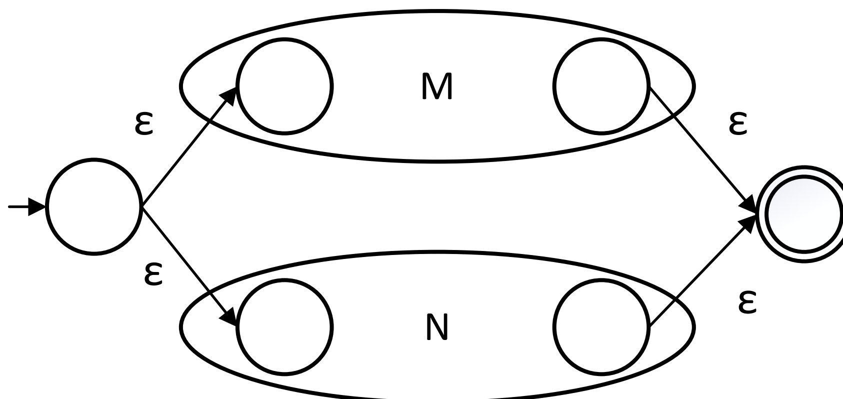
a) НКА для регулярного выражения ε



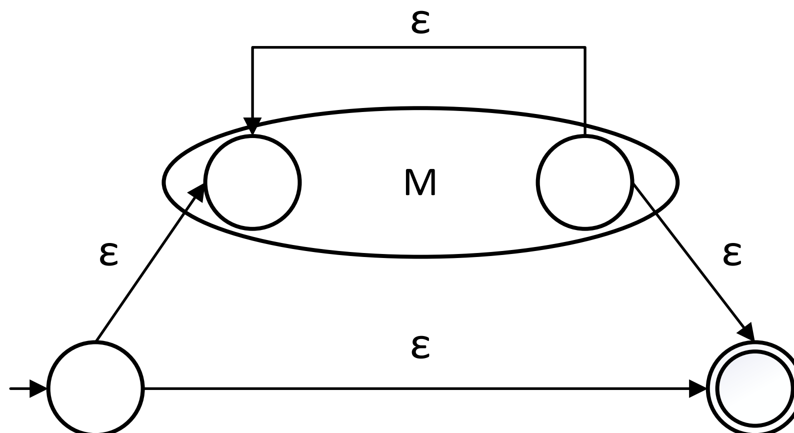
b) НКА для регулярного выражения a



c) НКА для регулярного выражения pq, где p и q – регулярные выражения, а N и M – НКА для них.



d) НКА для регулярного выражения p|q, где p и q – регулярные выражения, а N и M – НКА для них.



е) НКА для регулярного выражения p*, где p – регулярное выражения, M – НКА для него.

Рис 3.9. Правила преобразования базисных регулярных выражений в НКА.

3.5 Построение ДКА по НКА с ε-переходами

По существующему НКА с ε-переходами можно построить соответствующий ему ДКА. Каждому состоянию ДКА будет соответствовать некоторое множество состояний исходного НКА с ε - переходами. В общем случае, если S – множество состояний исходного НКА, то количество состояний получаемого ДКА может составлять 2^S, что может привести к сложностям с его реализацией.

Пусть N – НКА с ε-переходами, причем N = $<X_N, S_N, s_0, \delta_N, F_N>$. Введем следующие обозначения: s – состояние автомата N, $s \in S_N$; T – множество состояний автомата N, такое, что $T \subseteq S_N$.

Определим операции над множеством состояний НКА:

ε-closure(s)	Множество состояний НКА, достижимых из s по ε - переходам (эпсилон-замыкание).
ε-closure(T)	Множество состояний НКА, достижимых из $s \in T$ по ε - переходами. ε-closure(T) = $\cup_{s \in T}$ ε-closure(s)
move (T, a)	Множество состояний НКА, в которые существуют переходы из состояний $s_i \in T$ по входному символу a.

ε-замыкание рекурсивно определяется следующим образом:

- $s_i \in$ ε-closure(s_i);
- Шаг рекурсии. Пусть s_j является элементом ε-closure(s_i). Если $s_k \in \delta(s_j, \varepsilon)$, то $s_k \in$ ε-closure(s_i).
- Замыкание: $s_j \in$ ε-closure(s_i), только если оно достижимо из состояния s_i за конечное число применения шага рекурсии.

Конструирование множества ε-closure(T) удобно осуществлять с использованием стека.

Алгоритм 3.5

Вход: T– множество состояний НКА N

Выход: Множество состояний НКА, достижимых из s по ε - переходами.   ε-closure(T) = T;

```
foreach si in T Stack.push(si)
    while (!Stack.isEmpty()){
        t = Stack.pop();
        foreach ( ui in Q = {u | u ∈ SN &  ∃δ(t,  ε) = u})
        if (ui  ∉ ε-closure(T))     {
        ε-closure(T) = ε-closure(T) + ui; Stack.push(ui);}
}
```

Неформально процесс построения таблицы переходов ДКА можно описать следующим образом:

1. Получаем начальное состояние s ДКА как ε-замыкание начального состояния НКА (S_{0DFA} = ε-closure(S_{0NFA}));
2. Создаем пустое множество M. Для каждого символа t, по которому есть переход из s, строим множество Move(s, t) и добавляем к M.
3. Строим множество K = ε-closure(M)
4. Если K отсутствует в таблице переходов ДКА, помещаем это множество в таблицу в качестве нового состояния.
5. Повторяем шаги 2-4 до тех пор, пока появляются новые состояния.

Для конструирования таблицы переходов ДКА введем операции mark и unmark над его состояниями, которые помечают состояние в таблице переходов TransTableD, или, соответственно, снимают отметку.

Алгоритм 3.6

Вход НКА N

Выход таблица переходов ДКА D эквивалентного НКА N.

```
State state = new ε-closure(s0);
unmark(state);
TransTableD.AddState(state);
```

```
while(TransTableD.unmarkedState=true)
{
	State P = TransTableD.GetUnmarkedState();
	mark(P);
	foreach( a in X_N)
	{
		State Q = ε-closure(move(P, a));
		if (! TTD.IsIn(Q))
			unmark(Q);
		TTD.AddState(Q);
		TTD[P,a] = Q;
	}
}
```

Рассмотрим регулярное выражение (a(a|b)*b)*. Требуется построить минимальный ДКА, распознающий цепочки символов, принадлежащих языку, определяемому этим регулярным выражением. НКА, соответствующий этому регулярному выражению, показан на рис. 3.10.

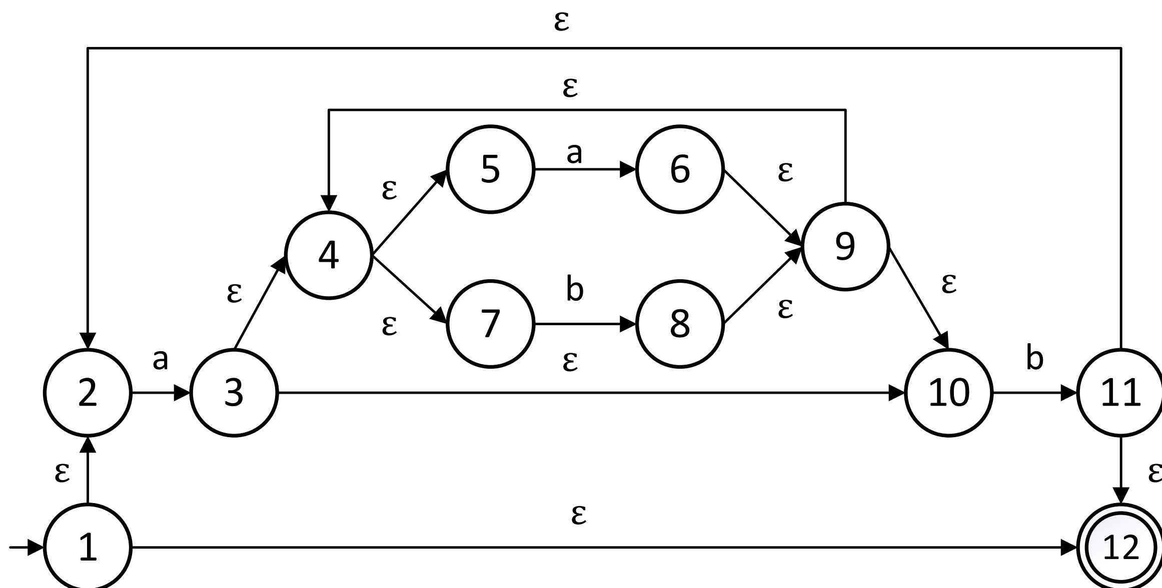


Рис. 3.10 НКА с ε-переходами для регулярного выражения (a(a|b)*b)*

Следующим шагом является построение ДКА, эквивалентного НКА на рис. 3.11. Ищем множество состояний исходного НКА, соответствующее начальному состоянию конструируемого ДКА, для этого строим ε-closure(1) = {1, 2, 12}. В ε-замыкание вошло состояние 12, значит полученное начальное состояние ДКА является допускающим.

Далее заполняется таблица переходов формируемого ДКА (алгоритм 3.6). Для множества состояний НКА, соответствующего состоянию ДКА, для каждого из символов входного алфавита строится множество состояний посредством функции Move(T, a), после чего осуществляется ε-замыкание объединенного множества.

Полученное в результате множество состояний НКА добавляется в качестве нового состояния ДКА в таблицу переходов, если оно не было получено ранее. Рассмотрим действия для формирования первой строки таблицы переходов ДКА.

Move({1, 2, 12}, a) = {3} ; ε-closure({3}) = {3, 4, 5, 7, 10}

Move({1, 2, 12}, b) = ∅ (перехода из первого состояния ДКА по символу b нет). Эти действия продолжаются для остальных строк таблицы переходов в соответствии с алгоритмом 3.6.

№	State	a	b
1	1, 2, **12**	3,4,5,7,10	–
2	3,4,5,7,10	4,5,6,7,9,10	2,4,5,7,8,9,10,11,12
3	4,5,6,7,9,10	4,5,6,7,9,10	2,4,5,7,8,9,10,11,12
4	2,4,5,7,8,9,10,11,**12**	3,4,5,6,7,9,10	2,4,5,7,8,9,10,11,12
5	3,4,5,6,7,9,10	4,5,6,7,9,10	2,4,5,7,8,9,10,11,12

Рис. 3.11 Таблица переходов ДКА эквивалентного НКА рис. 3.10

Состояния ДКА 1 и 4 являются допускающими. Диаграмма переходов ДКА, соответствующего НКА на рис. 3.10, показана на рис. 3.12.

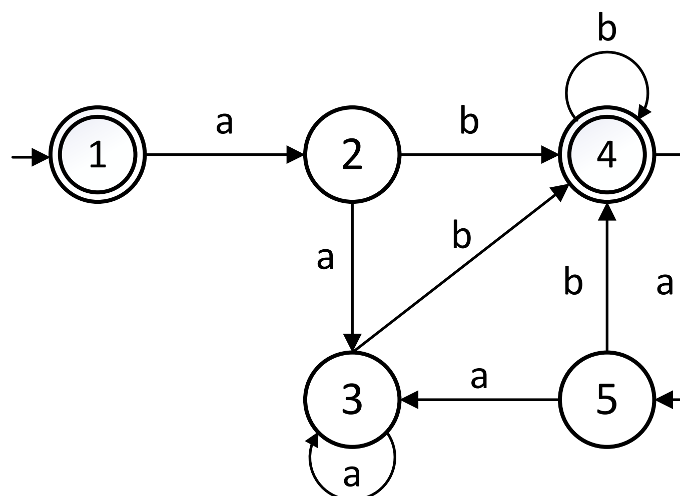


Рис 3.12 ДКА эквивалентный НКА рис. 3.10

Следующим шагом является минимизация полученного ДКА.

Обозначим через P_0 начальное разбиение состояний ДКА на классы эквивалентности. $P_0 = \{A_0 = <2, 3, 5> B_0 = <1, 4>\}$. Классу B_0 соответствуют все допускающие состояния, а классу A_0 не допускающие. Последующие шаги минимизации показаны на рис. 3.13.

			P_0		P_1		P_2	
δ	a	b	a	b	a	b	a	B
1	2	–	A_0	–	A_1	–	A_2	–
2	3	4	A_0	B_0	A_1	C_1	A_2	C_2
3	3	4	A_0	B_0	A_1	C_1	A_2	C_2
4	5	4	A_0	B_0	A_1	C_1	A_2	C_2
5	3	4	A_0	B_0	A_1	C_1	A_2	C_2

Рис. 3. 13. Классы эквивалентных состояний ДКА

После первого шага произошло разделения состояний 1 и 4 в разные классы эквивалентности: $P_1 = \{A_1 = <2, 3, 5> B_1 = <1> C_1 = <4>\}$.
После второго шага классы эквивалентных состояний остались неизменными: $P_2 = \{A_2 = <2, 3, 5> B_2 = <1> C_2 = <4>\}$. $P_1 = P_2$, минимизация завершена, у полученного минимального ДКА три состояния, из них два допускающих.

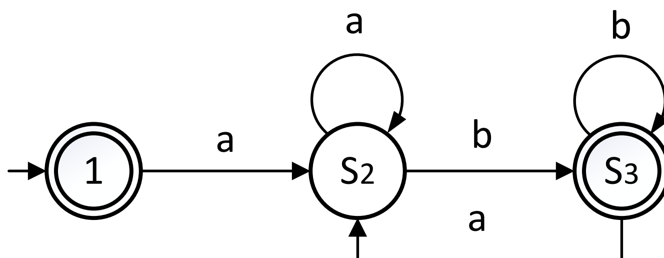


Рис 3.14 Диаграмма переходов минимального автомата для (a(a|b)*b)*.

3.6 Лемма о разрастании для регулярных языков

Существует простой метод проверки, является ли данный язык регулярным. Этот метод основан на проверке леммы о разрастании регулярного языка. Доказано, что если для некоторого заданного языка выполняется лемма о разрастании регулярного языка, то этот

язык является регулярным; если же лемма не выполняется, то и язык регулярным не является.

Лемма о разрастании или накачки (Pumping Lemma) для регулярных языков формулируется следующим образом: если L – регулярный язык и α – достаточно длинная цепочка символов, принадлежащая этому языку, то в этой цепочке можно найти непустую подцепочку, которую можно повторить сколь угодно много раз, и все полученные таким способом новые цепочки будут принадлежать тому же регулярному языку, исходная цепочка «разрастается» — отсюда и название «лемма о разрастании»). Лемма о разрастании имеет смысл только для бесконечных регулярных языков.

Формально эту лемму можно определить так: если задан регулярный язык L, то существует константа $n > 0$, такая, что если $\alpha \in L$ и $|\alpha| \geq n$, то цепочку α можно записать в виде $\alpha = \beta\gamma\delta$, где $0 < |\gamma|$, и тогда $\alpha = \beta\gamma^i\delta$, $\alpha \in L$ для всех $i > 0$. Используя лемму о разрастании регулярных языков, докажем, что язык $L = \{a^n b^n \mid n > 0\}$ не является регулярным.

Положим, что $L = \{a^n b^n \mid n > 0\}$ – регулярный язык. Тогда должны существовать три подцепочки β, γ и δ, причем $\gamma \neq \varepsilon$, и $\beta\gamma^n\delta \in L$ для всех $n > 0$. Существуют три возможных варианта подцепочки γ:

- γ состоит только из символов a (хотя бы одного a). Тогда цепочка $\beta\gamma^2\delta$ будет содержать больше символов a, чем b, и, значит, эта цепочка не принадлежит $\{ a^n b^n \mid n>0\}$;
- γ состоит только из символов b (хотя бы одного b). Тогда цепочка $\beta\gamma^2\delta$ будет содержать больше символов b, чем a, и, значит, эта цепочка не принадлежит $\{ a^n b^n \mid n>0\}$;
- $\gamma = a^i b^j$ $(i, j > 0)$, тогда даже цепочка $\beta\gamma^2\delta$ будет содержать символы b за которыми следуют символы a, что для языка $L = \{a^n b^n \mid n > 0\}$ недопустимо.

Мы рассмотрели все возможные варианты для подцепочки γ, и ни один из них не подошел; таким образом, язык L не является регулярным языком.

3.7 Свойства регулярных языков

Множество называется замкнутым относительно некоторой операции, если в результате выполнения этой операции над любыми элементами, принадлежащими данному множеству, получается элемент, принадлежащий тому же множеству. Например, множество

натуральных чисел замкнуто относительно операций сложения и умножения, но не замкнуто относительно операций вычитания и деления.

Регулярные языки, а язык – это множество, замкнуты [14] относительно следующих операций:

- пересечения;
- объединения;
- дополнения;
- разности;
- обращения;
- итерации;
- конкатенации;
- гомоморфизма (изменения имен символов и подстановки цепочек вместо символов).

Поскольку регулярные множества замкнуты относительно операций пересечения, объединения и дополнения, то они представляют булеву алгебру множеств. Пусть $A = (X, S, s_0, \delta, F)$ – конечный автомат, допускающий язык L, тогда в булевой алгебре языку $L \subseteq X^*$, допускаемому конечным автоматом A, будет соответствовать true (или 1), а пустому языку $L = \emptyset$ соответствует false (или 0). Все указанные ниже аксиомы булевой алгебры верны для регулярных языков x, y, z, причем $x, y, z \subseteq X^*$:

$x \cap y = y \cap x$ $\qquad$ $x \cup y = y \cup x$

$(x \cap y) \cup z = (x \cup z) \cap (y \cup z)$ $\qquad$ $(x \cup y) \cap z = (x \cap z) \cup (y \cap z)$

$\overline{x} \cup x = X^*$, где $\overline{x} = X^* - x$ $\qquad$ $\overline{x} \cap x = \emptyset$

$x \cup \emptyset = x$ $\qquad$ $x \cap X^* = x$

Доказано [9], что в рамках регулярных языков разрешимы следующие проблемы:

- *Проблема эквивалентности.* Пусть заданы два регулярных языка $L_1(VT_1)$ и $L_2(VT_2)$. Здесь VT_1 и VT_2 – словари терминальных символов грамматик, порождающих эти языки. Необходимо проверить, являются ли эти два языка эквивалентными. Достаточно доказать эквивалентность КА, соответствующего им.

- *Проблема принадлежности цепочки языку.* Дан регулярный язык L(VT) и цепочка символов $\alpha \in VT^*$. Необходимо проверить, принадлежит ли цепочка данному языку.
- *Проблема пустоты языка.* Дан регулярный язык L(VT). Необходимо проверить, является ли этот язык пустым, то есть найти хотя бы одну цепочку $\alpha \neq \varepsilon$, такую что $\alpha \in L(VT)$, иначе нужно доказать, что начальный символ грамматики – производящий, см. п. 4.4.1.

Эти проблемы разрешимы вне зависимости от того, каким из трех способов задан регулярный язык. Достаточно доказать разрешимость любой из этих проблем хотя бы для одного из способов представления языка, тогда для остальных способов можно воспользоваться алгоритмами преобразования, рассмотренными выше или в [9, 12, 15, 20].

Для регулярных грамматик также разрешима проблема однозначности (вопрос однозначности подробно рассмотрен в п. 4.2.) – доказано, что для любой регулярной грамматики можно построить эквивалентную ей однозначную регулярную грамматику, поскольку для любой регулярной грамматики можно однозначно построить регулярное выражение, определяющее язык, порождаемый этой грамматикой.

Контрольные вопросы

1. Дайте определение автомата-распознавателя.
2. Чем недетерминированный КА отличается от детерминированного КА?
3. Как соотносятся регулярные грамматики и регулярные выражения?
4. Дайте определение минимального ДКА.
5. Перечислите основные операции над регулярными языками.
6. Как формально определить принадлежность языка к типу регулярных языков?
7. Что понимается под замыканием регулярных языков относительно операции объединения?
8. Перечислите операции над множеством состояний НКА для его преобразования в ДКА.
9. Для чего используется ε-замыкание?
10. Перечислите свойства регулярных языков.

4 Контекстно-свободные языки и МП-автоматы

Исследования, посвященные контекстно-свободным грамматикам и порождаемым ими языкам, получили широкое развитие, и сегодня теория контекстно-свободных грамматик представляет собой центральную составную часть математической лингвистики.

Контекстно-свободные языки играют большую роль в проектировании синтаксических анализаторов в силу того, что посредством КС-грамматик можно определить основную часть синтаксиса языков программирования. Нужно заметить, что большинство языков программирования не являются КС-языками в силу контекстных зависимостей, например требования соответствия числа и типов формальных и фактических параметров процедур и функций.

4.1 Синтаксические деревья

Процесс вывода сентенциальной формы для КС-грамматики $G = (VT, VN, P, S)$ может быть наглядно представлен с помощью *синтаксического дерева*, или иначе дерева разбора. Синтаксическое дерево, соответствующее сентенциальной форме, это связный ацикличный граф такой, что:

- корень дерева – вершина с нулевой степенью захода – помечается начальным символом грамматики S;
- узлы дерева помечаются нетерминальными символами грамматики; если узел дерева помечен символом A, $A \in VT$, и все исходящие из этого узла дуги связаны с вершинами, помеченными $v_1, v_2, \dots v_n$, причем $n>0$; $v_i \in VN \cup VT \cup \{\varepsilon\}$; то в грамматике должно существовать правило $A \to v_1v_2 \dots v_n \in P$.
- листья дерева (терминальные узлы) для общего случая сентенциальной формы помечаются символами из множества $VN \cup VT \cup \{\varepsilon\}$; если сентенциальная форма является предложением языка, то терминальные узлы помечаются символами из множества $VT \cup \{\varepsilon\}$.

При обходе листьев дерева слева направо получается цепочка символов, соответствующая сентенциальной форме, вывод которой иллюстрирует синтаксическое дерево.

Пример 4.1. Рассмотрим процесс построения синтаксического дерева для грамматики $G_1 = (\{a, b, +\}, \{S, A\}, \{S \to A \mid A+S; A \to a \mid b\}, S)$.

В качестве примера рассмотрим левосторонний вывод для предложения a+b+b:

$$
\begin{aligned}
S &\Rightarrow A + S\\
&\Rightarrow a + S\\
&\Rightarrow a + A + S\\
&\Rightarrow a + b + S\\
&\Rightarrow a + b + A\\
&\Rightarrow a + b + b
\end{aligned}
$$

$S \Rightarrow A + S \Rightarrow a + S \Rightarrow a + A + S \Rightarrow a + b + S \Rightarrow a + b + A \Rightarrow a + b + b$

Этому выводу соответствует процесс построения синтаксического дерева, представленный на рис. 4.1.

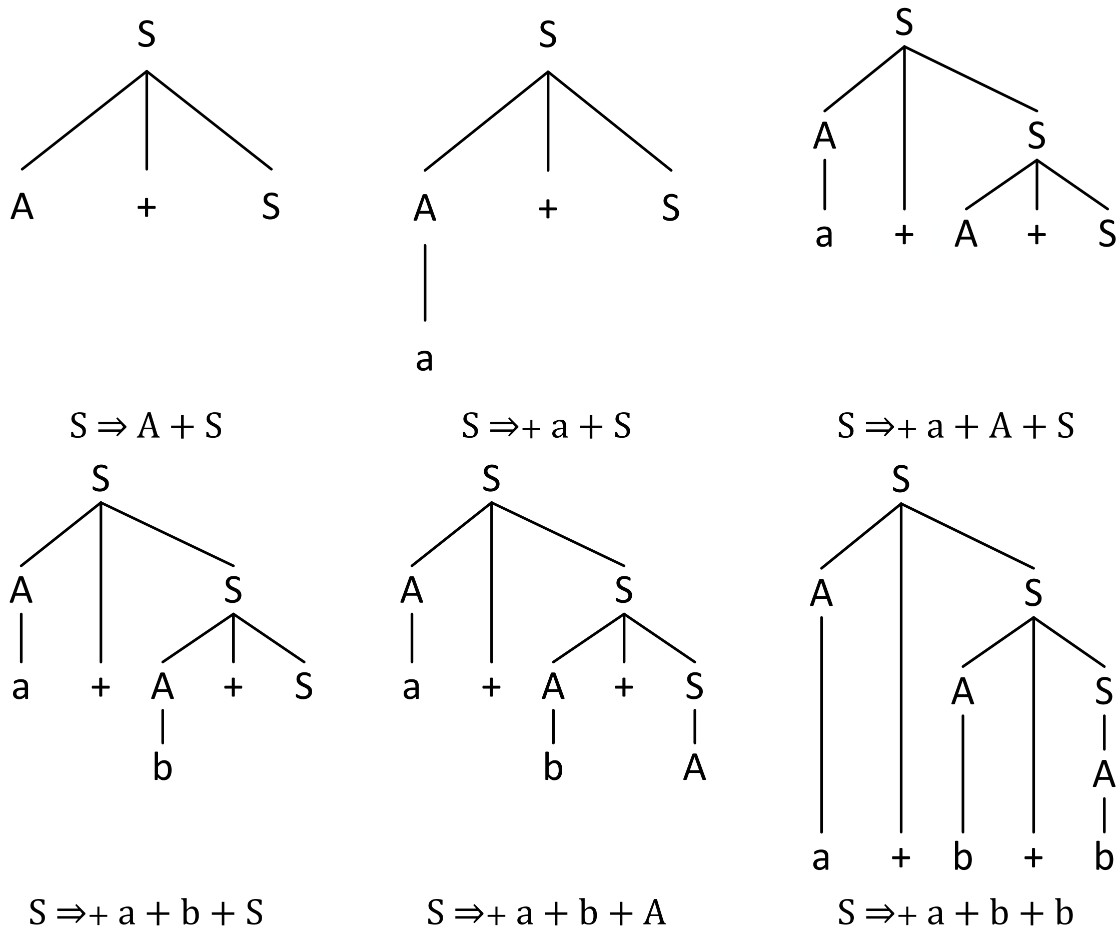


Рис. 4.1 Построение дерева вывода цепочки a + b + b

Пример 4.2. Рассмотрим грамматику $G_2 = (VT, VN, P, S)$, $VT = \{a, b\}$, $VN = \{S, A\}$; P:

$$
\begin{aligned}
S &\to AA\\
A &\to AAA \mid bA \mid Ab \mid a.
\end{aligned}
$$

Примеры вывода предложения ababaa представлены на рис. 4.2.

a)	b)	c)	d)
S ⇒ AA	S ⇒ AA	S ⇒ AA	S ⇒ AA
⇒ aA	⇒ AAAA	⇒ Aa	⇒ aA
⇒ aAAA	⇒ aAAA	⇒ AAAa	⇒ aAAA
⇒ abAAA	⇒ abAAA	⇒ AAbAa	⇒ aAAa
⇒ abaAA	⇒abaAA	⇒ AAbaa	⇒ abAAa
⇒ ababAA	⇒ ababAA	⇒ AbAbaa	⇒ abAbAa
⇒ ababaA	⇒ababaA	⇒ Ababaa	⇒ ababAa
⇒ ababaa	⇒ ababaa	⇒ ababaa	⇒ ababaa

Рис. 4.2. Альтернативные выводы цепочки ababaa для G_2.

Вывод a) является левосторонним, потому что на каждом шаге вывода раскрывается самый левый нетерминал сентенциальной формы; b) соответствует альтернативному левостороннему выводу, c) представляет собой правосторонний вывод, когда на каждом шаге заменяется самый правый нетерминал сентенциальной формы, вывод d) демонстрирует смешанную технику.

4.2 Неоднозначность грамматик

Грамматика называется *однозначной,* если для каждого предложения в языке, заданного этой грамматикой, можно построить *единственный левосторонний* (и/или единственный правосторонний) вывод. Каждому предложению языка, заданного однозначной грамматикой, соответствует единственное синтаксическое дерево.

Грамматика называется *неоднозначной*, если существует хотя бы одно предложение, порождаемое данной грамматикой, для которого существуют два и более синтаксических деревьев. Язык, порождаемый неоднозначной грамматикой, является неоднозначным, если он не может быть порожден какой-либо однозначной грамматикой, очевидно, что эти грамматики являются эквивалентными.

Пример 4.3 Пусть G_1 – грамматика вида S → aS | Sa | a. Для этой грамматики существуют два левосторонних вывода цепочки aa:

S ⇒ aS ⇒ aa

S ⇒ Sa ⇒ aa, таким образом грамматика G_1 – неоднозначна.

Пусть G_2 – грамматика вида S → aS | a. Вывод цепочки aa для этой грамматики является единственным: S ⇒ aS ⇒ aa, следовательно, эта грамматика однозначна. Пример 4.3 показывает, что две грамматики

порождают один и тот же язык L = {w | w = a^+}, и, значит, неоднозначность – свойство грамматики, а не языка.

Проблемы, связанные с неоднозначными грамматиками:

- проблема, порождает ли данная КС-грамматика однозначный язык; или иначе, существует ли однозначная КС-грамматика, эквивалентная данной – алгоритмически неразрешима;
- проблема, является ли данная грамматика однозначной – алгоритмически неразрешима. См. свойства КС-языков п. 4.9.

Можно доказать, что если грамматика содержит правила следующего вида:

$A \to AA \mid \alpha$ $\qquad A \in VN$

$A \to A\alpha A \mid \beta$ $\qquad \alpha, \beta, \gamma \in V^*$

$A \to \alpha A \mid A\beta \mid \gamma$

$A \to \alpha A \mid \alpha A\beta A \mid \gamma$

то она точно неоднозначна [8]. Однако отсутствие таких правил не гарантирует того, что грамматика является однозначной.

Грамматика из примера 4.1 является однозначной. Читателю предлагается построить синтаксическое дерево, соответствующее правостороннему выводу предложения a + b + b, и сравнить его с деревом, показанным на рис. 4.1.

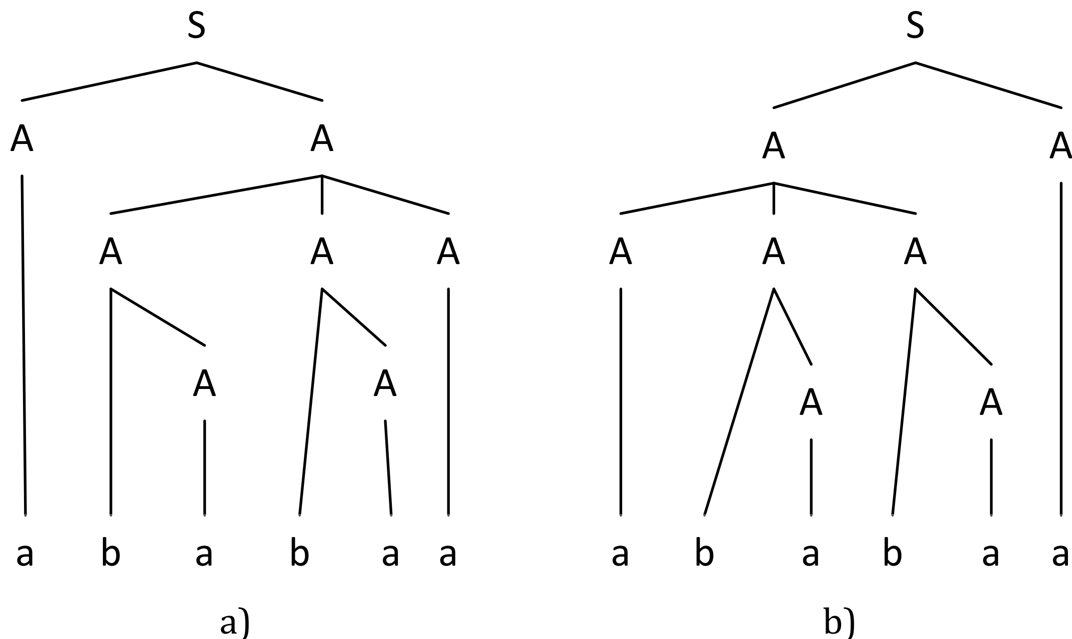


Рис. 4.3 Синтаксические деревья для выводов a) и b) на рис. 4.2

Деревья, полученные для выводов цепочки ababaa из примера 4.2, показаны на рис. 4.3. Грамматика из примера 4.2 неоднозначна.

4.3 Автоматы с магазинной памятью

Подобно тому, как регулярный язык может быть определен с помощью конечного автомата, так и контекстно-свободные языки могут быть определены с помощью автоматов с магазинной памятью (МП-автомат или pushdown automata). МП-автомат является расширением недетерминированного конечного автомата с ε-переходами.

МП-автомат состоит из трех компонент:

- входная лента, соответствующая цепочке входных символов;
- управляющее устройство (НКА с ε-переходами);
- стек (или иначе «магазин»).

Для стека определены две операции:

- pop – снять символ с вершины стека;
- push – поместить символ на вершину стека.

Схема МП-автомата представлена на рис. 4.4. Управляющий устройство имеет доступ к содержимому вершины стека, т. е. может считать символ с вершины стека без выполнения операции pop. Считывающая головка перемещается по входной ленте только слева-направо. МП-автомат является синхронным и в каждом такте может читать, или не читать, символ входной ленты, но должен учитывать содержимое вершины стека для выполнения каждого перехода.

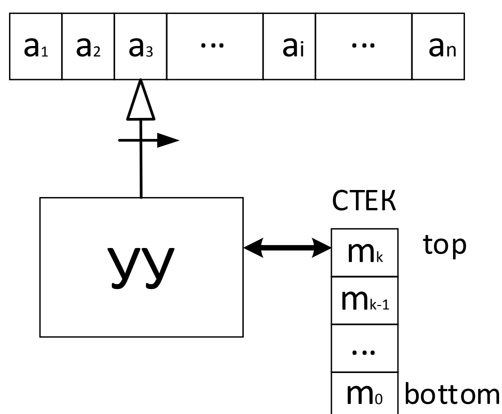


Рис 4.4. Структурная схема МП-автомата

Формально МП-автомат может быть определен как семерка или кортеж вида $PDA = (X, S, s0, F, \delta, M, m0)$, где

- X – множество входных символов (входной алфавит);
- S – конечное множество состояний МП-автомата;
- s_0 – начальное состояние МП-автомата ($s_0 \in S$);
- F – множество допускающих состояний ($F \subseteq S$);
- M – множество символов стека;
- m_0 – начальный символ стека ($m_0 \in M$), в начале работы стек содержит этот единственный символ;
- δ – функция переходов, δ: $(S \times (X \cup \{\varepsilon\}) \times M) \to (S \times M^*)$.

Под конфигурацией МП-автомата понимается тройка вида (s, ω, γ), где s – текущее состояние автомата, цепочка ω – необработанные часть входной ленты, включая самый левый символ, находящийся под головкой чтения, и цепочка γ – содержимое стека (обычно символы от вершины к дну стека записываются слева-направо).

Начальная конфигурация МП-автомата определяется тройкой (s_0, ω, m_0). На каждом такте работы МП-автомата в соответствии с функцией переходов осуществляется смена конфигурации. Введем отношение непосредственной смены конфигураций, обозначаемое $\vdash$, пусть $\delta(s_1, t, m)$, причем $s_1 \in S, t \in X, m \in M$, содержит пару (s_2, α) $s_2 \in S$, $\alpha \in M^*$, тогда для всех цепочек $\omega \in X^*$ и $\beta \in M^*$ справедливо следующее отношение: $(s_1, t\omega, m\beta) \vdash (s_2, \omega, \alpha\beta)$.

С учетом повторяющихся, но формально необходимых, цепочек ω и β процесс смены конфигурации можно упрощенно записать следующим образом: $(s_1, t, m) \vdash (s_2, \alpha)$. Под этим понимается следующее: если МП-автомат находится в состоянии s_1, причем t – это текущий символ на входной ленте, m – символ на вершине стека, то осуществляется передвижение считывающей головки к следующему символу входной ленты, символ m снимается с вершины стека (выполняется операция pop), а символы из цепочки α помещается на вершину стека (для всех символов из этой цепочки выполняется слева-направо выполняется операция push), и МП-автомат переходит в состояние s_2. Графически переход между состояниями МП-автомата показан на рисунке 4.5.

Возможны следующие случаи:

- $(s_1, \varepsilon, m) \vdash (s_2, \alpha)$ – текущий входной символ не рассматривается, продвижение считывающей головки нет;
- (s1, t, m) ⊢ (s2, ε) – символ m снимается с вершины стека, на вершину стека ничего не помещать. Допускаются использование пустой цепочки и в других вариантах, например $(s_1, \varepsilon, m) \vdash (s_2, \varepsilon)$.

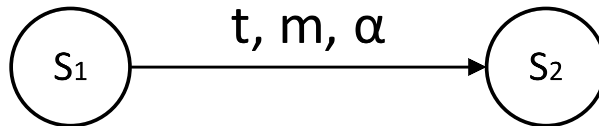


Рис 4.5. Помеченная дуга для диаграммы переходов МП-автомата

Конечной конфигурацией МП-автомата является тройка (q, ε, α) где $q \in F$, $\alpha \in M^*$. Обозначим $\vdash*$ последовательность ноль или более переходов между состояниями МП-автомата.

Говорят, что МП-автомат допускает цепочку входящих символов ω, если существует путь по конфигурациям $(s_0, \omega, m_0) \vdash* (q, \varepsilon, \alpha)$ для некоторых $q \in F$ и $\alpha \in M^*$. Множество, возможно бесконечное, допустимых цепочек является языком, допускаемым МП-автоматом.

Если МП-автомат прочитал входную цепочку до конца и в заключительной конфигурации он оказался в одном из допускающих состояний, то говорят о допуске цепочки *по допускающему состоянию*. Существует второй способ задания языка, допускаемого МП-автоматом, который называется допуск входной цепочки *по пустому стеку*. В этом случае цепочка допускается МП-автоматом, если после старта из начального состояния, после прочтения всех символов цепочки, стек МП-автомата пуст.

Язык L, допускаемый МП-автоматом по допускающему состоянию, определяется следующим образом:

$$L(M) = \{\omega \mid \omega \in X^* \,\&\, (s_0, \omega, m_0) \vdash* (q, \varepsilon, \alpha) \text{ для некоторых } q \in F \text{ и } \alpha \in M^*\}.$$

МП-автомат, начиная с начальной конфигурации, последовательно читает символы входной цепочки, которая считается допустимой, если она прочитана до конца, и МП-автомат перешел в одно из допускающих состояний. Содержимое стека при этом не имеет значения.

Язык L, допускаемый МП-автоматом по пустому стеку, определяется следующим образом: $L(M) = \{\omega \mid \omega \in X^* \,\&\, (s_0, \omega, m_0) \vdash*$

$(q, \varepsilon, \varepsilon)$ где $q \in S\}$, состояние q может быть любым, необязательно допускающим. МП-автомат, начиная с начальной конфигурации, последовательно читает символы входной цепочки, которая допускается МП-автоматом, если она прочитана до конца и стек пуст.

Классы КС-языков, допускаемых по заключительному состоянию и по пустому стеку, совпадают. Любой контекстно-свободный язык может быть допущен недетерминированным МП-автоматом либо по допускающему состоянию, либо по пустому стеку, и каждый недетерминированный МП-автомат допускает либо по допускающему состоянию, или по пустому стеку любой контекстно-свободный язык.

Для любой КС-грамматики может быть построен недетерминированный МП-автомат, который допускает язык, порождаемый данной грамматикой.

Алгоритм 4.1. Построение МП-автомата по КС-грамматике
Вход КС-грамматика $G = (VT, VN, P, S)$
Выход МП-автомат $PDA = \langle X, S, s_0, F, \delta, M, m_0 \rangle$

Шаг 1 $S = \{s\}$; $s_0 = s$; $X = VT$; $M = VT \cup VN$; $m_0 = S$; $F = \emptyset$;

Шаг 2 **foreach** $A \rightarrow \alpha$ **in** P

Добавить в функцию переходов

δ: $(s, \varepsilon, A) \rightarrow (s, \alpha)$

Шаг 3 **foreach** t **in** VT

Добавить в функцию переходов

δ: $(s, t, t) \rightarrow (s, \varepsilon)$

Результатом работы данного алгоритма является множество правил для функции переходов автомата. Полученный в результате работы алгоритма МП-автомат может быть использован для распознавания цепочек терминальных символов из соответствующего словаря грамматики. Если на вход этого автомата подать цепочку терминальных символов, то автомат будет осуществлять действия, аналогичные левостороннему выводу из начального символа грамматики – построение синтаксического дерева методом сверху вниз.

Можно построить МП-автомат, соответствующий произвольной КС-грамматике, который будет осуществлять правосторонний вывод для сентенциальных форм грамматики, пытаясь построить

синтаксическое дерево для входной цепочки терминальных символов, осуществляя свертку исходной цепочки к начальному символу грамматики.

Пример 4.4.

Рассмотрим в качестве исходной грамматику G, построим эквивалентный ей МП-автомат. G = ({+, (,), a}, {S, A}, P, {S}), где
P = {S→S+A | A, A→(S) | a}.

Шаг 1 — S = {s}; s_0 =s; X = {+, (,), a};
M = { S, A , +, (,), a };
m_0 = S; F = ∅;

Шаг 2. — (s, ε, S) → (s, S+A)
(s, ε, S) → (s, A)
(s, ε, A) → (s, (S))
(s, ε, A) → (s, a)

Шаг 3. — (s, +, +) → (s, ε)
(s, (, ( ) → (s, ε)
(s,),)) → (s, ε)
(s, a, a) → (s, ε)

Алгоритм 4.2 Построение расширенного МПА по КС-грамматике
Вход КС- грамматика G = (VT, VN, P, S)
Выход МП-автомат PDA = <X, S, s_0, F, δ, M, m_0>

Шаг 1 S = {s, f}; s_0 =s; X = VT; M = VT ∪ VN ∪ {#}; m_0 = #; F = {f};
Шаг 2 **foreach** A → α **in** P
Добавить в функцию переходов
δ: (s, ε, α) → (s, A)
Шаг 3 **foreach** t **in** VT
Добавить в функцию переходов
δ: (s, t, ε) → (s, t)
Шаг 4 Добавить в функцию переходов (s, ε, #S) → (f, ε)

Для грамматики запрещены ε-правила, а МП-автомат должен уметь выполнять операции pop и push не для отдельных символов грамматик, а для цепочек символов, соответствующих правым частям правил грамматики. Такой МП-автомат принято называть расширенным МП-автоматом. Процесс построения такого МП-автомата определяется алгоритмом 4.2.

Пример 4.5. Рассмотрим в качестве исходной грамматику из примера 4.4 и построим эквивалентный ей МПА, применив алгоритм 4.2.

G({+, (,), a}, {S, A}, {S→S+A | A, A→(S) | a}, {S}).

Шаг 1 — $S = \{s, f\}$; $s_0 = s$; $X = \{+, (,), a\}$;
$M = \{S, A, +, (,), a, \#\}$;
$m_0 = \#$; $F = \{f\}$;

Шаг 2. — (s, ε, S+A) → (s, S)
(s, ε, A) → (s, S)
(s, ε, (S)) → (s, A)
(s, ε, A) → (s, a)

Шаг 3. — (s, +, ε) → (s, +)
(s, (, ε) → (s, ( )
(s,), ε) → (s,))
(s, a, ε) → (s, a)

Шаг 4. — (s, ε, #S) → (f, ε)

МП-автоматы по определению являются недетерминированными, но можно выделить подкласс детерминированных МП-автоматов (ДМП-автоматы). МП-автомат $A = \langle X, S, s_0, F, \delta, M, m_0 \rangle$ определяется как детерминированный, если

- $\delta(s, x, m)$ имеет не более одного элемента для каждого $s \in S$,и $x \in X$ (даже если $x = \varepsilon$), и $m \in M$;
- если $\delta(s, x, m)$ не пусто для некоторого $x \in X$, то $\delta(s, \varepsilon, m)$ должно быть пустым.

В отличие от детерминированных и недетерминированных конечных автоматов, классы языков, допускаемых

недетерминированными и детерминированными МП-автоматами, не совпадают (класс языков, допускаемых недетерминированным МП-автоматом шире и включает в себя все КС-языки), следовательно, для общего случая КС-языка не может быть выполнено преобразование распознающего его недетерминированного МП-автомата к детерминированному. Более того, классы языков, допускаемых ДМП-автоматом по допускающему состоянию и по пустому стеку, тоже не совпадают; класс языков, допускаемых по допускающему состоянию, шире.

Важной особенностью ДМП-автоматов является то, что если он допускает КС-язык, то этот язык – однозначный [15]. Следовательно, для однозначных КС-языков может быть сконструирован ДМП-автомат, который будет решать задачу распознавания цепочки, порожденной такой грамматикой за линейное время. Понятно, что моделирование поведения ДМП-автоматов существенно проще, чем моделирование недетерминированных МП-автоматов.

Детерминированные КС-языки – это класс КС-языков, сентенциальные формы которых можно распознавать с помощью ДМП-автоматов. Класс детерминированных КС-языков является собственным подмножеством всего класса КС-языков.

Класс детерминированных КС-языков интересен тем, что для него разрешима проблема однозначности. Доказано, что если язык может быть распознан с помощью ДМП-автомата (и потому является детерминированным КС-языком), то он может быть описан на основе однозначной КС-грамматики. Поэтому данный класс КС-языков используется для построения синтаксических конструкций языков программирования.

4.4 Упрощение КС-грамматик

Прежде чем разрабатывать распознаватель для КС-языков, заданных КС-грамматикой, имеет смысл упростить эту грамматику, которая, например, могла быть получена формальным способом, удалив из нее все правила, которые невозможно использовать в выводе хотя бы одного предложения.

Пусть $G = (VT, VN, P, S)$ – КС-грамматика.

Нетерминальный символ $A \in VN$ называется непроизводящим, если множество терминальных цепочек, выводимых из него, пусто, обозначается $\{\alpha \mid \alpha \in VT^*, A \Rightarrow* \alpha\} = \emptyset$.

Символ, принадлежащий словарю грамматики V, называется недостижимым, если он не может быть получен ни в одной из сентенциальных форм. Непроизводящие и недостижимые символы называются *бесполезными*. Грамматика называется *приведенной*, если из неё удалены все правила, содержащие бесполезные символы.

Приведение грамматики – удаление правил с бесполезными символами, может быть выполнено на основании двух очевидных свойств.

Если все нетерминальные символы в правой части правила – производящие, то и символ в левой части правила – производящий. Производящий нетерминал N строго может быть определен следующим образом: $N \Rightarrow^{+} \mu, \mu \in VT^{+}$.

Если символ в левой части правила достижим, то и символы правой части правила достижимы. Иначе говоря, для любого нетерминала $A \in VN$ должно быть справедливо: $S \Rightarrow^{*} \alpha A \beta$, где $\alpha, \beta \in V^{*}$.

4.4.1 Удаление непроизводящих символов

Неформально процесс удаления непроизводящих символов может быть представлен следующим образом. Последовательно просматриваются правила грамматики. Нетерминальный символ из левой части правила считается порождающим, если цепочка в правой части правила содержит только терминальные символы, или все нетерминалы этой цепочки являются элементами множества производящих символов. Последовательные просмотры правил продолжаются до тех пор, пока после очередного просмотра в множество производящих символов не будет добавлено ни одного нового элемента. Все правила, содержащие нетерминалы, не попавшие в множество производящих символов, удаляются из грамматики. Алгоритм удаления непроизводящих символов представлен ниже.

Пусть задана исходная КС-грамматика G = (VT, VN, P, S). Необходимо построить эквивалентную ей грамматику G’ = (VT, VN’, P’, S) такую, что:

- VN’ содержит только нетерминальные символы, из которых могут быть выведены цепочки VT’, т.е. для всех нетерминалов $A \in VT'$ существует цепочка $\alpha \in VT^{*}$ такая, что $A \Rightarrow^{*} \alpha$; и
- P’ содержит только те правила, все символы которых являются элементами множества $VT \cup VN'$.

Множество VN’ может быть получено посредством следующего алгоритма 4.3. Затем в множество правил P’ помещаются только те правила из исходного множества P, у которых все нетерминальные символы в левой и правой частях принадлежат множеству VN’. Обычно, если $A \in VN'$, и $A \rightarrow \varepsilon \in P$, то правило $A \rightarrow \varepsilon$ помещается в P’.

Алгоритм 4.3 Удаление непроизводящих символов
Вход G = (VT, VN, P, S)
Выход G’ = (VT, VN’, P’, S)

```
VN’ = ∅
flag =1
while(flag){
        flag = 0
        foreach A → α in P {
            if(α ∈ (VT ∪ VN’)*)
              {VN’.add(A); flag =1}}
    }
```

Пример 4.6. Рассмотрим грамматику G = ({a, b}, {A, B, C, D, E, F, S}, P, S).

Исходное множество правил P	Формирование множества VN’	Результирующая грамматика
S → AC | BS |B	$VN'_0 = \{\emptyset\}$	G’ = (VT, VN’, P’, S)
A → aA | aF	$VN'_1 = \{B, F\}$	VT = {a, b}
B → CF | b	$VN'_2 = \{B, F, A, S\}$	VN’ = {A, B, E, F, S}
C → cC | D	$VN'_3 = \{B, F, A, S, E\}$	P’ = {S → BS | B
D → aD | BD | C	$VN'_4 = \{B, F, A, S, E\}$	A → aA | aF
E → aA | BSA	$VN'_4 = VN'_3$	B → b
F → bB | b	FINISH	E → aA | BSA
		F → bB | b}

4.4.2 Удаление недостижимых символов

Неформально процесс удаления из словаря V исходной грамматики недостижимых символов может быть описан следующим образом. Начальный символ грамматики S считается достижимым.

Далее последовательно просматриваются правила, в левой части которых находится достижимый нетерминальный символ, и все символы в правой части этих правил помещаются в множество достижимых. Последовательные просмотры правил продолжаются до тех пор, пока после очередного просмотра в множество достижимых символов не будет добавлено ни одного нового элемента. Все правила, содержащие недостижимые символы, удаляются из грамматики.

Пусть задана исходная КС-грамматика G = (VT, VN, P, S). Необходимо построить эквивалентную ей грамматику G' = (VT', VN', P', S) такую, что символ из (VT' ∪ VN') может быть получен в сентенциальной форме грамматики, иначе говоря для, всех символов $x \in (VT' \cup VN')$ существуют $\alpha, \beta \in (VT' \cup VN')^*$ такие, что $S \Rightarrow^* \alpha x \beta$.

Множества VT' и VN' могут быть получены как результат работы следующего алгоритма.

Алгоритм 4.4 Удаление недостижимых символов
Вход G = (VT, VN, P, S).
Выход G' = (VT, VN', P', S)

```
    VN' = {S}
    VT' = ∅
    flag = 1
    while(flag){
        foreach A → α in P {
            flag = 0
            foreach x in α {
                if (A ∈ VN' && x ∈ VN)
                    {VN'.add(x) flag = 1}
                if (A ∈ VN' && x ∈ VT)
                    VT'.add(x)
    }}}
```

Затем множество правил P' конструируется только из тех правил исходного множества P, все символы которого принадлежат (VT' ∪ VN'). Обычно, если A ∈ VN', и $A \rightarrow \varepsilon \in P$, то правило $A \rightarrow \varepsilon$ помещается в P'.

Исходная грамматика из примера 4.6, к которой были последовательно применены алгоритмы 4.3 и 4.4, становится

грамматикой без бесполезных символов, т. е. *приведенной*. Нетрудно убедится, что полученная приведенная грамматика из примера 4.7 порождает язык $L(G') = \{b^n \mid n > 0\}$.

Важным является последовательность применения алгоритмов 4.3 и 4.4. Применение этих же алгоритмов в обратной последовательности не гарантирует удаление из грамматики бесполезных символов, а значит, грамматику нельзя считать приведенной.

Пример 4.7 В качестве исходной берем результирующую грамматику из примера 4.6.

Исходное множество правил	Формирование множеств VT' и VN'	Результирующая грамматика
$S \to BS \mid B$	$VN'_0 = \{S\}$ $VT'_0 = \emptyset$	
$A \to aA \mid aF$	$VN'_1 = \{S, B\}$ $VT'_1 = \{b\}$	$G' = (VT', VN', P', S)$
$B \to b$	$VN'_2 = \{S, B\}$ $VT'_2 = \{b\}$	$VT' = \{b\}$
$E \to aA$ I BSA	$VN'_2 = VN'_1$	$VN' = \{A, B, E, F, S\}$
$F \to bB \mid b$	FINISH	$P' = \{S \to BS \mid B$ $B \to b\}$

Пусть задана КС-грамматика $G = (VT, VN, P, S)$; если $A \to B \in P$ – правило грамматики, а символы A и B $\in$ VN, то такое правило называется *цепным*. Дальнейшее упрощение грамматик может быть осуществлено в процессе удаления ε-правил и цепных правил. Важно понимать, что под упрощением понимается не уменьшение количества правил грамматики или символов её словаря, а повышение однотипности этих правил, что упрощает проектирование распознавателей, при этом количество правил грамматики, а также нетерминалов, скорее всего, увеличится.

4.4.3 Удаление ε-правил

Контекстно-свободная грамматика $G = (VT, VN, P, S)$ называется расширенной КС-грамматикой, если в множестве правил P содержится одно или более ε-правила (правила вида $A \to \varepsilon$).

Грамматика называется S-расширенной КС-грамматикой (VT, VN, P, S), если множество P содержит правило $S \to \varepsilon$.

Для любой расширенной КС-грамматики G, такой что $\varepsilon \notin L(G)$, существует эквивалентная КС-грамматика G' без ε-правил.

Для любой расширенной КС-грамматики G, такой что $\varepsilon \in L(G)$, существует эквивалентная S-расширенная грамматика G'.

Наличие ε-правил не является критичным для описания грамматики, но может усложнять процесс построения распознавателя для языка, определяемого этой грамматикой. В общем случае наличие ε-правил не является необходимым, хотя они могут быть получены в результате формальных преобразований. Удаление ε-правил для грамматики G позволяет упрощать процесс построения распознавателя для языка L(G), однако множество P правил грамматики может существенно усложниться.

Для заданной КС-грамматики G нетерминальный символ A называется nullable, если $A \Rightarrow^* \varepsilon$, иначе говоря, из A выводима пустая цепочка. Множество Nullable для грамматики G включает в себя все нетерминалы, из которых выводимы пустые цепочки.

Алгоритм 4.5 Построение множества Nullable для нетерминалов грамматики
Вход грамматика G = (VT, VN, P, S) и A ∈ VN
Выход множество Nullable(A)

```
Nullable = ∅
flag = 1
while(flag){
            flag = 0
            foreach A → α in P {
                if (α = ε)
                    {Nullable.add(A); flag = 1}

                if (α ∈ VN⁺ && forall Bi in α : Bi ∈ Nullable )
                    {Nullable.add(A); flag = 1}
    }}
```

После того, как было построено множество Nullable для нетерминальных символов грамматики, можно приступать к устранению ε-правил.

Алгоритм 4.6 Удаление ε-правил
Вход расширенная КС-грамматика G = (VT, VN, P, S)
Выход КС-грамматику G' = (VT, VN, P', S) без ε-правил

Шаг 1. Построение множества Nullable см. Алгоритм 4.5; P' = ∅
Шаг 2. P' = P - {(A→ε) ∈ P для всех A ∈ VN }.
Шаг 3. Если S ∈ Nullable поместить S → ε в P'
Шаг 4. Для всех оставшихся правил из P' вида заменить каждое правило A → $x_1 \ldots x_n$ для любых n>0 всеми возможными правилами следующего вида: A → $y_1 \ldots y_n$, где:

$y_i = x_i$ or $y_i = \varepsilon$ для всех x_i в правой части правила $\{x_1 \ldots x_n\}$ которые принадлежат Nullable

$y_i = x_i$ для всех x_i в правой части правила $\{x_1 \ldots x_n\}$ которые не принадлежат Nullable

Пусть задана исходная расширенная КС-грамматика G = (VT, VN, P, S), требуется построить эквивалентную ей S-расширенную КС-грамматику G' = (VT, VN, P', S), в которой правило S → ε присутствует тогда и только тогда ε ∈ L(G).

На первом шаге алгоритма 4.6 строится множество Nullable для нетерминальных символов грамматики. На втором шаге в множество правил P' помещаются все правила, кроме ε-правил. Шаг три выполняется только при условии наличия в P правила S → ε. На четвертом шаге необходимо пополнить множество P' правилами, которые получаются путем удаления из их правой части всех возможных комбинаций нетерминальных символов, принадлежащих множеству Nullable.

Пример 4.8.
Рассмотрим в качестве исходной грамматику G = (VN, VT, P, S), где:

VN = {S, A, B, C}
VT = {a, b, c}

P = {S → ACA
A → aAa | B | C
B → bB | b
C → cC | ε}

Шаг 1. Построение множество Nullable

Итерация	Множество Nullable	
0	∅	На четвертом шаге не было добавлено ни одного нового символа в Nullale.
1	{C}	
2	{A, C}	
3	{A, C, S}	
4	{A, C, S}	

Шаг 2.	Шаг 3.
P' = S → ACA A → aAa | B | C B → bB | b C → cC	Добавляем S → ε в P'.

Шаг 4.

Исходный набор правил P'	Результирующий набор правил P'
S → ε | ACA	S → ε | ACA | AA | AC | CA | A | C
A → aAa | B | C	A → aAa | aa | B | C
B → bB | b	B → bB | b
C → cC	C → cC | c

4.4.4 Удаление цепных правил.

Цепные правила удлиняют процесс вывода сентенциальных форм, усложняют грамматику и не являются обязательными. Цепное правило A → B показывает, что любая цепочка, выводимая из B, может быть выведена и из A. Идею удаления цепных правил рассмотрим на следующем примере:

Рассмотрим следующие правила	Цепное правило	Замена правой части правила A → B на все возможные правые части правил с B в левой части правила
A → aA | a | B B → bB | b | C	A → B	A → aA | a | bB | b | C B → bB | b | C

Пусть задана КС-грамматика G = (VT, VN, P, S) без ε-правил. Требуется построить грамматику G' = (VT, VN, P', S) без цепных правил.

Множество P’ содержит все не цепные правила из P, вместе со всеми подстановками вида $A \to \alpha$, если $A \Rightarrow * B$ посредством только цепных правил, и $B \to \alpha$.

Алгоритм 4.7 Построение множества Chain для нетерминала A из неукорачивающейся КС-грамматики G = (VT, VN, P, S)
Вход КС-грамматика G = (VT, VN, P, S) и нетерминал A
Выход Множество Chain(A)

```
Chain(A) = {A}
Prev = ∅
while(Chine(A) != Prev){
        Temp = Chain(A) - Prev
        Prev = Chain(A)
        foreach B in Temp
              foreach B → C in P
                    Chain(A).add(C)
}
```

Вывод $A \Rightarrow * C$, содержащий только цепные правила, назовем цепным. Для каждого нетерминала из грамматики G найдем множество всех нетерминальных символов, для которых существует цепной вывод. Обозначим это множество Chain(A), где $A \in VN$. Алгоритм 4.7 строит множество Chain для нетерминального символа.

Вспомогательное множество Temp содержит нетерминальные символы, которые были добавлены в множество Chain на предыдущем шаге работы алгоритма.

Множество нетерминальных символов, принадлежащих Chain(A), определяет замены, которые необходимо сделать для удаления цепных правил, в левой части которых находится нетерминальный символ A.

Следующий алгоритм позволяет получить грамматику G’ без цепных правил, эквивалентную грамматике G.

Алгоритм 4.8 Построение КС-грамматики без цепных правил
Вход: неукорачивающаяся (без ε-правил) $G = (VT, VN, P, S)$
Выход: $G' = (VT, VN, P', S)$ без цепных правил

```
P' = P - {(A → α) ∈ P | ∀ α ∉ VN}      //все кроме цепных правил
foreach A in VN{
    if (Chain(A) != {A}){
        foreach B in Chain(A)
            foreach B → α  in P
                if (α  ∉ VN )
                    P'.add(A → α)}
}
```

Пример 4.9.
Рассмотрим в качестве исходной КС-грамматику
$G = (\{a, b, c\}, \{A, B, C, S\}, P, S)$ $P = \{ S \to ACA \mid AA \mid AC \mid CA \mid A \mid C$
$A \to aAa \mid aa \mid B \mid C$
$B \to bB \mid b$
$C \to cC \mid c \}$

Действие	Множество P’
$P' = P - \{(A \to \alpha) \in P$ для всех $\alpha \notin VN\}$	$S \to ACA \mid AA \mid AC \mid CA$ $A \to aAa \mid aa$ $B \to bB \mid b$ $C \to cC \mid c$
Chain(S) = {S, A, C, B} Добавляем правила $S \to aAa \mid aa$ $S \to bB \mid b$ $S \to cC \mid c$	$S \to ACA \mid AA \mid AC \mid CA \mid aAa \mid aa \mid bB \mid b \mid cC \mid c$ $A \to aAa \mid aa$ $B \to bB \mid b$ $C \to cC \mid c$
Chain(A) = {A, C, B} Добавляем правила $A \to bB \mid b$ $A \to cC \mid c$	$S \to ACA \mid AA \mid AC \mid CA \mid aAa \mid aa \mid bB \mid b \mid cC \mid c$ $A \to aAa \mid aa \mid bB \mid b \mid cC \mid c$ $B \to bB \mid b$ $C \to cC \mid c$
Chain(B) = {B} Chain(C) = {C}	Без изменений Без изменений

В результате множество правил результирующей грамматики имеет следующий вид:

$S \rightarrow ACA \mid AA \mid AC \mid CA \mid aAa \mid aa \mid bB \mid b \mid cC \mid c$
$A \rightarrow aAa \mid aa \mid bB \mid b \mid cC \mid c$
$B \rightarrow bB \mid b$
$C \rightarrow cC \mid c$

Удаление цепных правил приводит к общему увеличению числа правил грамматики, но упрощает вывод её сентенциальных форм.

4.4.5 Нормальная форма Хомского

Для любой приведенной неукорачивающейся КС-грамматики без цепных правил можно получить эквивалентную ей грамматику в нормальной форме Хомского (Chomsky Normal Form). Правила в нормальной форме Хомского имеют следующий вид:

$A \rightarrow BC$ где $A, B, C \in VN$
$A \rightarrow a$ где $A \in VN$, $a \in VT$

Пусть задана исходная КС-грамматика $G = (VT, VN, P, S)$, требуется построить эквивалентную ей КС-грамматику $G' = (VT, VN', P', S)$ в нормальной форме Хомского.

Неформально этот процесс можно описать следующим образом. В множество правил P' переносятся все правила из P, длина правой части которых меньше трех. Остальные правила будут иметь следующий вид: $A \rightarrow a_1a_2\alpha$, где $A \in VN$; $a_1, a_2 \in (VT \cup VN')$; $\alpha \in (VT \cup VN')^+$. Каждое такое правило будут заменено парой правил $A \rightarrow a_1B$ и $B \rightarrow a_2\alpha$, причем новый нетерминальный символ B будет добавлен в VN'. Замены будут продолжаться до тех пор, пока длина правых частей всех правил в P не станет равна двум. После этого все правила из P добавляются в P'.

Далее, если в правой части правила есть терминальный символ, например $A \rightarrow cB$, то в грамматику вводится новый нетерминал C, новое правило $C \rightarrow c$, а правило $A \rightarrow cB$ заменяют на правило $A \rightarrow CB$. Это действие выполняется для всех правил грамматики, длина правой части которых равна двум, и правая часть содержит нетерминальный (один или два) символа.

Алгоритм 4.9 Построение грамматики в нормальной форме Хомского
Вход КС-грамматика G = (VT, VN, P, S)
Выход КС-грамматика G' = (VT, VN', P', S) в НФХ

```
P' = P - {(A → α) ∈ P | |α| ≤ 2}
flag = 1
while(flag){
    flag = 0
    foreach A → a1a2α in P{
        P.delete(A → a1a2α)          // сокращаем длину правил
        P.add(A → a1B)               // исходной грамматики
        P.add(B → a2α)
        VN'.add(B)
        flag = 1}
        }
P' = P' ∪ P
foreach A → α in P{
    if (|α|==2)                      // A →xy | x, y ∈ (VT ∪ VN')
      foreach a in α
        if( a ∈ VT ) {
          P'.delete(A → α)
          VN.add(B)
          P'.add(B → a)
          replace (a,B) in α
          P'.add(A → α ); flag = 1}}
    }
```

При преобразовании грамматики к нормальной форме Хомского количество правил и нетерминальных символов в грамматике увеличивается. Это, возможно, приведет к затруднению ее восприятия. Однако цель преобразования состоит в упрощении конструирования распознавателя языка на ее основе, что будет показано ниже, в процессе обсуждения построения распознавателей для КС-языков.

Пример 4.10. Рассмотрим работу алгоритма 4.9 на примере следующей грамматики.

Исходная грамматика	Грамматика в НФХ	
$S \to A+T \mid b \mid (A)$	$S \to AY \mid b \mid LZ$	Y соответствует +T
$A \to A+T \mid b \mid (A)$	$Z \to AR$	Z соответствует A)
$T \to b \mid (A)$	$A \to AY \mid b \mid LZ$	L соответствует (
	$T \to b \mid LZ$	R соответствует)
	$Y \to PT$	P соответствует +
	$P \to +$	
	$L \to ($	
	$R \to)$	

4.4.6 Нормальная форма Грейбах

Любой непустой КС-язык, без ε-правил, L(G) может быть задан грамматикой G = (VT, VN, P, S), все правила которой из P имеют вид: $A \to a\alpha$, где $a \in VT$, $A \in VN$, и $\alpha \in VN^*$. Грамматики такого вида называются грамматиками в *нормальной форме Грейбах*, (Sheila Greibach).

Поскольку в каждую правую часть правила грамматики входит один терминальный символ, то каждое применение правила в процессе вывода добавляет к сентенциальной форме ровно один терминальный символ, и, следовательно, предложение длины n порождается в точности за n шагов. Кроме того, если по грамматике в нормальной форме Грейбах построить МП-автомат из п. 4.3, то получается МП-автомат без ε - переходов, это показывает, что такие переходы в МП-автомате всегда можно удалить.

Преобразование КС-грамматики к нормальной форме Грейбах достаточно сложно, алгоритм преобразования представлен в [19], а состав правил существенно отличается от правил исходной произвольной КС-грамматики.

Например, для грамматики

$E \to E + T$

$T \to T \times F \mid F$

$F \to (E) \mid a$

нормальная форма Грейбах будет иметь следующий вид:

$E \to (ERMF \mid aMF \mid (ERYMF \mid aYMF \mid (ER \mid (ERMFZ \mid$
$aMFZ \mid (ERYMFZ \mid aYMFZ \mid (ERZ \mid aZ \mid a$

$Z \to +T \mid +TZ$

$T \to (ER \mid (ERY \mid aY \mid a$
$Y \to \times F \mid \times FY$
$F \to (ER \mid a$
$M \to \times$
$R \to)$

4.4.7 Устранение прямой левой рекурсии

Рассмотрим КС-грамматику $G = (VT, VN, P, S)$. Правило грамматики из P называется леворекурсивным, если оно имеет следующий вид: $A \to A\alpha$, $A \in VN$, и $\alpha \in V^*$. КС-грамматика называется леворекурсивной, если в множество P входит хотя бы одно леворекурсивное правило.

Для любой леворекурсивной грамматики $G = (VT, VN, P, S)$ можно построить эквивалентную ей КС-грамматику $G = (VT, VN', P', S)$ без леворекурсивных правил.

Алгоритм 4.10 Устранение левой рекурсии
Вход грамматика $G = (VT, VN, P, S)$ с леворекурсивными правилами
Выход грамматика $G = (VT, VN', P', S)$ без левой рекурсии

---

foreach A **in** VN **{**

Шаг 1 Рассмотрим все правила грамматики с A в левой части.
$A \to A\alpha_1 \mid ... \mid A\alpha_n$ (правила с левой рекурсией)
$A \to \beta_1 \mid ... \mid \beta_m$ (правила без левой рекурсии)

Шаг 2 Добавим новый нетерминал B в множество P.
Заменим все правила с A в левой части следующими:
$A \to \beta_1 \mid \beta_2 \mid ... \mid \beta_m$
$A \to \beta_1 B \mid \beta_2 B \mid ... \mid \beta_m B$
$B \to \alpha_1 \mid \alpha_2 \mid ... \mid \alpha_n$
$B \to \alpha_1 B \mid \alpha_2 B \mid ... \mid \alpha_n B$

}

В результате работы этого алгоритма в грамматике могут появиться цепные правила.

Пример 4.11.

Рассмотрим леворекурсивную грамматику:

$S \to SA \mid a$

$A \to a$

После удаления левой рекурсии грамматика примет следующий вид:

$S \to a \mid aB$

$B \to A \mid AB$

$A \to a$

Правило $B \to A$ является цепным, и его можно удалить из грамматики. Результирующая грамматика без левой рекурсии и цепных правил:

$S \to a \mid aB$

$B \to a \mid AB$

$A \to a$.

Алгоритм 4.10 основан на применении правила Ардена [15].

В общем случае грамматика G = (VT, VN, P, S) будет леворекурсивной если существует нетерминал A, такой, что $S \Rightarrow^* \alpha A\beta$ и $A \Rightarrow^+ A\gamma$ для некоторых $\alpha, \beta, \gamma \in (VT \cup VN)^*$.

4.4.8 Левая факторизация

В некоторых случаях, например для нисходящего разбора, наличие в множестве правил грамматики для одного и того же нетерминала в левой части двух и более альтернативных правых частей правил, начинающихся с одинаковых подцепочек, может затруднить процесс построения распознавателя.

Рассмотрим КС-грамматику G = (VT, VN, P, S), такую, что в множестве P есть правила вида $A \to \alpha\beta_1 \mid \alpha\beta_2$ с одинаковыми префиксами, тогда, введя дополнительный нетерминал B, исходные правила можно заменить правилами $A \to \alpha B$; $B \to \beta_1 \mid \beta_2$. На этой идее основан алгоритм левой факторизации.

Представленный ниже алгоритм применяется к множеству правил грамматики до тех пор, пока не будет добавлено ни одного нового правила.

Алгоритм 4.11 Левая факторизация

Вход грамматика G = (VT, VN, P, S)

Выход грамматика G = (VT, VN’, P’, S) без одинаковых префиксов

---

foreach A **in** VN**{**

Шаг 1 Для правил вида $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid ... \mid \alpha\beta_m \mid \gamma_1 \mid ... \mid \gamma_n$

Выделить самый длинный префикс α, и если $\alpha \neq \varepsilon$ переходим к шагу 2.

Шаг 2 Вводим новый нетерминал B и заменяем правила грамматики

$B \to \beta_1 \mid \beta_2 \mid ... \mid \beta_m$

$A \to \alpha B \mid \gamma_1 \mid ... \mid \gamma_n$

}

Пример 4.12

Рассмотрим грамматику S → aSb | aSc | d.

Для правил S → aSb | aSc выделяем префикс a.

Вводим новый нетерминал A.

Заменяем правила грамматики:

S → aSA | d

A → b | c

4.5 Распознаватели для контекстно-свободных языков

Пусть G = (VT, VN, P, S) – КС-грамматика, а α – цепочка из терминальных символов этой грамматики. Решение задачи о принадлежности этой цепочки языку, порождаемому данной грамматикой, может быть реализовано двумя способами:

- построение распознавателя, осуществляющего разбор цепочки сверху-вниз, что соответствует построению синтаксического дерева от корня к листьям, последовательно применяя правила грамматики для получения анализируемой цепочки – *нисходящий разбор*;
- построение распознавателя, осуществляющего разбор цепочки снизу-вверх, когда по заданной цепочке символов осуществляется свертка цепочки к начальному символу грамматики, или построению синтаксического дерева от листьев к начальному символу грамматики – *восходящий разбор*.

В обоих случаях задача разбора будет считаться выполненной, если удастся построить синтаксическое дерево (допуск цепочки). В противном случае будет принято решение, что анализируемая цепочка терминальных символов не является предложением языка, порожденного грамматикой G (цепочка отвергнута).

Нисходящий разбор соответствует построению по заданному предложению синтаксического дерева, начиная с начального символа

грамматики, так, как это показано на рис. 4.1. Нисходящий разбор соответствует левостороннему выводу. В случае нисходящего разбора на каждом шаге необходимо заменить самый левый нетерминальный символ сентенциальной формы, начиная с цепочки, содержащей единственный символ S – начальный символ грамматики – правой частью правила, связанного с ним. Если такому нетерминалу соответствует несколько альтернативных правых частей правил грамматики, то необходимо принять решение относительно того, какая из альтернатив должна быть выбрана.

Восходящий разбор обычно соответствует правостороннему выводу. В случае восходящего разбора в анализируемой цепочке нужно найти подцепочку, которая соответствует правой части одного из правил грамматики, и заменить её на нетерминальный символ из левой части этого правила, после чего разбор продолжается, и либо в результате его работы будет получен начальный символ грамматики, что соответствует успеху построения синтаксического дерева, либо на текущем шаге не удастся найти подцепочку для свертки, что соответствует тому, что анализируемая цепочка не принадлежит языку, порождаемому исходной грамматикой. Если в анализируемой подцепочке есть несколько подцепочек, соответствующих правым частям правил (возможно, одного и того же правила), возникает необходимость выбора одной из них.

4.5.1 Распознаватели с возвратами

Распознаватели с возвратами являются наиболее простым методом конструирования распознавателей для КС-языков. Логика их работы основана на моделировании недетерминированного МП-автомата.

Поскольку моделируется недетерминированный МП-автомат, то на некотором шаге работы моделирующего алгоритма возможно несколько переходов из текущей конфигурации МП-автомата, тогда на каждом шаге работы алгоритм должен запоминать все возможные альтернативные состояния МП-автомата, выбирать одно из них, перейти в это состояние и действовать так до тех пор, пока либо не будет достигнуто допускающее состояние автомата, либо автомат не перейдет в такую конфигурацию, когда следующее состояние будет не определено. Если достигнуто одно из конечных состояний — входная цепочка допущена, работа алгоритма завершается. В противном случае

алгоритм должен вернуть автомат на несколько шагов назад, когда был возможен выбор одного из альтернативных состояний, выбрать альтернативу и продолжить моделирование поведения МП-автомата. Алгоритм завершается ошибкой, когда все возможные варианты работы МП-автомата перебраны и ни одно из возможных допускающих состояний не было достигнуто.

МП-автоматы из примеров 4.4 и 4.5 осуществляют алгоритм разбора с возвратами, причем МП-автомат из примера 4.4 осуществляет разбор методом сверху-вниз, а МП-автомат из примера 4.5 – методом снизу вверх.

Время работы распознавателя с возвратами имеет экспоненциальную зависимость от длины анализируемой цепочки, что существенно ограничивает применимость таких алгоритмов.

Несмотря на то, что распознаватели с возвратами могут быть реализованы для языка, порождаемого любой КС-грамматикой, их подробное рассмотрение не имеет смысла в силу их неприменимости для реализации фазы синтаксического анализа компиляторов языков программирования.

Другим классом универсальных распознавателей для КС-языков являются табличные распознаватели.

4.5.2 Алгоритм Кока-Янгера-Касами

Алгоритм Кока-Янгера-Касами (*CYK algorithm*) позволяет определить, является ли цепочка символов предложением языка, порождаемого КС-грамматикой. Предварительно КС-грамматика должна быть преобразована к нормальной форме Хомского (п. 4.4.5.). В силу того, что любая КС-грамматика может быть преобразована к нормальной форме Хомского без ε-правил, данный алгоритм универсален и позволяет выполнить распознавание для любой КС-грамматики. Пусть $G = (VT, VN, P, S)$ – КС-грамматика в нормальной форме Хомского. Необходимо проверить, является ли цепочка $\alpha = a_1a_2...a_n$, где $S \Rightarrow* \alpha$, и $a_i \in VT$, предложением языка $L(G)$.

На вход алгоритма подается цепочка терминальных символов грамматики длиной n, в процессе работы алгоритма строится треугольная *распознающая матрица* размером $n \times n$, как это показано на рис 4.6. Элемент таблицы на пересечении i-й строки и j-го столбца содержит множество всех нетерминальных символов, из которых

может быть выведена подцепочка $a_j a_{j+1} \ldots a_{j+i-1}$, причем эта подцепочка длиной i, а первый символ этой подцепочки находится в позиции j.

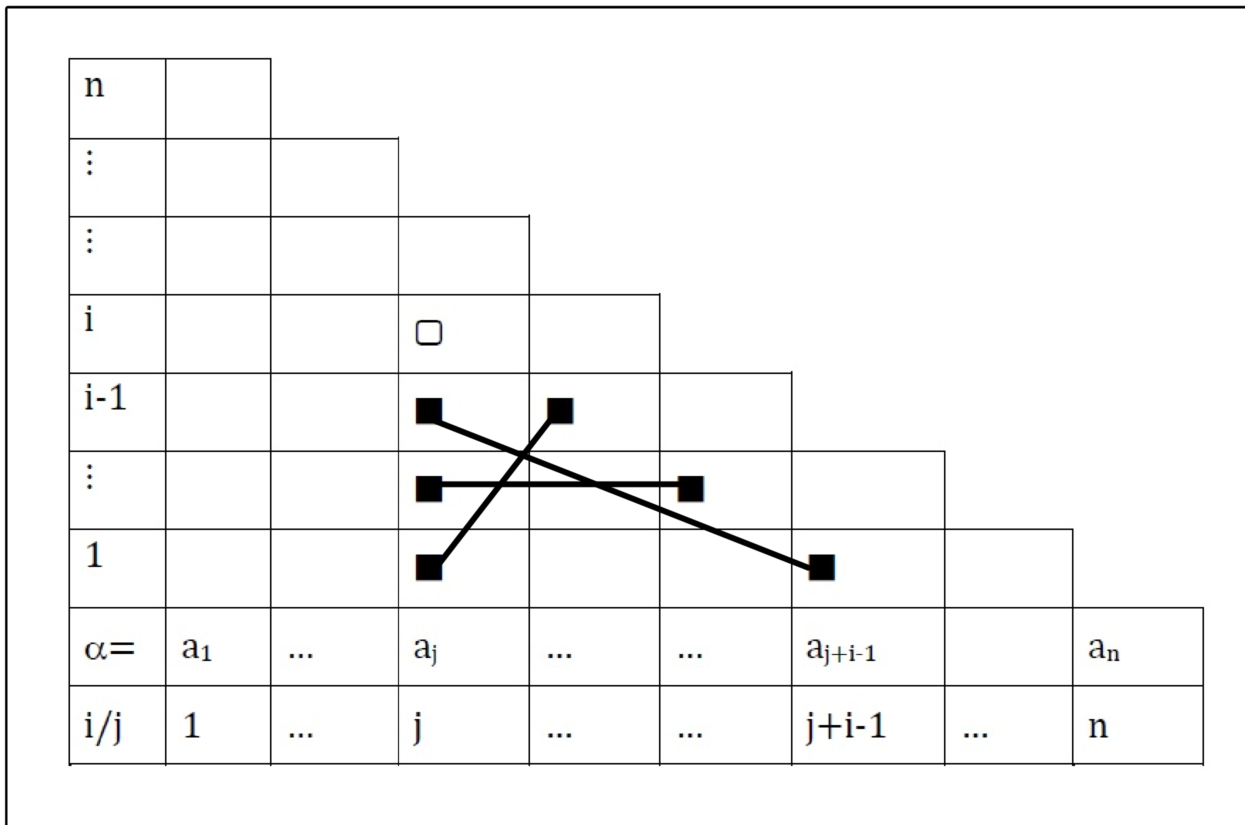


Рис. 4.6 Построение матрицы разбора для алгоритма Кока-Янгера-Касами.

Все нетерминалы в строке i =1 порождают цепочки, длина которых равна единице. Нетерминалы во второй строке порождают цепочки длиной 2, так далее. Цепочка принадлежит языку L(G), тогда и только тогда, когда в позиции <n,1> будет помещен символ S.

В общем случае в распознающей матрице мы будем помещать нетерминальный символ A в i-ю строку и j-й столбец, иначе в позицию (i, j), для $i = 1, \ldots, n$, и $j = 1, \ldots, n-(i-1)$, тогда и только тогда, когда:

A → BC и символ B есть в позиции (1, j) и C есть в позиции (i−1, j +1) или
A → BC и символ B есть в позиции (2, j) и C есть в позиции (i−2, j +2) или
… или …
A → BC и символ B есть в позиции (i−1, j) и C есть в позиции (1, j +i −1).

Рисунок 4.6 демонстрирует эту идею.

Рассмотрим КС-грамматику в НФХ G= ({S, A, B, C, D, E}, {a, b}, S, P), где множество P имеет следующий вид:

$S \rightarrow CB \mid FA \mid FB$
$A \rightarrow CS \mid FD \mid a$
$B \rightarrow FS \mid CE \mid b$
$C \rightarrow a$ $D \rightarrow AA$
$E \rightarrow BB$ $F \rightarrow b$

На вход алгоритма поступает цепочка $\alpha = aababb$. Для неё строится распознающая матрица, показаная на рис. 4.7. Имеется следующее соответствие между символами цепочки α их позициями:

α:	a	a	b	a	b	b
позиция:	1	2	3	4	5	6

В данной цепочке мы можем выделить подцепочку bab длиной три символа, голова которой находится в третьей позиции, и подцепочку ab длиной в два символа, голова которой находится в четвертой позиции. В распознающей матрице в позицию <1, j> для j = 1, 2, … n помещается нетерминальный символ A тогда и только тогда, когда для терминального символа a_j существует правило грамматики $A \rightarrow a_j$. В соответствии с этим в позицию <1, 3> помещаются символы F и B в соответствии с правилами $F \rightarrow b$ и $B \rightarrow b$, а в позицию <1, 4> помещаются символы A и C в соответствии с правилами $A \rightarrow a$ и $C \rightarrow a$. Для позиции <2, 3> подходят все нетерминалы, для которых справедливо $A \Rightarrow^* ba$. Результирующая матрица показана на рис. 4.7.

6	D, **S**					
5	A	A, B				
4	D	S	S, E			
3	A	A	B	A, B		
2	D	S	S	S	E, S	
i=1	A, C	A, C	B, F	A, C	B, F	B, F
α=	a	a	b	a	b	b
	j=1	2	3	4	5	6

Рис. 4.7 Распознающая матрица для цепочки aababb.

Временная сложность алгоритма Кока-Янгера-Касами определяется временем построения матрицы распознавания и вычисляется следующим образом. Пусть n – длина цепочки символов, которую нужно разобрать, и G = (VT, VN, P, S) – КС-грамматика в нормальной форме Хомского, тогда

- обозначим S_a множество терминальных символов, причем $S_a \subset VT$; требуется одна единица времени, чтобы найти максимальное подмножество $S_A \subset VN$, такое что для каждого $A \in S_A$ существует $a \in S_a$ и $A \to a \in P$;
- обозначим через S_B и S_C два подмножества VN; требуется одна единица времени для нахождения максимального подмножества S_A ($S_A \subset VN$) такого, что для каждого $A \in S_A$ существуют $B \in S_B$, $C \in S_C$ и $A \to BC \in P$;
- любая другая операция занимает 0 единиц времени.

Тогда:

- для построения первой строки распознающей матрицы требуется n единиц времени;
- для построения второй строки распознающей матрицы требуется (n – 1) × 1 времени;
- для i – й строки ($2 \geq i \geq n$) требуется $(n - i + 1) \times (i - 1)$ единиц времени,
- тогда время, которое требуется для построения распознающей матрицы, может быть определено следующим образом:

$$n + \sum_{i=2}^{n}(n-i+1)(i-1) = (n^3 + 5n)/3.$$

Это равенство показывает, что временная сложность алгоритма Кока-Янгера-Касами имеет порядок $O(n^3)$.

Алгоритм Кока-Янгера-Касами является *универсальным распознавателем* для КС-грамматик, включая неоднозначные, осуществляющий разбор методом снизу вверх.

Метод разбора, основанный на *алгоритме Эрли,* также является универсальным, он осуществляет разбор методом сверху вниз, и временная сложность его работы составляет $O(n^3)$. Подробно алгоритм Эрли рассмотрен в [8, 17].

Наибольший интерес представляют те алгоритмы реализации распознавателей, которые осуществляют обработку исходной цепочки символов за один проход, иначе говоря, время работы таких распознавателей растет линейно по отношению к длине

обрабатываемой цепочки. Обычно подобные алгоритмы не являются универсальными, а значит, на грамматики, для которых будут реализовываться алгоритмы, будут накладываться ограничения. С одной стороны, это усложняет процесс создания порождающей грамматики, с другой стороны, если такую грамматику удалось построить, то время работы распознавателя существенно уменьшается. Ниже будут рассмотрены только подобные алгоритмы.

4.6 Нисходящие распознаватели КС-языков

Нисходящие распознаватели пытаются для заданной цепочки терминальных символов построить синтаксическое дерево вывода этой цепочки, начиная от вершины дерева и двигаясь к листьям, раскрывая всякий раз самый левый нетерминал сентенциальной формы. Основная проблема заключается в том, что если с самым левым нетерминалом связаны несколько правых частей правил грамматики, то для распознавания за линейное время эти правила должны иметь такой вид, чтобы выбор альтернативы был единственным. Следовательно, такие нисходящие распознаватели не являются универсальными и могут обрабатывать ограниченные классы КС-грамматик.

4.6.1 Метод рекурсивного спуска

Метод рекурсивного спуска реализует разбор цепочки сверху-вниз следующим образом: для каждого нетерминального символа грамматики создается процедура, носящая его имя. Задача этой процедуры – начиная с указанного места исходной цепочки найти подцепочку, которая выводится из этого нетерминала.

Если такую подцепочку найти не удается, то процедура завершает свою работу вызовом процедуры обработки ошибок, которая выдает сообщение о том, что цепочка не принадлежит языку грамматики, и останавливает разбор. Если подцепочку удалось найти, то работа процедуры считается нормально завершенной, и осуществляется возврат в точку вызова. Тело каждой такой процедуры составляется непосредственно по правилам вывода соответствующего нетерминала, при этом терминалы распознаются самой процедурой, а нетерминалам соответствуют вызовы процедур, носящих их имена.

Рассмотрим следующую грамматику: G = ({S, B, G}, {a, b, (,)}, S, P)

P= {S → bBb, B → (G | a, G → Ba) }

тогда процедуры для распознавателя методом рекурсивного спуска будут иметь следующий вид:

```
str1 = "bab#"
pointer = 0

def getchar():
    global pointer
    global str1
    c = str1[pointer]
    pointer += 1
    return c

def S():
    c = getchar()
    if c != 'b':
        Error()
    B()
    c = getchar()
    if c != 'b':
        Error()
    c = getchar()
    if c != '#':
        Error()

def main():
    S()
    print("Success!")
```

```
def B():
    c = getchar()
    if c == '(':
        G()
    elif c != 'a':
        Error()

def G():
    B()
    c = getchar()
    if c != 'a':
        Error()
    c = getchar()
    if c != ')':
        Error()

def Error():
    print("No!")
    sys.exit()
```

Символ # используется в качестве концевого маркера, которому может соответствовать конец строки или файла. Процедура getchar() осуществляет чтение текущего символа из анализируемой цепочки, а процедура Error() прерывает выполнение работы распознавателя. Для разбора цепочки, заданной в str1, достаточно вызвать процедуру, соответствующую начальному символу грамматики. Данный метод прост в реализации, однако существуют ограничения на состав и тип правил грамматики, для которой может быть применен метод рекурсивного спуска.

4.6.2 Применимость метода рекурсивного спуска

Метод рекурсивного спуска применим к грамматике, если правила вывода грамматики имеют один из следующих видов:

- $A \rightarrow \alpha$ где $\alpha \in (VT \cup VN)^*$, и это единственное правило для нетерминала A;
- $A \rightarrow a_1\alpha_1 \mid a_2\alpha_2 \mid \ldots \mid a_n\alpha_n$, где $a_i \in VT$ $a_i \neq a_j$ $i \neq j$ $\alpha_i \in (VT \cup VN)^*$ если для нетерминала A существует несколько альтернативных правых частей правил вывода, то они должны начинаться с терминальных символов, причем эти терминальные символы должны быть различными.

Изложенные выше ограничения являются достаточными, но не необходимыми. Можно применить эквивалентные преобразования КС-грамматик, которые способствуют приведению грамматики к требуемому виду, но не гарантируют его достижения. Подробнее см. [4].

Если в грамматике есть нетерминалы, правила вывода которых леворекурсивны, т. е. имеют вид:

$A \rightarrow A\alpha_1 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$, где

$\alpha_i \in (VT \cup VN)^+$ для $i = 1, 2, \ldots, n$;

$\beta_i \in (VT \cup VN)^*$

то левую рекурсию можно заменить правой:

- $A \rightarrow \beta_1 A' \mid \ldots \mid \beta_m A'$
- $A' \rightarrow \alpha_1 A' \mid \ldots \mid \alpha_n A' \mid \varepsilon$

см. удаление прямой левой рекурсии п. 4.4.7.

В результате будет получена грамматика, эквивалентная данной, так как из нетерминала A по-прежнему выводятся цепочки вида $(b_j)(\alpha_i)^*$, где $i = 1, 2, \ldots, n$; $j = 1, 2, \ldots, m$.

Если в грамматике есть нетерминал, у которого несколько альтернативных правил начинаются одинаковыми терминальными символами, т. е. имеют вид

$A \rightarrow a\alpha_1 \mid a\alpha_2 \mid \ldots \mid a\alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$

где $a \in VT$; $\alpha_i, \beta_j \in (VT \cup VN)^*$;

β_j не начинается с a, $i = 1, 2\ldots, n$, $j = 1, 2, \ldots, m$,

то можно преобразовать правила вывода данного нетерминального символа, объединив правила с общими началами в одно правило:

$A \rightarrow aA' \mid \beta_1 \mid \ldots \mid \beta_m$

$A' \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$

В результате будет получена грамматика, эквивалентная исходной. См. п. 4.4.8. – левая факторизация.

Если в грамматике есть нетерминальный символ с несколькими альтернативными правыми частями, и среди них есть альтернативы, начинающиеся нетерминальными символами, т. е. имеющие вид

$A \rightarrow B_1\alpha_1 \mid B_2\alpha_2 \mid ... \mid B_n\alpha_n \mid a_1\beta 1 \mid ... \mid a_m\beta_m$

$B_i \rightarrow \gamma_{i1} \mid ... \mid \gamma_{ik}$

где $B_i \in VN$; $a_j \in VT$ $\alpha_i, \beta_j, \gamma_{ij} \in (VT \cup VN)^*$, то можно заменить вхождения нетерминальных символов B_i соответствующими им правыми частями правил (попытка устранения косвенной левой рекурсии) в надежде, что правило для нетерминального символа A станет удовлетворять требованиям метода рекурсивного спуска:

$A \rightarrow \gamma_{11}\alpha_1 \mid ... \mid \gamma_{1k}\alpha_1 \mid ... \mid \gamma_{n1}\alpha_n \mid ... \mid \gamma_{np}\alpha_n \mid a_1\beta 1 \mid ... \mid a_m\beta_m$

Если в исходной, или в преобразованной грамматике были, или появились в результате преобразования, ε-правила, то применимость метода рекурсивного спуска является проблематичной. Рассмотрим это подробнее.

Введем понятия множеств FIRST и FOLLOW для символов КС-грамматики. Множество FIRST(A) – это множество терминальных символов, с которых начинаются цепочки, выводимые из A.

Рассмотрим сентенциальную форму $S \Rightarrow^* \alpha Ab\beta$ ($A \Rightarrow^+ c\gamma$) рис. 4.8.

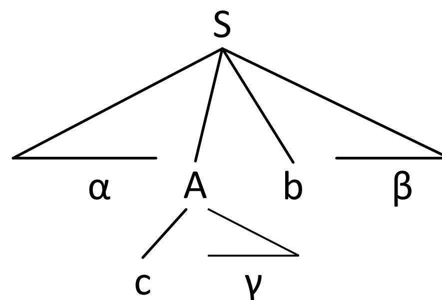


Рис. 4.8 Терминальный символ $c \in FIRST(A)$;
терминальный символ $b \in FOLLOW(A)$.

В алгоритме построения множества FIRST доля нетерминалов 4.12 используется функция isnullable, возвращающая true, если из нетерминального символа выводима пустая цепочка, и false в противном случае. Данная функция может быть реализована на основе алгоритма 4.5.

В том случае, если $A \Rightarrow^* \varepsilon$, то $\varepsilon \in FIRST(A)$, и значит $FIRST(\varepsilon) = \{ \varepsilon \}$.

Алгоритм 4.12 Построение множества FIRST
Вход: грамматика G = (VT, VN, P, S)
Выход: множества FIRST для нетерминалов грамматики G

```
foreach X in VN  do FIRST(X) = {};
foreach  t in VT do FIRST(t) = {t};
flag  = 1
while(flag){
      flag = 0
      foreach A → b1 … bk in P {
            FIRST(A) = FIRST(A) ∪ FIRST(b1);
            for i = 1 to k-1 do
                  if isnullable(bi)
                        FIRST(A) = FIRST(A) ∪ FIRST(bi+1);
                  else break;
                  if FIRST(A) изменилось, то flag = 1
            }
}
```

Пример 4.13

Рассмотрим грамматику G_1 = ({S, A, T}, {b, +, (,), #}, S, P), где множество P имеет следующий вид:

S → A

A → T | A + T

T → b | (A)

Данная грамматика является неоднозначной в силу правила A → A + T, содержащего прямую левую рекурсию. Воспользовавшись алгоритмом 4.10, прямую левую рекурсию можно устранить, тогда грамматика примет следующий вид:

G_2 = ({S, A, T, Z}, {b, +, (,), #}, S, P), где множество P:

S → A

A → T | TZ

Z → +T | +TZ

T → b | (A)

Грамматику G_2 также нельзя непосредственно использовать для построения распознавателя методом рекурсивного спуска, ввиду наличия для нетерминалов A и Z альтернативных правых частей

правил, начинающихся с одинаковых символов. После осуществления левой факторизации грамматика принимает следующий вид:

$G_3 = (\{S, A, B, T, Y, Z\}, \{b, +, (,), \#\}, S, P)$,

где множество P:

$S \rightarrow A$

$A \rightarrow TB$

$B \rightarrow Z \mid \varepsilon$

$Z \rightarrow +TY$

$Y \rightarrow Z \mid \varepsilon$

$T \rightarrow b \mid (A)$

Данная грамматика является однозначной; построим множество FIRST для её нетерминалов. Пошаговый процесс построения этих множеств показан на рис. 4.9.

Номер шага	FIRST(S)	FIRST(A)	FIRST(B)	FIRST(Z)	FIRST(Y)	FIRST(T)
0	∅	∅	∅	∅	∅	∅
1	∅	∅	{ ε }	{ + }	{ ε }	{ b, (}
2	∅	{ b, (}	{ ε, + }	{ + }	{ ε, + }	{ b, (}
3	{ b, (}	{ b, (}	{ ε, + }	{ + }	{ ε, + }	{ b, (}
4	{ b, (}	{ b, (}	{ ε, + }	{ + }	{ ε, + }	{ b, (}

Рис. 4.9 Построение множества FIRST для нетерминалов G_3

В общем случае множество FIRST может быть построено для цепочки α, при этом FIRST(α) включает в себя все терминальные символы, с которых могут начинаться цепочки, выводимые из α.

$FIRST(\alpha) = \{ a \in VT \mid \alpha \Rightarrow^* a\alpha_1, \alpha_1 \in (VT \cup VN)^* \}$

Расширим содержание функции isnullable следующим образом:

- isnullable(t) = false for $t \in VT$
- isnullable(ε) = true
- isnullable($b_1 b_2 \ldots b_k$) = nullable(b_1) $\wedge$ nullable(b_2) $\wedge \ldots \wedge$ nullable(b_k);

тогда

- FIRST(Aα) = FIRST(A) если isnullable(A) == false
- FIRST(Aα) = FIRST(A) $\cup$ FIRST (α) если isnullable(A) == true.

Множество FOLLOW(A) для нетерминала A определяется как множество терминальных символов b, которые в сентенциальных

формах для некоторой грамматики могут располагаться непосредственно справа от A; $S \Rightarrow^* \alpha Ab\beta$, см. рисунок 4.8.

$FOLLOW(A) = \{a \in VT \mid S \Rightarrow^* \alpha A\beta, \beta \Rightarrow^* a\gamma, A \in VN, \alpha, \beta, \gamma \in (VT \cup VN)^*\}$

Будем считать, что любая анализируемая цепочка заканчивается символом $, который не является терминальным символом грамматики, и обозначает концевой маркер строки.

Алгоритм 4.13 Построение множества FOLLOW
Вход: G = (VT, VN, P, S)
Выход: множества FOLLOW для нетерминалов грамматики G

```
    foreach A in VN  do FOLLOW(A) = {}
    FOLLOW(S) = {$}
    flag  = 1
    while(flag) {
        flag = 0
        foreach A in VN {
          foreach B → αAβ in P{
            FOLLOW(A) = FOLLOW(A) ∪ FIRST(β);
            if isnullable(β)
                FOLLOW(A) = FOLLOW(A) ∪ FOLLOW(B);
            if FOLLOW(A) изменилось flag = 1
            }
        }
    }
```

Пример 4.14. Пошаговый процесс построения множеств FOLLOW для грамматики G_3 из примера 4.13 показан на рис. 4.10.

	FLLW(S)	FLLW(A)	FLLW(B)	FLLW(T)	FLLW(Y)	FLLW(Z)
0	$	Ø	Ø	Ø	Ø	Ø
1	$	{ $,) }	Ø	Ø	Ø	Ø
2	$	{ $,) }	{ $,) }	Ø	Ø	Ø
3	$	{ $,) }	{ $,) }	{ $, + }	Ø	{ $,) }
4	$	{ $,) }	{ $,) }	{ $, +,) }	Ø	{ $,) }
5	$	{ $,) }	{ $,) }	{ $, +,) }	{ $,) }	{ $,) }
6	$	{ $,) }	{ $,) }	{ $, +,) }	{ $,) }	{ $,) }

Рис. 4.10. Построение множеств FOLLOW для нетерминалов G_3

Метод рекурсивного спуска не применим к грамматикам в следующих случаях:

- если в грамматике существуют альтернативные правые части для некоторого нетерминала $A \to \alpha \mid \beta$ такие, что $FIRST(\alpha) \cap FIRST(\beta) \neq \emptyset$, то метод рекурсивного спуска неприменим для этой грамматики;
- если в грамматике есть правила $A \to \alpha \mid \beta$, такие что $\beta \Rightarrow^* \varepsilon$, и $FIRST(\alpha) \cap FOLLOW(\beta) \neq \emptyset$, то метод рекурсивного спуска неприменим к данной грамматике.

С учетом достаточности становится понятно, что метод рекурсивного спуска применим к весьма узкому подклассу КС-грамматик. Известны более широкие подклассы КС-грамматик, для которых существуют эффективные распознаватели, обладающие тем же свойством, что и распознаватели на основе рекурсивного спуска: входная цепочка считывается один раз слева направо, процесс разбора полностью детерминирован, и время работы такого алгоритма линейно зависит от длины входной цепочки. К таким грамматикам относятся, например, грамматики простого предшествования.

4.7 Восходящие распознаватели КС-языков

При разборе снизу-вверх необходимо, просматривая цепочку входных символов слева-направо (*сдвиг*), найти такую подцепочку, которая соответствует правой части одного из правил грамматики, и заменить эту подцепочку на символ из левой части правила (*свертка*). Этот процесс продолжается до тех пор, пока не удастся свернуть входную цепочку к начальному символу грамматики, что означает успех, либо на очередном шаге невозможно осуществить свертку, т. е. исходная цепочка не является предложением языка, порождаемого грамматикой.

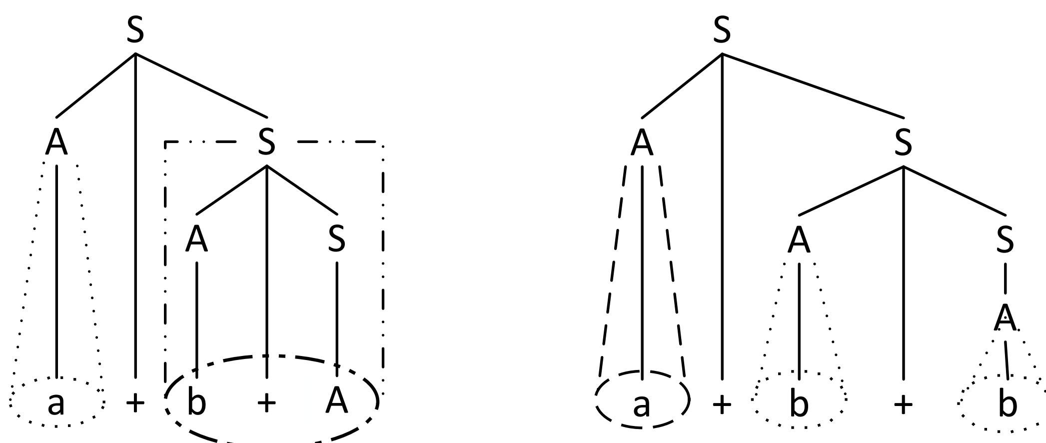
Если в цепочке одновременно присутствует несколько подцепочек, соответствующих правым частям правил, то возникает вопрос: относительно которой из подцепочек необходимо совершить свертку.

Определение. Пусть α – сентенциальная форма грамматики, причем $\alpha = \gamma_0\beta\gamma_1$ ($S \Rightarrow^* \alpha$). Тогда β является *фразой* сентенциальной формы α для нетерминала B, если $S \Rightarrow^* \gamma_0 B \gamma_1$ и $B \Rightarrow^+ \beta$. В том случае, когда $B \Rightarrow \beta$, говорят, что β – *простая фраза*.

Самая *левая простая фраза* называется *основой* сентенциальной формы.

Рисунок 4.11 демонстрирует фразу, простые фразы и основу для сентенциальных форм из примера 4.1.

В ходе распознавания без возвратов методы, основанные на разборе снизу-вверх, выполняют поиск *основы* сентенциальной формы, и если основу удалось найти, то осуществляется её свертка. Этот процесс повторяется до тех пор, пока не удастся свернуть анализируемую цепочку к начальному символу грамматики, что соответствует успеху. Если на очередном шаге не удастся найти основу, или после последней свертки единственный нетерминальный символ не является начальным символом грамматики, то в этом случае можно утверждать, что анализируемая цепочка не принадлежит языку, порождаемому грамматикой.



a) простая фраза и фраза для сентенциальной формы a+b+A

b) основа и простые фразы для сентенциальной формы a+b+b

Рис. 4.11 Фразы сентенциальной формы.

4.7.1 Грамматики простого предшествования

Одним их наглядных примеров работы восходящих распознавателей является разбор сентенциальных форм для грамматик простого предшествования. Идея применения грамматик предшествования состоит в том, что для двух любых символов L и R ($L, R \in V$), стоящих рядом в сентенциальной форме, причем L стоит слева от R, можно определить три отношения относительно основы:

- $L \doteq R$: оба символа принадлежат основе сентенциальной формы;
- $L \lessdot R$: R является самым левым символом основы (голова основы), а L не принадлежит основе и расположен непосредственно перед основой в сентенциальной форме;
- $L \gtrdot R$: L является самым правым символом основы (хвост основы), а R не принадлежит основе и расположен сразу за ней в сентенциальной форме.

Грамматикой простого предшествования называют такую приведенную без циклических ($A \to A$, $A \in VN$) и ε-правил КС-грамматику G(VN, VT, P, S), где $V = VT \cup VN$, в которой для каждой упорядоченной пары терминальных и нетерминальных символов:

1. Для каждой упорядоченной пары терминальных и нетерминальных символов выполняется не более чем одно из трех отношений предшествования:
   - $L \doteq R$ ($\forall$ L, R $\in$ V), тогда и только тогда, когда существует правило $A \to \alpha LR\beta \in P$, где $A \in VN$, $\alpha, \beta \in V^*$;
   - $L \lessdot R$ ($\forall$ L, R $\in$ V), тогда и только тогда, когда существует правило $A \to \alpha LD\beta \in P$ и вывод $D \Rightarrow^* R\gamma$, где $A, D \in VN$, $\alpha, \beta, \gamma \in V^*$;
   - $L \gtrdot R$ ($\forall$ L, R $\in$ V), тогда и только тогда, когда существуют правило $A \to \alpha CR\beta \in P$ и вывод $C \Rightarrow^* \gamma L$ или правило $A \to \alpha CD\beta \in P$ и выводы $C \Rightarrow^* \gamma L$ и $D \Rightarrow^* R\delta$, где $A, C, D \in VN$, $\alpha, \beta, \gamma, \delta \in V^*$.

2. Двум любым нетерминалам в левой части правила не могут соответствовать одинаковые правые части (то есть в грамматике не должно быть двух различных правил с одной и той же правой частью).

Отношения $\doteq$, $\lessdot$ и $\gtrdot$ называют отношениями предшествования для символов грамматики. Отношение предшествования единственно для каждой упорядоченной пары символов. Между какими-либо двумя символами может и не существовать отношения предшествования, это значит, что они не могут находиться рядом ни в одной сентенциальной форме. Отношения предшествования зависят от порядка, в котором стоят символы в сентенциальной форме, и в этом смысле их нельзя путать со знаками математических операций, они не обладают свойствами коммутативности и ассоциативности. Из того, что существует отношение $L \lessdot R$, не следует, что существует отношение $L \gtrdot R$,

отношения предшествования не являются симметричными. Из существования отношения L ≐ R не следует существования отношения R ≐ L.

4.7.2 Построение отношений простого предшествования

Отношения предшествования между символами грамматики удобно строить, введя два дополнительных отношения FIRST и LAST для символов грамматики.

Пусть дана грамматика G (VN, VT, P, S), где $V = VT \cup VN$, а L и R – два символа из словаря грамматики V. Тогда L FIRST R тогда и только тогда, когда в множестве P существует правило вида $L \rightarrow R\alpha$, где $\alpha \in V^*$. Отношение L LAST R существует тогда и только тогда, когда в множестве P существует правило вида $L \rightarrow \alpha R$, где $\alpha \in V^*$.

Рассмотрим грамматику G_1 = ({S, B, G}, {a, b, (,)}, S, P), где множество

P: S → bBb
 B → (G | a
 G → Ba)

Для этой грамматики отношение FIRST = {(S, b), (B, a), (B, ( ), (G, B)}, а отношение LAST = {(S, b), (B, a), (B, G), (G,))}. $FIRST^+$ и $LAST^+$ – есть транзитивные замыкания отношений FIRST и LAST, соответственно (о построение транзитивного замыкания отношений см. п. 1.2).

Отношения FIRST и LAST можно задать в виде бинарных матриц следующим образом:

FIRST

	S	B	G	b	(	a	)
S				1			
B					1	1	
G					1		
b							
(							
a							
)							

LAST

	S	B	G	b	(	a	)
S				1			
B			1			1	
G							1
b							
(							
a							
)							

Рис 4.12. Отношения FIRST и LAST для символов G_1

Отношение $\doteq$ строится тривиально: для любого правила грамматики $A \to \alpha LR\beta \in P$ следует, что $L \doteq R$.

В [2] доказано, что отношения $\lessdot$ и $\gtrdot$ можно получить следующим образом:

- $\lessdot = (\doteq)(FIRST^{+})$; в виде бинарной матрицы как произведение бинарных матриц отношений $\doteq$ и $FIRST^{+}$;
- $\gtrdot = ((LAST^{+})^{T})(\doteq)(FIRST^{*})$, где $(LAST^{+})^{T}$ – транспонированная матрица транзитивного замыкания отношения LAST, а $FIRST^{*}$ – транзитивно-рефлексивное замыкание отношения FIRST.

Результирующая матрица для грамматики G_1 показана на рис. 4.13. Матрица отношений простого предшествования называется бесконфликтной, если между любыми двумя символами грамматики из словаря V существует не более одного отношения простого предшествования.

	S	B	G	b	(	a	)
S							
B				$\doteq$		$\doteq$	
G				$\gtrdot$		$\gtrdot$	
b		$\doteq$			$\lessdot$	$\lessdot$	
(		$\lessdot$	$\doteq$		$\lessdot$	$\lessdot$	
a				$\gtrdot$		$\gtrdot$	$\doteq$
)				$\gtrdot$		$\gtrdot$	

Рис. 4.13 Матрица отношений предшествования для G_1

4.7.3 Алгоритм сдвиг-свертка для грамматик простого предшествования

Отношения простого предшествования служат для того, чтобы определить в процессе выполнения алгоритма разбора, какое действие: сдвиг или свертка, должно выполняться на каждом шаге алгоритма, и однозначно выбрать цепочку для свертки.

Обычно вводят начальный и концевой маркер для анализируемой цепочки символов, например #, тогда для любого $x \in V$ выполняются следующие отношения: $(\# \lessdot x)$ и $(x \gtrdot \#)$. В словарь

грамматики этот символ обычно не входит и введен для удобства описания работы МП-автомата.

В начальном состоянии МП-автомата считывающая головка находится над первым символом анализируемой цепочки, в стеке МП-автомата находится символ #, в конец анализируемой цепочки помещается такой же символ.

МП-автомат допускает цепочку, если в результате завершения алгоритма в стеке находятся начальный символ грамматики S и символ #, причем вся анализируемая цепочка обработана, и считывающая головка находится над замыкающим символом #.

Алгоритм 4.14 Разбор методом сдвиг-свертка на базе матрицы предшествования символов КС-грамматики.

Шаг 1. Поместить на вершину стека символ #, считывающую головку в начало входной цепочки символов.

Шаг 2. Определить с помощью матрицы предшествования отношение между символом, находящийся на вершине стека (левый символ отношения), и текущим символом входной цепочки, находящимся под считывающей головкой (правый символ отношения).

Шаг 3. Если это отношение $\lessdot$ или $\doteq$, то произвести сдвиг (перенести текущий символ из входной цепочки на вершину стека; переместить считывающую головки на один шаг вправо) и вернуться к шагу 2. Иначе перейти к шагу 4.

Шаг 4. Если это отношение $\gtrdot$, то произвести свертку. Для этого с вершины стека снимаются символы до тех пор, пока между последним снятым символом и символом на вершине стека не будет отношения $\lessdot$. Основа найдена, и она в соответствии с определением грамматики простого предшествования является правой частью для единственного нетерминала A → “основа”. Если такого правила в грамматике нет, то выполнение алгоритма прерывается, с сообщением о непринадлежности анализируемой цепочки языку, порождаемому грамматикой. Если правило существует, то нетерминал A помещается на вершину стека.

Шаг 5. Если считывающая головка находится над символом #, то перейти к шагу 6, иначе к шагу 2.

Шаг 6. Если содержимое стека #S – разбор закончен успехом (цепочка допущена), если нет, то анализируемая цепочка отвергнута.

Если на любом шаге работы алгоритма, в котором проверяется наличие отношений предшествования между соседними символами анализируемой цепочки, отношение предшествования не будет найдено, то это говорит об ошибке, и процесс разбора должен быть остановлен с отрицательным результатом (цепочка отвергнута).

Читателю предлагается самостоятельно проверить, что цепочка b((aa)a)b допускается алгоритмом 4.15, использующим матрицу отношений простого предшествования см. рис 4.13 , а цепочка b(a)b – не допускается.

Метод разбора для грамматик простого предшествования один из самых простых, он иллюстрирует многие приемы более сложных алгоритмов, реализующих идею восходящего анализа типа «сдвиг-свертка» для грамматик других классов КС-языков. С теоретической точки зрения рассмотренный метод кажется безупречным и эффективным. Однако на практике он не всегда хорош. Более того, почти любая другая техника разбора оказывается лучше метода простого предшествования.

Пример 4.15 Разбор цепочки b(aa)b для грамматики G1

Шаг	Обработанная цепочка	отношение	Необработанная цепочка	Действие	Правило
0	#	⋖	b(aa)b#	сдвиг	
1	#b	⋖	(aa)b#	сдвиг	
2	#b(	⋖	aa)b#	сдвиг	
3	#b(a	⋗	a)b#	свертка	B → a
4	#b(B	≐	a)b#	сдвиг	
5	#b(Ba	≐	)b#	сдвиг	
6	#b(Ba)	⋗	b#	свертка	G → Ba)
7	#b(G	⋗	b#	свертка	B → (G
8	#bB	≐	b#	сдвиг	
9	# bBb	⋗	#	свертка	S → bBb
10	#S		#	допуск	

Рис. 4.14 Допуск цепочки b(aa)b

В процессе конструирования матрицы отношений простого предшествования для некоторой КС-грамматики может быть определено более чем одно отношение. В этом случае говорят о конфликте; и это значит, что данный метод разбора неприемлем для этой грамматики. Тогда требуется изменить грамматику так, чтобы

удалить этот конфликт. В результате такого изменения, или изменений, может существенно измениться множество правил грамматики. Проблема преобразования произвольной КС-грамматики в грамматику, для которой можно применить метод разбора, основанный на основе отношений простого предшествования, алгоритмически неразрешима.

Более того, для того чтобы определить, может ли язык, порождаемый некоторой КС-грамматикой, быть распознан методом простого предшествования, необходимо попытаться построить бесконфликтную матрицу отношений предшествования. Если это удалось, то грамматика является грамматикой простого предшествования, и нет в противном случае.

4.8 Лемма о разрастании КС-языков

Как и для регулярных языков, лемма о разрастании для КС-языков позволяет проверить принадлежности заданного языка классу КС-языков. Доказано, что всякий язык является КС-языком тогда и только тогда, когда для него выполняется лемма о разрастании КС-языков.

Лемма о разрастании (накачке) КС-языков. Пусть L – КС-язык. Тогда существует такая константа n, что если взять достаточно длинную цепочку символов, принадлежащую произвольному КС-языку, причем длина этой цепочки не меньше n, то в ней всегда можно выделить две подцепочки, длина которых в сумме больше нуля, таких, что, повторив их сколь угодно большое число раз, можно получить новую цепочку символов, принадлежащую языку L.

Формально лемму о разрастании для КС-языков можно определить следующим образом: если L – это КС-язык, цепочка $\alpha \in L$, то существует $n \in N$, $n > 0$, такое что если $|\alpha| \geq n$, то $\alpha = \mu\beta\gamma\delta\eta$, где $\beta\delta \neq \varepsilon$ (хотя бы одна из цепочек непуста), $|\beta\gamma\delta| \leq n$ и $\mu\beta^i\gamma\delta^i\eta \in L$ для всех $i \geq 0$ (где N – это множество натуральных чисел).

Пользуясь леммой, докажем, что язык $L = \{a^n b^n c^n \mid n > 0\}$ не является КС-языком.

Допустим, что этот язык является КС-языком. Тогда для него должна выполняться лемма о разрастании, и существует константа n, заданная в этой лемме. Возьмем цепочку $\alpha = a^k b^k c^k$, $|\alpha| \geq n$, принадлежащую этому языку. Если ее записать в виде $\alpha = \mu\beta\gamma\delta\eta$, то по условиям леммы $|\beta\gamma\delta| \leq n$, следовательно, цепочка $\beta\gamma\delta$ не может

содержать вхождений всех трех символов a, b и c – каких-то символов в ней нет. Рассмотрим цепочку $\mu\beta^0\gamma\delta^0\eta = \mu\gamma\eta$. По условиям леммы она должна принадлежать языку, но в то же время она содержит либо n символов a, либо n символов c и при этом не может содержать n вхождений каждого из символов a, b и c, так как $| \mu\gamma\eta | < 3n$. Значит, какой-то символ в ней встречается меньше, чем другие — такая цепочка не может принадлежать языку L. Следовательно, язык L не удовлетворяет требованиям леммы о разрастании КС-языков и поэтому не является КС-языком.

Читателю предлагается самостоятельно доказать, что язык $L_6 = \{ \alpha\alpha^R \mid \alpha = T^+\}$ (п. 2.2, стр. 17) является КС-языком.

4.9 Свойства КС-языков

Класс КС-языков замкнут относительно операции подстановки. Это означает, что если в каждую цепочку символов КС-языка вместо некоторого символа подставить цепочку символов из другого КС-языка, то получившаяся новая цепочка также будет принадлежать КС-языку.

Это основное свойство КС-языков. Формально оно может быть записано так.

Если $L, L_{a1}, L_{a2}, \ldots, L_{an}$ — это произвольные КС-языки и $\{a_1, a_2, \ldots, a_n\}$ — алфавит языка L, n > 0, то язык $L' = \{x_1x_2\ldots x_k \mid a_{j1}a_{j2}\ldots a_{jk} \in L, x_1 \in L_{aj1}, x_2 \in L_{aj2}, \ldots, x_k \in L_{ajk}, k>0; \forall k \geq i > 0: Si > 0{:}n \geq j > 0\}$ также является КС-языком.

Например:

$L = \{0^n1^n \mid n > 0\}$, $L_0 = \{a\}$, $L_1 = \{b^mc^m \mid m > 0\}$ - это исходные КС-языки, тогда после подстановки получаем новый КС-язык: $L' = \{a^nb^{m1}c^{m1}b^{m2}c^{m2}\ldots b^{mn}c^{mn} \mid n > 0, \forall i: m_i>0\}$.

На основе замкнутости относительно операции подстановки можно доказать другие свойства КС-языков. В частности, класс КС-языков замкнут относительно следующих четырех операций:

- объединения;
- конкатенации;
- итерации;
- гомоморфизма (замена символов сентенциальной формы на цепочки).

Класс КС-языков не замкнут относительно операции пересечения. Как следствие, этот класс языков не замкнут и относительно операции дополнения.

Например:
$L_1 = \{a^n b^n c^i \mid n > 0, i > 0\}$ и $L_2 = \{a^i b^n c^n \mid n > 0, i > 0\}$ - КС-языки, но $L = L_1 \cap L_2 = \{a^n b^n c^n \mid n > 0\}$ не является КС-языком (это можно проверить с помощью леммы о разрастании КС-языков см. п. 4.8).

Для КС-языков разрешимы проблема пустоты языка и проблема принадлежности заданной цепочки языку — для их решения достаточно построить МП-автомат, распознающий данный язык. Но для КС-языков является неразрешимой проблема эквивалентности двух произвольных КС-грамматик и, как следствие, также и проблема однозначности заданной КС-грамматики. Не разрешима даже более узкая проблема — проблема эквивалентности заданной произвольной КС-грамматики и произвольной регулярной грамматики.

Хотя в общем случае проблема однозначности для КС-языков не разрешима, для некоторых КС-грамматик можно построить эквивалентную им однозначную грамматику.

Детерминированные КС-языки – значительно более узкий класс, чем все КС-языки. Класс детерминированных КС-языков не замкнут относительно операции объединения и операции пересечения. Класс детерминированных КС-языков замкнут относительно операции дополнения, хотя произвольный КС-язык не замкнут относительно этой операции.

Класс детерминированных КС-языков интересен тем, что для него разрешима проблема однозначности. Доказано, что если язык может быть распознан с помощью ДМП-автомата (и потому является детерминированным КС-языком), то он может быть описан на основе однозначной КС-грамматики [16].

Контрольные вопросы

1. Дайте определение синтаксического дерева. Для каких типов грамматик по Хомскому можно построить синтаксическое дерево вывода?
2. Что понимается под неоднозначностью для контекстно-свободных языков?
3. Дайте определение конфигурации МП-автомата. Перечислите типы конфигураций МП-автомата.

4. Дайте определение приведенной грамматики.
5. Что понимается под бесполезными символами грамматики.
6. Какими свойствами обладают правила грамматики в нормальной форме Хомского?
7. Назовите универсальные распознаватели для контекстно-свободных языков.
8. В чем отличия восходящего и нисходящего метода распознавания контекстно-свободных языков?
9. Что понимается под методом рекурсивного спуска?
10. Какими свойствами должна обладать грамматика чтобы язык, порождаемый ею, мог быть разобран методом рекурсивного спуска?
11. Перечислите отношения предшествования для контекстно-свободных грамматик.
12. Что понимается под бесконфликтной матрицей отношений предшествования.
13. Дайте определение основы сентенциальной формы.
14. Что понимается под сверткой в процессе восходящего разбора?
15. Перечислите свойства детерминированных контекстно-свободных языков.

Список литературы

1. Ахо А. В., Лам М. С., Сети Р, Ульман Д. Д. Компиляторы. Принципы, технологии, инструментарий, 2-е изд. – М.: Издательский дом «Вильямс», 2008. – 1184 с.
2. Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции. В 2-х т.: Пер. с англ. – М.: Мир, 1978.
3. Вирт, Н. Построение компиляторов / Никлаус Вирт. – М.: ДМК Пресс, 2013. – 192 с.
4. И. А. Волкова, А. А. Вылиток, Т. В. Руденко Формальные грамматики и языки. Элементы теории трансляции: Учебное пособие. — М.: Издательский отдел факультета ВМиК МГУ им. М.В.Ломоносова, 2009 — 115 с.
5. Грис Д. Конструирование компиляторов для цифровых вычислительных машин. Пер. с англ. – М.: Мир, 1975.
6. Замятин А. П., Шур А. М. Языки, грамматики, распознаватели: Учебное пособие. – Екатеринбург: Изд-во Урал. ун-та , 2007. – 248 с.
7. Карпов Ю. Г., Теория автоматов: Учебник для вузов. - 1-е издание. – СПб: Издат. дом ПИТЕР, 2003 год. – 208с.
8. Карпов Ю. Г., Теория и технология программирования. Основы построения трансляторов. – СПб.; БХИ-Петербург, 2005, – 272 с., ил.
9. Мозговой М.В., Классика программирования: алгоритмы, языки, автоматы, компиляторы. Практический подход / – СПб.: Наука и техника, 2006. – 320 с.
10. Молчанов А. Ю. Системное программное обеспечение: Учебник для вузов – СПб.: Питер, 2006. - 395 с
11. Новиков Ф. А. Дискретная математика: Учебник для вузов. 2-е изд. Стандарт третьего поколения. – СПб.: Питер, 2013. – 432 с: ил.
12. Пентус А. Е., Пентус М. Р. Теория формальных языков: Учебное пособие. – М.: Изд-во ЦПИ при механико-математическом ф-те МГУ, 2004. – 80 с.
13. Пратт Т., Зелковиц. М. Языки программирования: разработка и реализация. 4-е издание. М.: Питер, 2002, – 688 с., ил.
14. Серебряков В. А., Галочкин М. П. Основы конструирования компиляторов. – М.: Эдиториал, 2001. – 224 с.
15. Хопкрофт, Д. Введение в теорию автоматов, языков и вычислений /Д. Хопкрофт, Р.Мотвани, Д.Ульман. –М.: Изд.дом Вильямс, 2008. – 527 с., ил.

16. Dexter Kozen: Automata and Computability, Springer, 1997
17. Handbook of formal languages / G. Rozenberg, A. Salomaa, (eds.). Word, Language, Grammar. Volume 1, 2, 3. Springer-Verlag Berlin Heidelberg 1997.
18. Levelt, W. J. M. An introduction to the theory of formal languages and automata / – John Benjamins B.V. 2008.
19. Linz, Peter. An introduction to formal languages and automata / Peter Linz. – 5th ed. Jones & Bartlett Learning, 2012.
20. Sudkamp, Thomas A. Languages and machines: an introduction to the theory of computer science / Thomas A. Sudkamp. - 3nd ed. Addison Wesley 2005.

Лаздин Артур Вячеславович

Формальные языки, грамматики, автоматы
учебное пособие

В авторской редакции
Редакционно-издательский отдел Университета ИТМО
Зав. РИО Н. Ф. Гусарова
Подписано к печати
Заказ №
Отпечатано на ризографе

Редакционно-издательский отдел
Университета ИТМО
197101, Санкт-Петербург, Кронверский пр., 49