

Федеральное агентство по образованию
Российской Федерации
Омский государственный университет им. Ф.М. Достоевского
Факультет компьютерных наук
Кафедра информационной безопасности

Н.Ф. Богаченко, Р.Т. Файзуллин

АВТОМАТЫ, ГРАММАТИКИ, АЛГОРИТМЫ

Омск 2006

УДК 519.713:519.766:510.5

ББК 22.18я73

Б 733

Богаченко Н.Ф., Файзуллин Р.Т.

Б 733 Автоматы, грамматики, алгоритмы: Учебно-методическое пособие. – Омск: Издательство На-
следие. Диалог-Сибирь, 2006. – 140 с.: ил.

В пособии представлен обзор современных направ-
лений развития теории автоматов, даны элементы тео-
рии формальных языков и грамматик, теории алго-
ритмов, теории сложности. Теоретический материал со-
провождается примерами решения типовых задач с по-
дробными пояснениями.

Для студентов, обучающихся по специальности
075200 – «Компьютерная безопасность» и по специаль-
ности 220100 – «Вычислительные машины, комплексы,
системы и сети».

УДК 519.713:519.766:510.5

ББК 22.18я73

Одобрено учебно-методической комиссией и ученым
советом факультета компьютерных наук ОмГУ.

© Богаченко Н.Ф., Файзуллин Р.Т., 2006

© Омский госуниверситет, 2006

Глава 1

Языки и грамматики

С появлением электронных вычислительных машин возникла потребность в общении с подобными устройствами. В связи с этим появились специальные языки, которые стали называться искусственными в отличие от естественных языков общения людей. Искусственные языки должны быть, с одной стороны, удобными и понятными для человека, а с другой – должны восприниматься устройствами. Совмещение этих требований в одном языке оказалось трудной задачей, поэтому появились средства для преобразования текстов с языка, понятного человеку, на язык устройства. Такие средства называются *трансляторами*.

Транслятор может быть *интерпретатором* или *компилятором*. Интерпретатор последовательно читает предложения входного языка, анализирует их и сразу же выполняет, а компилятор не выполняет предложения языка, а строит программу, которая в дальнейшем может быть запущена для получения результата. На вход компилятора подается текст, написанный на входном языке – языке, понятном человеку, а результатом работы компилятора является текст на языке, понятном устройству. Для построения компилятора необходимо однозначное и точное задание входного и выходного языков. Такое задание предполагает определение правил построения

допустимых конструкций (выражений) языка. Множество таких правил называют **синтаксисом** языка. Кроме того, задание должно включать описание назначения и смысла каждой конструкции языка. Такое описание называют **семантикой** языка. Для построения точных и недвусмысленных описаний применяют метод абстракций, который предполагает выделение наиболее существенных свойств рассматриваемого объекта и опускание свойств, менее значимых для рассматриваемого случая. Например, при построении модели входных языков можно рассматривать входной текст как последовательность символов, построенную по определенным правилам, отвлекаясь от характера начертания символов и их расположения на листе. Математические модели, использующие представление текстов в виде цепочек символов (слов), называют *формальными языками и грамматиками*.

1.1. Основные определения

Буква – это простой неразделимый знак. Множество букв образует **алфавит**.

Пример 1.1. Обычный алфавит, обычные буквы: $A = \{a, b, c, \dots\}$. Бинарный алфавит $B = \{0, 1\}$. Команды языка программирования $C = \{\text{got to}, \text{stop}, \dots\}$, в этом алфавите буквы состоят из нескольких символов. Алфавит $D = \{a, b, c, +, !\}$ содержит 5 букв. Алфавит $E = \{00, 01, 10, 11\}$ содержит 4 буквы, каждая из которых состоит из двух символов.

Если $A \subseteq B$, то A называют **подалфавитом** B или B – **расширением** алфавита A .

Последовательность букв алфавита называется **словом** или **цепочкой** в этом алфавите. Число букв, входящих в слово α , называется его **длиной** и обозначается $|\alpha|$ или $l(\alpha)$.

Пример 1.2. Слово $\alpha = ab++c$ в алфавите D имеет длину $l(\alpha) = 5$, а слово $\beta = 00110010$ в алфавите E имеет длину $l(\beta) = 4$.

Для полноты картины приведем так называемый альтернативный набор терминов:

- 1) «слово» \Leftrightarrow «буква»,
- 2) «словарь» \Leftrightarrow «алфавит»,
- 3) «предложение» («строка») \Leftrightarrow «слово» («цепочка»).

Следует следить за контекстом, чтобы не допустить смешение понятий.

Пусть задан алфавит A . Множество всевозможных слов, которые могут быть построены из букв алфавита A , обозначим A^* . При этом предполагается, что «пустое» слово ε также принадлежит множеству A^* . Возникает вопрос, как образуется множество всех слов? Ответ на него дает следующая схема.

$$A^0 = \varepsilon, \quad (1.1)$$

$$A^1 = A, \quad (1.2)$$

$$A^2 = A \times A, \quad (1.3)$$

...

$$A^* = A^0 \cup A^1 \cup A^2 \cup \dots = \cup A^i, \quad (1.4)$$

$$A^+ = A^* - \varepsilon. \quad (1.5)$$

Одна из основных операций над словами – это бинарная операция **конкатенация**:

$$(\cdot)(\alpha, \beta) \rightarrow \alpha\beta \quad (1.6)$$

Можно расширить определение конкатенации:

$$(\cdot)(\alpha, \beta), (\beta, \gamma) \rightarrow \alpha\beta\gamma \quad (1.7)$$

Операция (\cdot) не коммутативна.

Будем говорить, что β есть **подслово** слова α , если $\alpha = \gamma\beta\delta$, где γ и δ – это также слова из A^* .

Пример 1.3. Пусть $\alpha = abac$, тогда $\varepsilon, a, b, ab, ba, bac, ac, c$ – подслова. Кстати, α – подслово самому себе.

Другая естественная операция – это **замена** одного под- слова другим. Ее можно представить как функцию, вернее как соответствие, так как результат действия зависит от порядка применения. Естественно ввести правило, по которому операция замены проводится *слева направо*.

Пример 1.4. Замена $f(ab) = baa$, то есть подслово ab заменяется на подслово baa .

Язык L – это подмножество всех слов:

$$L \subseteq A^*. \quad (1.8)$$

Для языков справедливы отношения:

$$L^0 = \{\varepsilon\}, \quad (1.9)$$

$$L_i L_j = \{\alpha\beta \mid \alpha \in L_i, \beta \in L_j\}, \quad (1.10)$$

$$L^n = L^{n-1} L, n \in \mathbb{N}. \quad (1.11)$$

Возникают следующие основные задачи:

1. Как породить язык, слова которого будут семантически осмысленны? Для этого вводится понятие *грамматики* или *порождающей системы*.

2. Понятие грамматики может способствовать решению двух проблем:

2.1. Как порождать новые слова, принадлежащие заданному языку L ?

2.2. Как по заданному языку L и слову α выяснить, принадлежит ли это слово данному языку? Последняя задача называется *задачей грамматического разбора* (*проблемой разрешимости или распознавания*), то есть, если быть ближе к практике, это проверка того, что представленное слово построено по правилам языка.

1.2. Грамматика

Множество $\Gamma = \{N, T, P, S\}$ называется **грамматикой**, если его элементы определяются по следующим правилам:

1) N – алфавит **нетерминальных** символов; буквы этого алфавита могут входить в промежуточные слова, но не должны входить в *продукт действия грамматики* или в *результат порождения* (то есть в слова, порождаемые грамматикой);

2) T – алфавит **терминальных** символов, причем $N \cap T = \emptyset$; это в некотором смысле завершающие символы, то есть те, из которых будет состоять *продукт действия грамматики*;

3) P – конечное **множество продукций**:

$$P \subseteq (N \cup T)^+ \times (N \cup T)^*. \quad (1.12)$$

Поясним, что подразумевается под множеством продукций. Это не что иное как множество **правил вывода** или **порождающих правил**, которые регламентируют замену слов, содержащих нетерминальные и терминальные символы, на другие слова. Причем пустое слово нельзя заменять на непустое, а наоборот – можно. Как правило, элемент множества P представляет собой пару слов, разделенных знаком « \rightarrow »: $\alpha \rightarrow \beta$;

4) S – **начальный символ**. Заметим, что начальный символ принадлежит алфавиту нетерминальных символов.

Кроме того, множество $V = N \cup T$ – это **словарь** грамматики Γ .

Пусть $\alpha \rightarrow \beta$ – правило грамматики Γ и $\xi = \gamma_1 \alpha \gamma_2$ – слово, причем $\gamma_1, \gamma_2 \in V^*$. Тогда слово $\psi = \gamma_1 \beta \gamma_2$ может быть получено из слова ξ путем применения правила $\alpha \rightarrow \beta$ (то есть заменой в ξ подслова α на β). В этом случае говорят, что слово ψ **непосредственно выведено** из слова ξ и обозначают $\xi \Rightarrow \psi$ (**непосредственный вывод**).

Если задан набор слов $\Omega = (\omega_0, \omega_1, \dots, \omega_n)$ таких, что существует последовательность непосредственных выводов:

$$\omega_0 \Rightarrow \omega_1, \omega_1 \Rightarrow \omega_2, \dots, \omega_{n-1} \Rightarrow \omega_n, \quad (1.13)$$

то такую последовательность называют **выводом** ω_n из ω_0 в грамматике Γ и обозначают $\omega_0 \Rightarrow^* \omega_n$.

Множество конечных слов терминального алфавита T грамматики Γ , выводимых из начального символа S , называется **языком, порождаемым грамматикой** Γ , и обозначается $L(\Gamma)$:¹

$$L(\Gamma) = \{\omega \in T^* | S \Rightarrow^* \omega\}. \quad (1.14)$$

Исходя из определения, получаем, что среди множества правил грамматики должно присутствовать хотя бы одно правило вида $S \rightarrow \alpha$, где S – начальный символ, $\alpha \in V^* = (N \cup T)^*$ – произвольное слово.

Две грамматики Γ_1 и Γ_2 называются **эквивалентными**, если они порождают один и тот же язык, то есть $L(\Gamma_1) = L(\Gamma_2)$.

Пример 1.5.

1. $N = \{ \text{(предложение)}, \text{(подлежащее)}, \text{(артикль)}, \text{(существительное)}, \text{(сказуемое)}, \text{(дополнение)} \}$.

2. $T = \{ \text{the, dog, bit, me} \}$.

3. $P = \{ \text{(предложение)} \rightarrow \text{(подлежащее)} \text{(сказуемое)} \text{(дополнение)},$

$\text{(подлежащее)} \rightarrow \text{(артикль)} \text{(существительное)},$

$\text{(артикль)} \rightarrow \text{the},$

$\text{(существительное)} \rightarrow \text{dog},$

$\text{(сказуемое)} \rightarrow \text{bit},$

$\text{(дополнение)} \rightarrow \text{me} \}$

4. $S = \text{(предложение)}$.

Определим язык, порождаемый представленной грамматикой $\Gamma = \{N, T, P, S\}$. Для этого построим вывод в этой грамматике:

$(\text{предложение}) \Rightarrow (\text{подлежащее}) (\text{сказуемое}) (\text{дополнение})$
 $\Rightarrow (\text{артикль}) (\text{существительное}) (\text{сказуемое}) (\text{дополнение}) \Rightarrow$
 $\text{the} (\text{существительное}) (\text{сказуемое}) (\text{дополнение}) \Rightarrow \text{the dog}$
 $(\text{сказуемое}) (\text{дополнение}) \Rightarrow \text{the dog bit} (\text{дополнение}) \Rightarrow \text{the}$
 $\text{dog bit me}.$

Таким образом $L(\Gamma) = \{ \text{the dog bit me} \}$.

¹В некоторых случаях для обозначения языка удобно использовать запись $L(S)$, где S – начальный символ грамматики.

Для того, чтобы не путаться в обозначениях и автоматизировать процесс *производства* слов языка, применяется символика Бэкуса-Наура или, иначе, запись ведется в **нормальной форме Бэкуса-Наура** (БНФ):

1. «<» (*мета-открыть*), «>» (*мета-закрыть*) – такими символами выделяются элементы из N , чтобы не спутать их с элементами из T .

2. «::=» (*мета-присвоить*). Мета-присвоить можно интерпретировать как отображение одного слова в другое или, иначе, замену (это альтернатива символа « \rightarrow »).

3. «|» (*мета-или*). Если в P имеются элементы $(\alpha \rightarrow \beta), (\alpha \rightarrow \gamma)$, то этот факт можно записать как $\alpha ::= \beta | \gamma$.

Отметим, что впервые БНФ была успешно применена для определения синтаксиса языка Алгол-60.

Пример 1.6.

$$N = \{ \langle A \rangle, \langle S \rangle \},$$

$$T = \{ a, b, c, d \},$$

$$P = \{ \langle S \rangle \rightarrow a \langle A \rangle d, \\ \langle A \rangle \rightarrow b \langle A \rangle | c \langle A \rangle | c \}.$$

Эта грамматика будет порождать все слова вида: $a\{b, c\}^+d$. Здесь знак плюс означает повтор один или более раз, и на каждой итерации выбирается элемент из фигурных скобок.

Пример 1.7.

$$N = \{ \langle S \rangle \},$$

$$T = \{ (,) \},$$

$$P = \{ \langle S \rangle \rightarrow \langle S \rangle \langle S \rangle | (\langle S \rangle) | () \}.$$

Выведем слово $()(())()$:

$$() (()) () \Leftarrow \langle S \rangle (()) \Leftarrow \langle S \rangle (\langle S \rangle ()) \Leftarrow$$

$$\Leftarrow \langle S \rangle (\langle S \rangle \langle S \rangle) \Leftarrow \langle S \rangle (\langle S \rangle) \Leftarrow$$

$$\Leftarrow \langle S \rangle \langle S \rangle \Leftarrow \langle S \rangle .$$

1.3. Иерархия Хомского

Грамматика $\Gamma = \{N, T, P, S\}$ без всяких ограничений называется **грамматикой Хомского типа 0** или грамматикой **общего вида**. Грамматики типа 0 не имеют никаких ограничений на правила вывода. Любое правило может быть построено с использованием произвольных слов.

Если продукции грамматики имеют вид

$$\alpha \rightarrow \beta, \quad \alpha = \gamma_1 A \gamma_2, \quad \beta = \gamma_1 \delta \gamma_2, \quad (1.15)$$

$$\gamma_1, \gamma_2 \in V^* = (N \cup T)^*, \quad A \in N, \quad \delta \in V^+ = (N \cup T)^+, \quad (1.16)$$

то грамматика Γ называется **контекстно-зависимой** (КЗГ) или **грамматикой Хомского типа 1**. Нетерминальный символ A заменяется на непустое слово δ . Слова γ_1 и γ_2 остаются неизменными в ходе применения правила вывода, поэтому их называют **контекстами** (соответственно, левым и правым). Альтернативное определение КЗГ дает следующая теорема [8].

Теорема 1.1. *Грамматика Γ является контекстно-зависимой тогда и только тогда, когда для любой ее продукции $\alpha \rightarrow \beta$ выполнено условие*

$$1 \leq |\alpha| \leq |\beta|. \quad (1.17)$$

Иными словами, КЗГ не укорачивает слова.

Грамматики типа 1 значительно удобнее на практике, чем грамматики типа 0, поскольку в левой части продукции всегда заменяется один нетерминальный символ, который можно связать с некоторым синтаксическим понятием, в то время как в грамматике типа 0 можно заменять сразу несколько символов, в том числе и терминальных.

Пример 1.8. Грамматика $\Gamma = \{N, T, P, S\}$, в которой $N = \{S, A, B\}$, $T = \{a, b, c, d\}$, $P = \{P_1, P_2, P_3, P_4, P_5, P_6\}$, где

$$P_1 = S \rightarrow aAS,$$

$$P_2 = AS \rightarrow AAS,$$

$$P_3 = AAA \rightarrow ABA,$$

$$P_4 = A \rightarrow b,$$

$$P_5 = bBA \rightarrow bcdA,$$

$$P_6 = bS \rightarrow ba,$$

является контекстно-зависимой, поскольку второе и шестое правила имеют непустой левый контекст, третье и пятое правила содержат оба контекста. Вывод в такой грамматике может иметь вид:

$$S \Rightarrow aAS \Rightarrow aAAS \Rightarrow abAS \Rightarrow abbS \Rightarrow abba.$$

Пример 1.9. Рассмотрим КЗГ: $N = \{S, Y, Z\}$, $T = \{x, y, z\}$, $P = \{P_1, \dots, P_7\}$, где

$$P_1 = S \rightarrow xSYZ$$

$$P_2 = S \rightarrow xYZ$$

$$P_3 = xY \rightarrow xy$$

$$P_4 = yY \rightarrow yy$$

$$P_5 = yZ \rightarrow yz$$

$$P_6 = ZY \rightarrow YZ$$

$$P_7 = zZ \rightarrow zz$$

Заметим, что правило P_6 в данной формулировке не укладывается в определение КЗГ, но оно очевидным образом может быть заменено следующей цепочкой правил:

$$ZY \rightarrow ZA \quad ZA \rightarrow YA \quad YA \rightarrow YZ$$

Эта грамматика продуцирует слова вида: $x^n y^n z^n, n > 0$.

Следующим типом в классификации Хомского является *тип 2*, или *контекстно-свободные* грамматики (КСГ). Правила вывода для КСГ имеют вид:

$$A \rightarrow \alpha, \quad (1.18)$$

$$A \in N, \quad \alpha \in V^*. \quad (1.19)$$

Эти правила получаются из правил грамматики типа 1 при условии $\gamma_1 = \gamma_2 = \varepsilon$ и $\delta \in V^*$. Очевидно, что это более сильное ограничение на грамматику, но область приложений такой грамматики уже. То есть, есть языки, представимые КЗГ, но не КСГ.

Заметим, что именно КСГ используются для описания языков программирования.

Пример 1.10. $S \rightarrow aSb, S \rightarrow \varepsilon$.

Эта грамматика продуцирует выражения вида $a^n b^n$, $n > 0$.

Пример 1.11. $S \rightarrow aSa, S \rightarrow bSb, S \rightarrow aa, S \rightarrow bb$.

Эта грамматика порождает язык, каждое слово которого состоит из двух частей: слова $\alpha \in T^+$ и зеркального отображения этого слова α' : $L = \{\alpha\alpha' \mid \alpha \in T^+\}$. С помощью правил этой грамматики может быть построено следующее слово:

$$S \Rightarrow aSa \Rightarrow abSba \Rightarrow abaSaba \Rightarrow ababbaba.$$

Автоматной грамматикой (грамматикой *типа 3* или *А-грамматикой*) называется контекстно-свободная грамматика с правилами вида:

$$A \rightarrow aB \text{ (или } A \rightarrow Ba), \quad A \rightarrow \varepsilon, \quad (1.20)$$

$$A, B \in N, \quad a \in T. \quad (1.21)$$

А-грамматика может иметь только правила вида $A \rightarrow aB$ – *правосторонние правила*, либо только правила вида $A \rightarrow Ba$ – *левосторонние правила*.

Пример 1.12. Правосторонняя A-грамматика:

$$S \rightarrow aS, \quad S \rightarrow aA, \quad A \rightarrow bA, \quad A \rightarrow bZ, \quad Z \rightarrow \varepsilon$$

и левосторонняя A-грамматика:

$$S \rightarrow Ab, \quad A \rightarrow Ab, \quad A \rightarrow Za, \quad Z \rightarrow Za, \quad Z \rightarrow \varepsilon.$$

Эти грамматики являются эквивалентными и порождают язык $L = \{\underbrace{a \dots a}_n \underbrace{b \dots b}_m \mid n, m > 0\}$.

Множества языков различных типов (порожденных грамматики различных типов) соединены отношением включения:

$$L_{\text{типа 3}} \subset L_{\text{типа 2}} \subset L_{\text{типа 1}} \subset L_{\text{типа 0}}. \quad (1.22)$$

Можно показать, что данные включения являются строгими. Отметим, что наибольшее практическое применение находят грамматики и языки типа 2 и типа 3.

1.4. Построение вывода

1.4.1. Синтаксическое дерево

Каждый язык, задаваемый при помощи грамматики, представляет собой множество слов, построенных по определенным правилам. Напомним, что процесс построения каждого слова называется **выводом**. Чтобы сделать вывод более наглядным, его изображают в виде дерева, которое называется **синтаксическим деревом** или **деревом вывода**.

Учитывая, что вывод любого слова языка, порождаемого заданной грамматикой, должен начинаться с начального символа, *правила построения дерева* можно сформулировать так:

1) начальная вершина или корень дерева обозначается начальным символом грамматики S ; эта вершина образует нулевой ярус дерева;

2) если при выводе слова на очередном шаге используется правило грамматики $A \rightarrow \alpha$ и вершина, отмеченная нетерминальным символом A , расположена на ярусе с номером $(k-1)$, то к построенному дереву нужно добавить столько вершин, сколько содержится символов в слове α , расположить эти вершины на ярусе k , обозначить их символами слова α и соединить эти вершины дугами с вершиной A .

Результатом вывода является множество конечных узлов – *листьев*, которые выписываются при обходе дерева слева – вниз – направо – вверх.

Заметим, что пункт 2 сформулирован для грамматик типа 2 и 3.

Пример 1.13. Рассмотрим грамматику

$$S \rightarrow aSb, \quad S \rightarrow ab,$$

которая порождает язык $L = \{aa...abb...b\}$, где a и b повторяются по n раз, $n = 1, 2, \dots$. Дерево вывода для слова $aaabbb$ представлено на рисунке 1.1

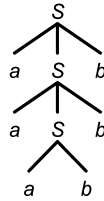


Рис. 1.1. Пример синтаксического дерева (дерева вывода)

1.4.2. Синтаксический разбор

Рассмотрим другой способ задания вывода слова. Для этого пронумеруем все элементы множества продукций или, что то же самое, все правила грамматики.

Последовательность номеров правил грамматики, применение которых позволяет построить вывод заданного слова α

из начального символа грамматики, называется *синтаксическим разбором* α .

Пример 1.14. Возвращаясь к грамматике примера 1.13:

$$(1) S \rightarrow aSb, \quad (2) S \rightarrow ab,$$

получаем, что вывод

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaabbb$$

имеет синтаксический разбор $[1, 1, 2]$.

Заметим, что в ходе построения вывода могут появляться подслова, содержащие несколько нетерминальных символов. В этом случае, чтобы продолжить вывод, можно выбрать для замены любой из них. Таким образом, при выводе слова одни и те же правила могут быть использованы в разном порядке.

Пример 1.15. Пусть $N = \{S, T, M\}$, $T = \{i, +, *, (,)\}$, P состоит из правил:

$$(1) S \rightarrow S + T, \quad (2) S \rightarrow T, \quad (3) T \rightarrow T * M,$$

$$(4) T \rightarrow M, \quad (5) M \rightarrow (S), \quad (6) M \rightarrow i.$$

Один из вариантов вывода слова $i + i * i$ и соответствующий синтаксический разбор имеют следующий вид:

$$S \Rightarrow S + T \Rightarrow T + T \Rightarrow M + T \Rightarrow i + T \Rightarrow i + T * M \Rightarrow$$

$$\Rightarrow i + M * M \Rightarrow i + i * M \Rightarrow i + i * i;$$

$$[1, 2, 4, 6, 3, 4, 6, 6].$$

Очевидно, что существуют и другие выводы, а значит и синтаксические разборы этого слова.

Упражнение 1.1. Выведите слово $i + i$ в этой же грамматике десятью различными способами.

1.4.3. Характеристики вывода

Вывод называется *левым* или *левосторонним*, если при каждом применении правила вывода заменяется самый левый нетерминальный символ. Вывод называется *правым* или *правосторонним*, если при его построении всегда заменяется самый правый нетерминальный символ промежуточного слова.

Заметим, что левосторонние и правосторонние правила вывода уже упоминались при определении автоматной грамматики.

Приведенный выше пример 1.15 вывода слова $i + i * i$ является левосторонним выводом.

Пусть в грамматике Γ существует несколько выводов (а значит и несколько синтаксических разборов) некоторого слова α . В зависимости от α и Γ имеет место два случая:

- различным синтаксическим разборам слова α соответствует одно и то же синтаксическое дерево;
- синтаксические деревья также различны.

Упражнение 1.2. Покажите, что различным вариантам вывода слова $i + i * i$ соответствует одно синтаксическое дерево. Проверьте, что аналогичная ситуация имеет место и для слова $i + i$.

Пример 1.16. В грамматике

$$S \rightarrow AB, \quad A \rightarrow a, \quad A \rightarrow ac, \quad B \rightarrow b, \quad B \rightarrow cb$$

слово acb можно вывести двумя способами:

$$S \Rightarrow AB \Rightarrow Ab \Rightarrow acb$$

или

$$S \Rightarrow AB \Rightarrow Acb \Rightarrow acb.$$

Упражнение 1.3. Проверьте, что синтаксические деревья, соответствующие этим выводам, также различны.

Слово α языка $L(\Gamma)$ называется **неоднозначным**, если для его вывода существует более одного синтаксического дерева. В этом случае грамматика Γ также является **неоднозначной**.

Неоднозначность характерна для КСГ и может иметь место не только в искусственных языках. Примерами неоднозначности в естественных языках являются предложения, допускающие неоднозначную семантику. Во фразе «Пальто испачкало окно» не ясно, что является подлежащим, а что дополнением. Другим примером служит английская фраза: «They are flying planes», которая может быть понята двояко: «Они пилотируют самолет» («are flying» – сказуемое) или «Это летящие самолеты» («flying planes» – дополнение).

1.5. Построение КС- и А-грамматик

Решаемую в этом параграфе задачу можно поставить так: задан вид слов языка, требуется описать язык с помощью *порождающей* грамматики. Эта задача относится к классу эвристических задач, не имеющих общих алгоритмов решения.

Более детально, задача построения грамматик состоит в том, что для заданного в виде описания на естественном языке множества конечных слов, требуется построить грамматику, порождающую это множество слов. Терминальный словарь грамматики включает все символы, используемые для построения слов, входящих в заданное множество. Результатом решения задачи должны явиться нетерминальный словарь и правила грамматики. Построение этих объектов представляется весьма сложным, поскольку оно должно выполняться неформально и требует мысленного охвата всевозможных вариантов построения слов заданного множества. Построение усложняется еще и тем, что оно, как и всякая другая задача синтеза, имеет много решений.

1.5.1. Структура слов и правила грамматики

Чтобы показать, каким образом структура слов отображается в правила грамматики, рассмотрим следующие примеры:

1. Слову, состоящему из заданных символов abc , соответствует правило $S \rightarrow abc$.
2. Слову, начинающемуся с заданного символа a , соответствует правило $S \rightarrow aA$.
3. Слову, заканчивающемуся заданным символом a , соответствует правило $S \rightarrow Aa$.
4. Слову, начинающемуся и заканчивающемуся заданными символами a, b , соответствует правило $S \rightarrow aAb$.
5. Слову, содержащему в середине символ a , соответствует правило $S \rightarrow AaB$.
6. Слову заданной длины $l = 2$ соответствуют правила $S \rightarrow aA$ и $A \rightarrow a$.
7. Слову, состоящему из повторяющихся символов a (структура таких слов характеризуется фразой «один или много»), соответствуют правила, содержащие **рекурсию**:
 $A \rightarrow aA|a$ (правосторонняя рекурсия) или
 $A \rightarrow Aa|a$ (левосторонняя рекурсия) или
 $A \rightarrow AA|a$ (двусторонняя рекурсия).
8. Слову, состоящему из чередующихся символов a и b , соответствуют правила $A \rightarrow aB|a$ и $B \rightarrow bA|b$.

1.5.2. Описание списков

Рассмотрим построение грамматик для последовательностей символов и последовательностей символов с разделителями. Такие последовательности часто называют **списками**.

1. Обозначим элемент последовательности символом a . Простейшая последовательность может состоять из одного элемента a . Все другие последовательности могут быть получены путем приписывания к уже построенной последовательности еще одного элемента. Если обозначить оставшуюся часть последовательности нетерминальным символом R , а всю

последовательность символом L , то получим правила грамматики в виде:

$$L \rightarrow aR, \quad R \rightarrow aR|\varepsilon.$$

Заметим, что этот же список порождают правила $L \rightarrow a|aL$.

2. В предыдущей задаче предполагалось, что список L должен содержать хотя бы один элемент. Если же допустить, что множество слов, порожденных правилами грамматики, может включать пустой символ, то к построенным правилам нужно добавить еще одно правило $L \rightarrow \varepsilon$.

3. Рассмотрим построение списка, между элементами которого должны стоять разделители. Выберем в качестве разделителя звездочку. Простейший список, как и в предыдущем случае, состоит из одного элемента, а построение списка из нескольких элементов может быть выполнено приписыванием к уже построенному списку разделителя с элементом списка. Правила, соответствующие этому построению, имеют вид:

$$L \rightarrow aR, \quad R \rightarrow *aR|\varepsilon.$$

Стоит заметить, что в случае списка с разделителем, обойтись одним нетерминалом нельзя.

4. Если список с разделителями может быть пустым, то приведенный выше набор правил нужно дополнить еще одним правилом с пустой правой частью: $L \rightarrow \varepsilon$.

В общем случае, если описано множество слов, представляющих собой некоторый язык, и требуется построить грамматику, порождающую это множество слов, то следует поступать так:

1. Выписать несколько примеров из заданного множества слов.
2. Проанализировать структуру слов, выделяя начало, конец, повторяющиеся символы или группы символов.
3. Ввести обозначения для сложных структур, состоящих из групп символов. Такие обозначения являются нетерминальными символами искомой грамматики.

4. Построить правила для каждой из выделенных структур, используя для задания повторяющихся структур рекурсивные правила.
5. Объединить все правила.
6. Проверить с помощью выводов возможность получения слов с разной структурой.

Пример 1.17. Применение приведенных рекомендаций рассмотрим на следующем примере. Требуется построить грамматику для языка L , терминальный словарь которого $T = \{*, +\}$, а слова, образующие язык, имеют следующую структуру:

а) каждое слово начинается звездочкой и заканчивается двумя звездочками;

б) между началом и концом слова могут быть либо непустая последовательность плюсов, либо несколько таких последовательностей, разделенных символами $*$.

1. Вначале построим несколько слов заданного языка, которые могут быть представлены в следующем виде:

* + + + **,
 * + * + * + **,
 * + + * + + + * + + + + **,
 * + + + * + * + + * + + + + + **.

2. Рассматривая приведенные слова, можно выделить следующие их структурные компоненты:

- начало слова (символ $*$),
- конец слова (символы $**$),
- непустая группа плюсов,
- последовательность групп плюсов, разделенных звездочками.

3. Обозначим группу плюсов символом A , а последовательность групп плюсов символом C .

4. Выделенные структуры можно рассматривать как списки. Так группа плюсов представляет собой список без разделителей. Правила грамматики, задающей такой список, имеют вид: $A \rightarrow +B$, $B \rightarrow +B|\varepsilon$. Последовательность групп плюсов, разделенных звездочкой, представляет собой список с разде-

лителем, элементом такого списка является группа плюсов A , а разделителем – звездочка. Правила грамматики, задающей такой список, можно записать так: $C \rightarrow AE$, $E \rightarrow *AE|\varepsilon$. Учитывая, что каждое слово языка должно иметь начало и конец, и, выбирая в качестве начального символа грамматики символ S , получаем правило, определяющее общий вид слова: $S \rightarrow *C**$.

5. Объединяя построенные правила, окончательно получаем схему искомой грамматики:

$$S \rightarrow *C**, \quad C \rightarrow AE, \quad E \rightarrow *AE|\varepsilon,$$

$$A \rightarrow +B, \quad B \rightarrow +B|\varepsilon.$$

6. С помощью правил построенной грамматики может быть получено, например, следующее слово: $S \Rightarrow *C** \Rightarrow *AE** \Rightarrow *A*AE** \Rightarrow *A*A*AE \Rightarrow *A*A*A** \Rightarrow *A*A*+B** \Rightarrow *A*A*+** \Rightarrow *A*+B*+** \Rightarrow *A*+*+** \Rightarrow *+B*+*+** \Rightarrow *+*+*+**$.

1.5.3. Основные конструкции языков программирования

Одной из основных областей применения грамматик является описание языков программирования. Учитывая широкое использование подобных описаний в литературе и их важное значение для создания компиляторов, рассмотрим построение грамматик для основных конструкций языков программирования. Чтобы сократить размеры грамматик, на рассматриваемые конструкции накладываются некоторые, иногда существенные, ограничения.

Грамматики, описывающие целые числа без знака и идентификаторы. Целые числа представляют собой последовательность цифр, поэтому их можно рассматривать как списки, элементами которых являются цифры. Используя в качестве аналога грамматику, задающую список без разде-

лителей, получаем схему грамматики для *целых чисел* в виде:

$$N \rightarrow DR, \quad R \rightarrow DR|\varepsilon, \quad D \rightarrow 0|1|\dots|9.$$

Структуру *идентификатора* можно представить в виде двух компонент: начала и основной части. Началом может быть любая из букв, а основная часть представляет собой список без разделителей, элементами которого могут быть либо буквы, либо цифры. Используя выделенные компоненты, получаем схему грамматики вида:

$$\begin{aligned} I &\rightarrow CR_1, & R_1 &\rightarrow CR_1|DR_1|\varepsilon, \\ D &\rightarrow 0|1|\dots|9, & C &\rightarrow a|d|c|\dots|z. \end{aligned}$$

Если наложить ограничения на длину идентификатора, например, допустить использование идентификаторов, состоящих только из трех символов, то схема грамматики получается проще (отсутствует рекурсия):

$$\begin{aligned} I_1 &\rightarrow CA_1, & A_1 &\rightarrow CA_2|DA_2, \\ A_2 &\rightarrow C|D, & D &\rightarrow 0|1|\dots|9, & C &\rightarrow a|d|c|\dots|z. \end{aligned}$$

Граматики для арифметических выражений.

Условимся рассматривать арифметические выражения, использующие *только знаки сложения и умножения*. Вначале построим грамматику для *выражений без скобок*. Такие выражения представляют собой слова, которые можно рассматривать как списки с разделителями, где роль разделителей выполняют знаки операций. В соответствии с этой аналогией получаем схему грамматики, в которой символ I обозначает идентификатор, определенный ранее:

$$H \rightarrow IR_2, \quad R_2 \rightarrow WIR_2|\varepsilon, \quad W \rightarrow +|*.$$

Чтобы построить грамматику, допускающую использование в арифметических выражениях *скобок без вложенности*, представим структуру таких выражений в виде списка с разделителями, элементами которого являются выражения без скобок

или выражения без скобок, заключенные в скобки. Разделителями этого списка являются знаки операций. Такой структуре соответствует следующая схема грамматики:

$$G \rightarrow HR_3|(H)R_3, \quad R_3 \rightarrow WG|\varepsilon.$$

Для построения грамматики арифметических выражений, допускающих применение *вложенных скобок*, следует предусмотреть возможность использования в качестве элемента списка не только выражения без скобок, но и выражения, в котором могут быть использованы скобки. Схема грамматики, учитывающая эту возможность, может быть записана в виде:

$$E \rightarrow HR_4|(E)R_4, \quad R_4 \rightarrow WE|\varepsilon.$$

Грамматика для описаний. Пусть требуется построить грамматику для описания целых и вещественных переменных. Описание переменных определенного типа должно начинаться указателем типа "real" или "int". В полном тексте описания переменных определенного типа могут повторяться. Например, полное описание может включать три разных описания переменных целого типа. Полное описание должно заканчиваться точкой. В качестве разделителя описаний переменных разных типов примем точку с запятой, а в качестве разделителя переменных одного типа – запятую. Структуру полного описания можно представить в виде двух вложенных списков с разделителями. Внутренний список, рассматриваемый как элемент внешнего списка, представляет собой описание переменных одного типа. Он имеет заголовок в виде указателя типа (с пробелом), за которым следует последовательность идентификаторов, разделенных запятыми. Внешний список использует в качестве разделителя точку с запятой. Схема грамматики рассматриваемого вида может быть записана так:

$$Z \rightarrow Z_1R_5; \quad R_5 \rightarrow; Z_1R_5|\varepsilon$$

$$Z_1 \rightarrow real\ L|int\ L \quad L \rightarrow IR_6 \quad R_6 \rightarrow, IR_6|\varepsilon$$

Грамматика, задающая последовательность операторов присваивания. Допустим, что в правой части оператора присваивания могут быть использованы только выражения без скобок. В качестве разделителя операторов в последовательности примем точку с запятой. Учитывая, что последовательность операторов соответствует списку с разделителем в виде точки с запятой, и, используя определенные ранее конструкции идентификатора и выражения без скобок, получаем искомую схему грамматики в виде:

$$U \rightarrow U_1 R_7, \quad R_7 \rightarrow ; U_1 R_7 | \varepsilon,$$

$$U_1 \rightarrow I := H.$$

Грамматики, описывающие условные операторы и операторы цикла. Допустим, что рассматриваются *условные операторы*, аналогичные используемым в языке Паскаль, с разделителями "if", "then", "else". В качестве условия в таких операторах разрешается использовать отношения, состоящие из двух идентификаторов, соединенных знаками «=» и «≤». Структура такого оператора определяется последовательностями фиксированной длины, для описания которых можно воспользоваться простым перечислением компонент. Одна последовательность определяет полный и сокращенный условные операторы, а другая – конструкцию «отношение». Схема грамматики, задающая эти последовательности, может быть изображена так:

$$V \rightarrow \text{if } V_2 \text{ then } U_1 V_1, \quad V_1 \rightarrow \text{else } U_1 | \varepsilon,$$

$$V_2 \rightarrow IV_3, \quad V_3 \rightarrow \leq I | I.$$

В этой грамматике U_1 определяется схемой предыдущей грамматики.

Рассмотрим описание *операторов цикла*, подобных используемым в языке Паскаль, с разделителями "while", "do", "repeat", "until". Каждый оператор может быть описан в виде

простой последовательности ограниченной длины, в которой используются две последние грамматики:

$$W \rightarrow \textit{while } V_2 \textit{ do } U_1$$

и

$$W_1 \rightarrow \textit{repeat } U_1 \textit{ until } V_2.$$

1.5.4. Различия между КС- и А-грамматиками

В основе создания правил грамматики лежит способ выделения структуры заданного множества слов.

Контекстно-свободную грамматику строить проще, общий подход ее построения состоит в пошаговой детализации. Разбиваем слово на компоненты и вводим нетерминалы, отражающие их суть. Каждую компоненту, полученную на предыдущем шаге, разбиваем на подкомпоненты и так до тех пор, пока не дойдем до терминалов.

Пример 1.18. Рассмотрим КС-грамматику *пассажирского поезда*:

$\langle \text{поезд} \rangle \rightarrow \langle \text{локомотив} \rangle \langle \text{вагоны} \rangle$
 $\langle \text{локомотив} \rangle \rightarrow \text{паровоз} \mid \text{тепловоз} \mid \text{электровоз}$
 $\langle \text{вагоны} \rangle \rightarrow \langle \text{вагон} \rangle \mid \langle \text{вагон} \rangle \langle \text{вагоны} \rangle$
 $\langle \text{вагон} \rangle \rightarrow \text{купейный} \mid \text{плацкартный} \mid \text{общий} \mid \text{СВ} \mid \text{ресторан}$

Здесь терминалами мы считаем объекты типа «электровоз», «ресторан» и т. п., хотя могли спуститься и до стоп-крана и колесных пар.

Пример 1.19. Построим *КС-грамматику действительного числа*, то есть грамматику слов от полного представления числа, например $-123.567E-21$, до вырожденных: 0. или .1 (в этом числе обязательна точка и хотя бы одна цифра в целой или дробной части).

Терминалами здесь являются знаки «+» и «-», цифры от «0» до «9», десятичная точка «.» и символ «E». То есть

$T = \{+, -, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ., E\}$. В слове <число> можно выделить следующие основные компоненты: <знак числа>, <целая часть>, «.», <дробная часть>, «E», <знак порядка> и <порядок>.

Отдельные компоненты или их группы могут отсутствовать. При таком разложении «.» и «E» – терминалы, а все остальное – нетерминалы, требующие дальнейшей декомпозиции. Так <целая часть> – это одна цифра или несколько цифр, а <знак числа> – «+» или «-». Нетрудно видеть, что <знак числа> и <знак порядка> ничем не отличаются друг от друга и для них можно ограничиться одним понятием – <знак>. С точки зрения синтаксиса нет разницы и между <целой частью>, <дробной частью> и <порядком>, и их вполне можно определить одно через другое.

При построении грамматики используем БНФ и квадратные скобки для слов, которые необязательны. Заметим, что в этом случае правила вида $A \rightarrow a|\varepsilon$ можно описать как $A \rightarrow [a]$.

<число> \rightarrow [<<знак>]<целая часть>.
 [<дробная часть>][E[<знак>]<порядок>]|
 [<знак>]<целая часть>].
 <дробная часть>[E[<знак>]<порядок>]
 <целая часть> \rightarrow <цифра>[<целая часть>]
 <дробная часть> \rightarrow <целая часть>
 <порядок> \rightarrow <целая часть>
 <цифра> \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
 <знак> \rightarrow + | -

Техника построения А-грамматик определяется видом их правил, где в правой части первым должен стоять терминал². Для формирования таких правил просматриваются исходные слова, выписываются терминалы, которые можно встретить в качестве первого символа, и вводятся нетерминалы, описывающие остаток слова. Созданные нетерминалы детализируются

²Имеется ввиду правосторонняя А-грамматика. Для левосторонней – дальнейшие рассуждения аналогичны, но вместо остатка слова рассматривается его начальная часть.

точно также как и исходный, то есть просматривается остаток слова и т. д.

Пример. 1.20. Построим *автоматную грамматику действительного числа*. <Число> может начинаться с «+» или «-» и тогда оставшуюся часть числа резонно назвать <числом без знака>. <Число> также может начаться любой цифрой от 0 до 9 и остаток слова будет тем же <числом без знака>. Наконец число может начинаться десятичной точкой «.», и остаток такого слова уже иной. Если <число без знака> может в качестве продолжения содержать цифры и должно завершиться той же точкой, то после точки идет фрагмент слова, который точки уже не содержит. Есть смысл назвать этот фрагмент <дробной частью и порядком>. Рассуждая аналогичным образом мы получим А-грамматику, где использованы следующие сокращения для имен нетерминалов: <чис> – <число>, <чбз> – <число без знака>, <дчп> – <дробная часть и порядок>, <пор> – <порядок>, <пбз> – <порядок без знака>:

$$\begin{aligned} \langle \text{чис} \rangle &\rightarrow +\langle \text{чбз} \rangle \mid -\langle \text{чбз} \rangle \mid 0\langle \text{чбз} \rangle \mid 1\langle \text{чбз} \rangle \mid \dots \mid 9\langle \text{чбз} \rangle \\ &\mid \langle \text{дчп} \rangle \\ \langle \text{чбз} \rangle &\rightarrow 0\langle \text{чбз} \rangle \mid 1\langle \text{чбз} \rangle \mid \dots \mid 9\langle \text{чбз} \rangle \mid \langle \text{дчп} \rangle \mid \cdot \\ \langle \text{дчп} \rangle &\rightarrow 0\langle \text{дчп} \rangle \mid 1\langle \text{дчп} \rangle \mid \dots \mid 9\langle \text{дчп} \rangle \mid 0 \mid 1 \mid \dots \mid 9 \\ &\mid E\langle \text{пор} \rangle \\ \langle \text{пор} \rangle &\rightarrow +\langle \text{пбз} \rangle \mid -\langle \text{пбз} \rangle \mid 0\langle \text{пбз} \rangle \mid 1\langle \text{пбз} \rangle \mid \dots \mid 9\langle \text{пбз} \rangle \\ &\mid 0 \mid 1 \mid \dots \mid 9 \\ \langle \text{пбз} \rangle &\rightarrow 0\langle \text{пбз} \rangle \mid 1\langle \text{пбз} \rangle \mid \dots \mid 9\langle \text{пбз} \rangle \mid 0 \mid 1 \mid \dots \mid 9 \end{aligned}$$

Данная грамматика допускает порождение совсем уж вырожденных слов, вида «-.» или «.E1», т. е. слов без целой и дробной частей. На данном этапе проигнорируем этот факт, чтобы не усложнять грамматику.

Заметим, что построенная грамматика включает правила в форме $A \rightarrow b$ ($\langle \text{дчп} \rangle \rightarrow 0$, $\langle \text{пбз} \rangle \rightarrow 0$ и др.). Введем дополнительный нетерминал F и пополним грамматику правилом $F \rightarrow \varepsilon$, а правила вида $A \rightarrow b$ заменим на $A \rightarrow bF$. После такого преобразования все требования автоматной грамматики будут выполнены.

В дальнейшем будем допускать наличие в А-грамматике правил вида $A \rightarrow b$.

1.6. Представление А-грамматики в виде графа состояний

Наглядным способом представления А-грамматики является *граф состояний* (или *граф переходов*). Вершины этого графа соответствуют нетерминалам. Из вершины A в вершину B проводится дуга, отмеченная терминалом a , если в грамматике существует правило $A \rightarrow aB$.

Добавим еще одну дополнительную *заключительную вершину* F и проведем дуги из A в F , отмеченные символом b , для каждого правила вида $A \rightarrow b$. Каждое такое правило заменим на $A \rightarrow bF$. Тем самым получим *модифицированную* А-грамматiku. Заметим, что каждому выводу в А-грамматике будет соответствовать путь по графу состояний из начальной вершины S в вершину F . Очевидно, что заключительных вершин может быть несколько.

Пример 1.21. Граф А-грамматики действительного числа (см. пример 1.20) представлен на рисунке 1.2.

А-грамматика называется *недетерминированной*, если в ней найдутся нетерминал A и терминал a , для которых существует несколько нетерминалов B таких, что выполняются правила $A \rightarrow aB$. Это нежелательно с точки зрения построения программ синтаксического анализа (вследствие неоднозначности).

Пример 1.22. А-грамматика действительного числа (см. пример 1.21) недетерминированная, так как содержит, в частности, правила:

$$\langle \text{чбз} \rangle \rightarrow \langle \text{дчп} \rangle \text{ и } \langle \text{чбз} \rangle \rightarrow \langle F \rangle$$

или

$$\langle \text{дчп} \rangle \rightarrow 0\langle \text{дчп} \rangle \text{ и } \langle \text{дчп} \rangle \rightarrow 0F.$$

А-грамматика называется *детерминированной*, если

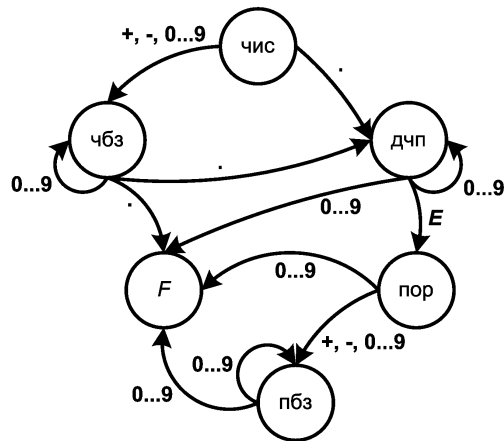


Рис. 1.2. Граф А-грамматики действительного числа

для любого нетерминала A ($A \neq F$) и любого терминала a существует не более одного нетерминала B , для которого выполняется правило $A \rightarrow aB$.

А-грамматика называется **вполне детерминированной**, если для любого нетерминала A ($A \neq F$) и любого терминала a существует единственный нетерминал B , для которого выполняется правило $A \rightarrow aB$.

Для практических приложений важным является следующий факт.

Утверждение 1.1. Любую недетерминированную А-грамматику возможно свести к детерминированной [18].

Отметим, что на практике иногда отказываются от построения детерминированной А-грамматики, эквивалентной исходной недетерминированной. Вместо этого, чтобы добиться детерминированности, расширяют терминальный алфавит, например, используют специальный символ – ограничитель слова.

Пример 1.23. Если считать, что действительное число из примера 1.21 завершается символом «;», то детерминирован-

ная А-грамматика строится элементарно. Граф состояний для этого случая представлен на рисунке 1.3.

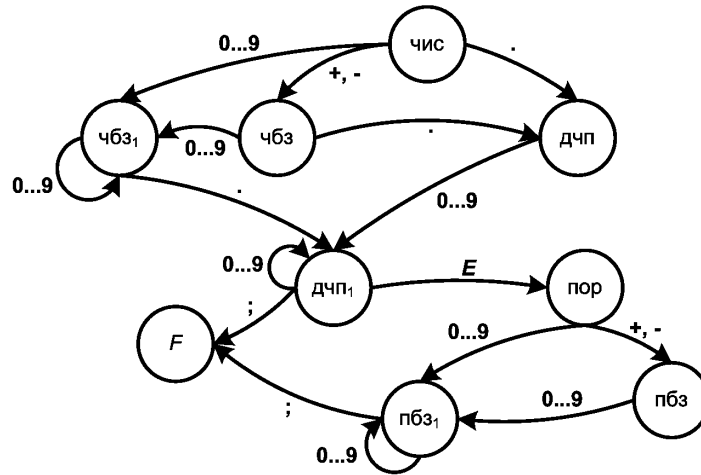


Рис. 1.3. Граф детерминированной А-грамматики действительного числа

Нетерминалы $\langle \text{чбз}_1 \rangle$, $\langle \text{дчп}_1 \rangle$, $\langle \text{пбз}_1 \rangle$ добавлены для того, чтобы исключить возможность формирования числа без целой и дробной частей и числа без порядка после символа «E».

Вполне детерминированная форма должна включать, кроме того, «ошибочный» нетерминал $\langle \text{ош} \rangle$ и правила вида $\langle \text{чбз} \rangle \rightarrow +\langle \text{ош} \rangle$ или $\langle \text{дчп} \rangle \rightarrow \cdot \langle \text{ош} \rangle$ и т. п.

В общем случае для нетерминалов A ($A \neq F$) и терминалов a таких, что в модифицированной детерминированной А-грамматике нет правил $A \rightarrow aB$, необходимо для формирования вполне детерминированной формы добавить правила $A \rightarrow a \langle \text{Error} \rangle$, где $\langle \text{Error} \rangle$ – «ошибочный» нетерминал.

1.7. Преобразование КС-грамматик

Из четырех типов грамматик контекстно-свободные грамматики являются наиболее важными с точки зрения приложений к языкам программирования и компиляции. С помощью КС-грамматики можно определить большую часть структуры языка программирования. При построении грамматик, задающих конструкции языков программирования, часто приходится прибегать к их преобразованию, чтобы порождаемый язык приобрел нужную структуру.

Определение непроеизводящих символов.

Нетерминальный символ X называется **непроеизводящим**, если из него не может быть выведено конечное терминальное слово. Рассматривая правила грамматики, можно сделать вывод, что *если все символы правой части являются производящими, то производящим является и символ, стоящий в левой части*. Последнее утверждение позволяет написать процедуру обнаружения непроеизводящих символов в следующем виде:

1. Составить список нетерминалов, для которых найдется хотя бы одно правило, правая часть которого не содержит нетерминалов.
2. Если найдено такое правило, что все нетерминалы, стоящие в его правой части, уже занесены в список, то добавить в список нетерминал, стоящий в его левой части.
3. Если на шаге 2 список больше не пополняется, то получен список всех производящих нетерминалов грамматики, а все нетерминалы, не попавшие в него, являются непроеизводящими.

Пример 1.24. Анализируя в соответствии с приведенными правилами следующую грамматику:

$$I \rightarrow aIa|bAd|c, \quad A \rightarrow cBd|aAd, \quad B \rightarrow dAf,$$

находим, что здесь непроеизводящими являются символы A и B . После удаления правил, содержащих непроеизводящие символы, получаем: $I \rightarrow aIa|c$.

Определение недостижимых символов.

Символ $X \in T \cup N$ называется **недостижимым** в КС-грамматике, если X не появляется ни в одном выводимом слове. Рассматривая правила грамматики, можно заметить, что *если нетерминал в левой части правила является достижимым, то и все символы правой части являются достижимыми*. Это свойство правил является основой процедуры выявления недостижимых символов, которую можно записать так:

1. Образовать одноэлементный список, состоящий из начального символа.
2. Если найдено правило, левая часть которого уже имеется в списке, то включить в список все символы, содержащиеся в его правой части.
3. Если на шаге 2 новые символы в список больше не добавляются, то получен список всех достижимых символов. Символы, не попавшие в список, являются недостижимыми.

Пример 1.25. Применяя приведенные правила к следующей грамматике:

$$I \rightarrow aIb|c, \quad A \rightarrow bI|a,$$

находим, что A является недостижимым символом.

Определение бесполезных символов.

Символ $X \in T \cup N$ называется **бесполезным** в КС-грамматике, если он является недостижимым или непроизводящим. Исключить бесполезные символы из грамматики можно удаляя вначале правила, содержащие непроизводящие символы, а затем – правила с недостижимыми символами.

Пример 1.26. В качестве иллюстрации применения приведенных правил выполним преобразование следующей грамматики:

$$\begin{aligned} I &\rightarrow ac|bA, & A &\rightarrow cBC, \\ B &\rightarrow aIA, & C &\rightarrow bc|d. \end{aligned}$$

Вначале находим, что A и B являются непроизводящими символами и, исключая правила с непроизводящими символами,

получаем:

$$I \rightarrow ac, \quad C \rightarrow bc|d.$$

В полученной схеме символ C является недостижимым символом. Исключая правила, содержащие этот символ, получаем:

$$I \rightarrow ac.$$

КС-грамматика называется **приведенной**, если она не содержит бесполезных символов.

Исключение леворекурсивных правил.

Напомним, что правило вида $A \rightarrow \alpha A$, где $A \in N$, $\alpha \in (T \cup N)^*$, называется **праворекурсивным**, а правило вида $A \rightarrow A\alpha$ – **леворекурсивным**.

Утверждение 1.2. Для каждой КС-грамматики Γ , содержащей леворекурсивные правила, можно построить эквивалентную грамматику Γ' , не содержащую леворекурсивных правил.

Доказательство. Способ построения эквивалентной грамматики заключается в следующем. Допустим, что исходная грамматика Γ содержит правила:

$$A \rightarrow A\alpha_1 | \dots | A\alpha_m | \beta_1 | \dots | \beta_n, \quad (1.23)$$

где ни одно слово β_i не начинается с A , и $\alpha_j, \beta_i \in (T \cup N)^*$. Введем новый нетерминал A' и преобразуем правила так:

$$A \rightarrow \beta_1 | \dots | \beta_n | \beta_1 A' | \dots | \beta_n A' \quad (1.24)$$

$$A' \rightarrow \alpha_1 | \dots | \alpha_m | \alpha_1 A' | \dots | \alpha_m A'. \quad (1.25)$$

Заменяя все правила с левой рекурсией в Γ описанным способом, получим грамматику Γ' , причем $L(\Gamma) = L(\Gamma')$, поскольку каждое слово, выведенное в грамматике Γ , может быть построено в грамматике Γ' . Для доказательства этого факта рассмотрим построение выводов в Γ и Γ' . В грамматике Γ вывод слова имеет вид:

$$A \Rightarrow A\alpha_1 \Rightarrow A\alpha_1\alpha_1 \Rightarrow A\alpha_1\alpha_1\alpha_1 \Rightarrow \beta_1\alpha_1\alpha_1\alpha_1,$$

в грамматике Γ' это же слово выводится так:

$$A \Rightarrow \beta_1 A' \Rightarrow \beta_1 \alpha_1 A' \Rightarrow \beta_1 \alpha_1 \alpha_1 A' \Rightarrow \beta_1 \alpha_1 \alpha_1 \alpha_1.$$

Заметим, что для более строгого доказательства факта равенства языков $L(\Gamma)$ и $L(\Gamma')$ следует применить индукцию.

Пример 1.27. Требуется преобразовать грамматику

$$E \rightarrow E + T|T, \quad T \rightarrow T * F|F, \quad F \rightarrow (E)|a.$$

Следуя описанному способу, правила для E преобразуем в правила

$$E \rightarrow T|TE', \quad E' \rightarrow +T| + TE',$$

а правила для T преобразуем в правила

$$T \rightarrow F|FT', \quad T' \rightarrow *F| * FT'.$$

В результате получаем грамматику, не содержащую леворекурсивных правил.

Исключение цепных правил.

Правило грамматики вида $A \rightarrow B$, где $A, B \in N$, называется **цепным**.

Утверждение 1.2. Для КС-грамматики, содержащей цепные правила, можно построить эквивалентную ей грамматику, не содержащую цепных правил [18].

Преобразование неукорачивающих грамматик.

Правило грамматики вида $A \rightarrow \varepsilon$ называется **аннулирующим** правилом.

Грамматика называется **неукорачивающей** или грамматикой без аннулирующих правил, если

1) либо схема грамматики не содержит аннулирующих правил,

2) либо схема грамматики содержит только одно правило вида $S \rightarrow \varepsilon$, где S – начальный символ грамматики, и этот символ не встречается в правых частях остальных правил грамматики.

Утверждение 1.3. *Для каждой КС-грамматики, содержащей аннулирующие правила, можно построить эквивалентную ей неукорачивающую грамматику [18].*

1.8. Разрешимость

Еще раз подчеркнем, что наиболее удобными с теоретической и практической точек зрения являются А-грамматики и А-языки. Но их класс слишком узок. Даже язык арифметических выражений не является А-языком. Тем не менее, теория автоматных языков используется при построении трансляторов. Класс языков типа 0, напротив, очень широк, но *неразрешим* в общем случае.

Язык называется **разрешимым**, если для него существует алгоритм, определяющий за конечное число шагов принадлежность или непринадлежность слова языку (конечный алгоритм **грамматического разбора**). Сама задача определения принадлежности слова языку называется **задачей распознавания** (или **задачей разрешимости**).

Языки типа 1, 2 или 3 будем называть **контекстными**. Для них справедливо следующее утверждение.

Теорема 1.2. *Любой контекстный язык разрешим.* (Доказательство этого факта см., например, в [18].)

Задачу распознавания позволяют решать *распознаватели*, о которых мы поговорим в следующей главе.

Глава 2

Распознаватели

2.1. Автоматы как распознаватели

Кроме грамматик существует еще один распространенный метод, обеспечивающий задание языка конечными средствами. Он заключается в использовании *распознавателей (автоматов)*. Распознаватель можно интерпретировать как схематизированный алгоритм, определяющий некоторое множество.

Распознаватель состоит из четырех частей (см. рис. 2.1):

- 1) входной ленты,
- 2) устройства чтения /записи (или указателя),
- 3) управляющего устройства с конечной памятью и
- 4) внешней (вспомогательной, рабочей) памяти.

Входная лента – это линейная последовательность клеток (ячеек). В каждой ячейке может содержаться только один входной символ из некоторого конечного алфавита T . Самую левую и самую правую ячейки могут занимать особые символы – *концевые маркеры*; маркер может стоять только на правом конце ленты или его может не быть совсем.

Устройство чтения (или *указатель*) в каждый момент времени читает (обозревает) одну входную ячейку. За

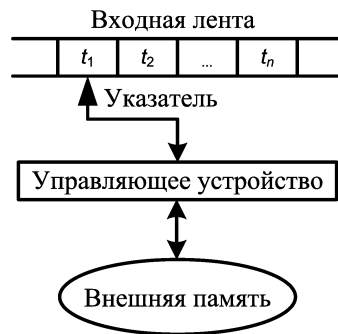


Рис. 2.1. Общий вид распознавателя (автомата)

один шаг работы распознавателя указатель может сдвинуться на ячейку вправо – R, остаться неподвижным – N или сдвинуться на ячейку влево – L. Распознаватель, который никогда не передвигает указатель влево, называется *односторонним*.

Обычно предполагается, что указатель только читает, то есть в ходе работы распознавателя символы на входной ленте не меняются. Но можно рассматривать и такие распознаватели – *преобразователи*, указатель которых не только читает, но и пишет, например, *машина Тьюринга общего вида*.

Внешняя память распознавателя хранит информацию (или данные) определенного типа. *Алфавит памяти* M конечен и информация в памяти образована только из символов этого алфавита. Предполагается, что в любой момент времени можно конечными средствами описать содержимое и структуру памяти, хотя с течением времени объем памяти может становиться сколь угодно большим.

Поведение внешней памяти для заданного класса распознавателей характеризуется двумя функциями: *функцией доступа к памяти* (ФДП) и *функцией преобразования памяти* (ФПП).

ФДП – это отображение множества *состояний памяти* в конечное множество *информационных символов*, которое, как правило, совпадает с алфавитом памяти (ФДП: $M^* \rightarrow M$).

Пример 2.1. Единственная информация, доступная в каждый момент времени в *стеке* – верхний символ стека. Таким образом, функция доступа к памяти, представляющей собой стек, – это такое отображение f из M^* в M , что

$$f(m_1 m_2 \dots m_k) = m_1, \quad (2.1)$$

где $m_i \in M$ ($i = 1, \dots, k$).

ФПП – это отображение, описывающее изменение памяти. Она отображает состояние памяти и *управляющее слово* в состояние памяти (ФПП: $M^* \times M^* \rightarrow M^*$).

Пример 2.2. Пусть операция над памятью, организованной как стек, заменяет верхний символ стека конечным словом символов памяти. Тогда соответствующую ФПП можно определить как такое отображение $g : M^+ \times M^* \rightarrow M^*$, что

$$g(m_1 m_2 \dots m_k, x_1 x_2 \dots x_s) = x_1 x_2 \dots x_s m_2 \dots m_k, \quad (2.2)$$

где $m_i, x_j \in M$. Если верхний символ стека m_1 заменяется пустым словом, то m_2 становится верхним символом стека.

Замечание 2.1. Именно *тип внешней памяти определяет название распознавателя*. Например, распознаватель, память которого организована как стек или, что тоже самое, магазин, называется распознавателем с магазинной памятью. (Обычно его называют автоматом с магазинной памятью или МП-автоматом.)

Ядром распознавателя является *управляющее устройство* с конечной памятью, под которым можно понимать программу, управляющую поведением распознавателя. Управляющее устройство представляет собой *множество состояний* Q вместе с *отображением* δ , которое описывает как меняется состояние в соответствии с текущим входным сим-

волом (то есть находящимся под указателем) и текущей информацией, извлеченной из внешней памяти. Управляющее устройство определяет также в каком направлении сдвинуть указатель и какую информацию поместить в память.

Обобщим вышесказанное. Распознаватель работает, проделывая некоторую последовательность шагов или тактов. В начале такта читается текущий входной символ и с помощью ФДП исследуется память. Входной символ и информация из памяти вместе с текущим состоянием управляющего устройства определяют каким должен быть следующий такт. При этом на каждом такте:

- 1) указатель сдвигается на одну ячейку вправо, влево или сохраняет исходное положение;
- 2) в память при помощи ФПП помещается некоторая информация;
- 3) изменяется состояние управляющего устройства.

Поведение распознавателя удобно описывать в терминах **конфигурации** – мгновенного «снимка» распознавателя, на котором изображены:

- 1) состояние управляющего устройства;
- 2) содержимое входной ленты вместе с положением указателя;
- 3) содержимое памяти.

Конфигурация называется **начальной**, если управляющее устройство находится в заданном **начальном состоянии** q_0 , указатель расположен над самым левым входным символом на ленте, а память имеет заранее установленное начальное содержимое.

Конфигурация называется **заключительной**, если управляющее устройство находится в одном из состояний заранее выделенного **множества заключительных состояний** $F \subseteq Q$, а указатель расположен над правым концевым маркером. Часто требуют, чтобы память в заключительной конфигурации тоже удовлетворяла некоторым условиям.

Управляющее устройство распознавателя является **неде-**

терминированным, если для каждой конфигурации существует конечное множество всевозможных следующих шагов, любой из которых распознаватель может сделать, исходя из этой конфигурации. Устройство называется **детерминированным**, если для каждой конфигурации существует не более одного следующего шага. Недетерминированный распознаватель – удобная математическая абстракция, но ее трудно моделировать на практике.

Говорят, что распознаватель *допускает (распознает) входное слово* α , если начиная с начальной конфигурации, в которой слово α записано на входной ленте, распознаватель может проделать последовательность шагов, завершающуюся заключительной конфигурацией. Недетерминированный распознаватель может при этом проделать много последовательностей шагов и если хотя бы одна из них заканчивается заключительной конфигурацией, то исходное входное слово будет допущено.

Язык, определяемый распознавателем, – это множество входных слов, которые он допускает (распознает).

Для каждого класса грамматик в иерархии Хомского существует класс распознавателей, определяющий тот же класс языков. Справедливы следующие утверждения [18].

Утверждение 2.1. *Язык L автоматный тогда и только тогда, когда он определяется односторонним детерминированным конечным автоматом.*

Утверждение 2.2. *Язык L контекстно-свободный тогда и только тогда, когда он определяется односторонним недетерминированным автоматом с магазинной памятью.*

Утверждение 2.3. *Язык L контекстно-зависимый тогда и только тогда, когда он определяется двухсторонним недетерминированным линейно-ограниченным автоматом.*

Утверждение 2.4. *Язык L без ограничений (общего вида) тогда и только тогда, когда он определяется машиной Тьюринга общего вида.*

2.2. Конечные автоматы и А-грамматики

Конечный автомат – это частный случай одностороннего распознавателя, в котором указатель движется всегда вправо на каждом такте работы и отсутствует внешняя память. Конечный автомат R можно определить множеством из пяти элементов:

$$R = \{Q, T, \delta, q_0, F\}, \quad (2.3)$$

где Q – конечное **множество состояний** автомата; T – конечный **входной алфавит**; $\delta : Q \times T \rightarrow Q$ – **функция переходов**, отображающая множество состояний и входных символов в множество состояний; $q_0 \in Q$ – **начальное состояние**; $F \subseteq Q$ – **множество заключительных состояний**. Очевидно, что конечный автомат можно задать при помощи таблицы, в которой столбцы соответствуют состояниям, а строки – символам входного алфавита. В клетках таблицы записываются значения функции переходов. Состояния, принадлежащие множеству F , отмечаются символом «1». Остальным состояниям приписывается символ «0».

Заметим, что *автоматную грамматику также можно представить в виде таблицы*, где строки соответствуют терминалам, а столбцы – нетерминалам. На пересечении строки a и столбца A записываются нетерминалы B , которые входят в правила $A \rightarrow aB$. Столбец A отмечается единицей, если в грамматике присутствует правило $A \rightarrow \varepsilon$, и нулем – в противном случае.

Очевидно, что *конечный автомат является эквивалентом автоматной грамматики*. Собственно, термин «автоматная» грамматика и связан с этой эквивалентностью. На практике используются обе модели описания А-языков, так как одни утверждения и теоремы проще подтвердить и доказать, используя конечные автоматы, другие – автоматные грамматики.

Отметим, что таблица А-грамматики или конечного автомата представляет собой ни что иное, как отмеченную таблицу

переходов некоторого абстрактного последовательного автомата, представленного моделью Мура [3].

Пример 2.3. В таблице 2.1 записана А-грамматика действительного числа. Данная грамматика является детерминированной, в отличие от недетерминированной А-грамматики примера 1.20, и не содержит терминальный символ «;», как в примере 1.23.

Таблица 2.1. А-грамматика действительного числа

| | <чис> | <чбз> | <дчп> | <пор> | <пбз> |
|----------|-------|-------|-------|-------|-------|
| + | <чбз> | | | <пбз> | |
| − | <чбз> | | | <пбз> | |
| 0 | <чбз> | <чбз> | <дчп> | <пбз> | <пбз> |
| ... | <чбз> | <чбз> | <дчп> | <пбз> | <пбз> |
| 9 | <чбз> | <чбз> | <дчп> | <пбз> | <пбз> |
| . | <дчп> | <дчп> | | | |
| <i>E</i> | | | <пор> | | |
| | 0 | 0 | 1 | 0 | 1 |

Недетерминированный конечный автомат (НДКА) – это автомат, где значение функции переходов δ – не отдельное состояние, как в **детерминированном** конечном автомате (ДКА), а множество состояний. В общем случае у НДКА может быть также не одно, а множество начальных состояний.

На практике для заданного множества иногда легче найти недетерминированное описание. Но по-прежнему справедливо утверждение о том, что *для каждого НДКА существует эквивалентный ему ДКА* (см. § 1.6).

В заключении параграфа приведем в соответствие данные нами определения. **Автоматная грамматика** $\Gamma = \{N, T, P, S\}$ и **конечный автомат** $R = \{Q, T, \delta, q_0, F\}$ называются **соответствующими**, если

$$N = Q, \quad (2.4)$$

$$S = q_0, \quad (2.5)$$

$$(X \rightarrow aY) \in P \Leftrightarrow \delta(X, a) = Y, \quad (2.6)$$

$$(X \rightarrow \varepsilon) \in P \Leftrightarrow X \in F. \quad (2.7)$$

Продолжая аналогию с рассмотренным в [3] **абстрактным последовательным автоматом** $S = \{A, Z, W, \delta, \lambda, a_1\}$, получаем:

$$N_{\text{грамматика}} = Q_{\text{конечный автомат}} = A_{\text{последовательный автомат}}$$

$$T_{\text{грамматика}} = T_{\text{конечный автомат}} = Z_{\text{последовательный автомат}}$$

$$S_{\text{грамматика}} = q_0_{\text{конечный автомат}} = a_1_{\text{последовательный автомат}}$$

$$P_{\text{грамматика}} \Leftrightarrow \delta_{\text{конечный автомат}} \Leftrightarrow \delta_{\text{последовательный автомат}}$$

Отношение принадлежности состояния (нетерминала) множеству $F_{\text{конечный автомат}}$ можно рассматривать как бинарную функцию $\lambda_{\text{последовательный автомат}}$ для модели Мура, в этом случае $W_{\text{последовательный автомат}} = \{0, 1\}$.

2.3. Магазинные автоматы и КС-грамматики

Автомат с магазинной памятью (МП-автомат) – это односторонний недетерминированный распознаватель, внешняя память которого представляет собой стек или, что тоже самое, **магазин**¹ (см. рис. 2.2).

Будем представлять стек, как и входную ленту, в виде цепочки символов или слова, причем верхним элементом стека будем считать самый левый символ.

Исходя из утверждения 2.2, автомат с магазинной памятью – это тип распознавателя, представляющего собой модель синтаксических анализаторов контекстно-свободных языков.

¹Стек часто называют «магазином» в связи с тем, что патроны в магазине автоматического оружия хранятся и используются по правилам стека, то есть в каждый момент доступен только верхний элемент магазина.

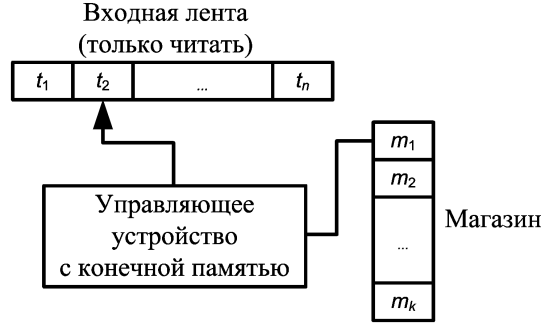


Рис. 2.2. Общий вид автомата с магазинной памятью

Автомат с магазинной памятью можно определить множеством из семи элементов:

$$R = \{Q, T, M, \delta, q_0, m_0, F\}, \quad (2.8)$$

где Q – конечное **множество состояний** управляющего устройства; T – конечный **входной алфавит**; M – конечный **алфавит символов стека**; $\delta : Q \times \{T \cup \{\varepsilon\}\} \times M \rightarrow Q \times M^*$ – **функция переходов**; $q_0 \in Q$ – **начальное состояние** управляющего устройства; $m_0 \in M$ – символ, находящийся в стеке в начальный момент (**начальный символ стека**); $F \subseteq Q$ – **множество заключительных состояний**.

Конфигурацией МП-автомата называется тройка

$$(q, \alpha, \beta) \in Q \times T^* \times M^*, \quad (2.9)$$

где q – текущее состояние управляющего устройства; α – неиспользованная часть входного слова; первый символ слова α находится под указателем; если $\alpha = \varepsilon$, то считается, что входная лента прочитана; β – содержимое стека; левый символ слова β считается верхним символом стека; если $\beta = \varepsilon$, то стек считается пустым.

Такт работы автомата – это **бинарное отношение пере-**

хода (\vdash), определенное на конфигурациях. Будем писать:

$$(q, a\alpha, c\beta) \vdash (q', \alpha, \gamma\beta), \quad (2.10)$$

если множество $\delta(q, a, c)$ содержит (q', γ) , где $q, q' \in Q$, $a \in T \cup \{\varepsilon\}$, $\alpha \in T^*$, $c \in M$ и $\beta, \gamma \in M^*$.

Если $a \neq \varepsilon$, то говорят, что МП-автомат R , находясь в состоянии q и имея a в качестве текущего входного символа, расположенного под указателем, а c в качестве верхнего символа стека, может перейти в новое состояние q' , сдвинуть указатель на ячейку вправо и заменить верхний символ стека словом γ символов стека. Это слово может совпадать с заменяемым символом c . Если $\gamma = \varepsilon$, то верхний символ удаляется из стека, и тем самым список стека сокращается.

Если $a = \varepsilon$, будем называть этот такт ε -тактом. В ε -такте текущий входной символ не принимается во внимание и указатель не сдвигается. Однако состояние управляющего устройства и содержимое стека могут измениться. Заметим, что ε -такт может происходить и тогда, когда все входное слово прочитано.

Важно помнить, что *следующий такт невозможен, если стек пуст*.

Отношение \vdash^i традиционно обозначает выполнение i тактов ($i \geq 0$), \vdash^* – рефлексивно-транзитивное ($i \geq 0$), а \vdash^+ – транзитивное ($i > 0$) замыкание отношения \vdash .

Начальной конфигурацией МП-автомата R называется конфигурация вида (q_0, α, m_0) , где $\alpha \in T^*$, то есть управляющее устройство находится в начальном состоянии q_0 , входная лента содержит слово α , которое нужно распознать, а в стеке есть только начальный символ m_0 .

Заключительная конфигурация – это конфигурация вида (q, ε, β) , где $q \in F$ и $\beta \in M^*$.

Говорят, что слово α **допускается** МП-автоматом, если

$$(q_0, \alpha, m_0) \vdash^* (q, \varepsilon, \beta) \quad (2.11)$$

для некоторых $q \in F$ и $\beta \in M^*$.

Напомним, что *языком, определяемым* (или *допускаемым*) МП-автоматом R (обозначается $L(R)$), называется множество слов, допускаемых автоматом R .

Пример 2.4.² Построим МП-автомат, определяющий язык

$$L = \{0^n 1^n \mid n \geq 0\}.$$

Пусть

$$R = \{\{q_0, q_1, q_2\}, \{0, 1\}, \{z, 0\}, \delta, q_0, z, \{q_0\}\},$$

где

$$\begin{aligned} (1) \delta(q_0, 0, z) &= \{(q_1, 0z)\}, & (2) \delta(q_1, 0, 0) &= \{(q_1, 00)\}, \\ (3) \delta(q_1, 1, 0) &= \{(q_2, \varepsilon)\}, & (4) \delta(q_2, 1, 0) &= \{(q_2, \varepsilon)\}, \\ (5) \delta(q_2, \varepsilon, z) &= \{(q_0, \varepsilon)\}. \end{aligned}$$

Работа МП-автомата R состоит в том, что он копирует в стек начальную часть входного слова, состоящую из нулей, а затем устраняет из стека по одному нулю на каждую единицу, которую он видит на входе. Кроме того, переходы состояний гарантируют, что все нули предшествуют единицам. Например, для входного слова 0011 автомат R проделает следующую последовательность тактов:

$$\begin{aligned} (q_0, 0011, z) &\vdash (q_1, 011, 0z) \vdash (q_1, 11, 00z) \vdash \\ &\vdash (q_2, 1, 0z) \vdash (q_2, \varepsilon, z) \vdash (q_0, \varepsilon, \varepsilon). \end{aligned}$$

Для строгого доказательства необходимо показать, что 1) $L \subseteq L(R)$ и 2) $L \supseteq L(R)$, то есть, что R допускает *только* цепочки вида $0^n 1^n$. Вторая часть доказательства труднее. *Обычно легче доказать, что такие-то цепочки распознаватель допускает, и так же, как для грамматик, гораздо труднее доказать, что он допускает цепочки только определенного вида.* Доказательство этих двух включений оставим в качестве упражнения.

²Пример взят из [18].

Пример 2.5.³ Построим МП-автомат, допускающий язык

$$L = \{\alpha\alpha^R \mid \alpha \in \{a, b\}^+\},$$

где α^R – «зеркальный перевертыш» слова α . Пусть

$$R = \{\{q_0, q_1, q_2\}, \{a, b\}, \{Z, a, b\}, \delta, q_0, Z, \{q_2\}\},$$

где

- (1) $(q_0, a, Z) = \{(q_0, aZ)\},$ (2) $(q_0, b, Z) = \{(q_0, bZ)\},$
- (3) $(q_0, a, a) = \{(q_0, aa), (q_1, \varepsilon)\},$ (4) $(q_0, a, b) = \{(q_0, ab)\},$
- (5) $(q_0, b, a) = \{(q_0, ba)\},$ (6) $(q_0, b, b) = \{(q_0, bb), (q_1, \varepsilon)\},$
- (7) $(q_1, a, a) = \{(q_1, \varepsilon)\},$ (8) $(q_1, b, b) = \{(q_1, \varepsilon)\},$
- (9) $(q_1, \varepsilon, Z) = \{(q_2, \varepsilon)\}.$

МП-автомат R вначале копирует в стек какую-то часть входного слова по правилам (1), (2), (4) и (5) и первым альтернативам правил (3) и (6). Однако R – недетерминированный распознаватель. В любой момент, когда текущий входной символ совпадает с верхним символом стека, он может перейти в состояние q_1 и начать сравнивать слово в стеке с оставшейся частью входного слова. Этот выбор осуществляют вторые альтернативы правил (3) и (6), а по правилам (7) и (8) происходит сравнение. Если R обнаруживает несовпадение очередных символов, то этот экземпляр МП-автомата «умирает», то есть перестает работать. Так как автомат R – недетерминированный, то разные его экземпляры могут дать все возможные для него такты. Если какой-то выбор тактов приводит к тому, что Z снова оказывается верхним (и единственным) символом стека, то по правилу (9) R стирает Z и попадает в заключительное состояние q_2 . Итак, R допускает слово тогда и только тогда, когда все сравнения обнаружили совпадение символов.

Например, для входного слова $abba$ автомат R может среди прочих сделать следующие последовательности тактов:

$$(q_0, abba, Z) \vdash (q_0, bba, aZ) \vdash$$

$$\vdash (q_0, ba, baZ) \vdash (q_0, a, bbaZ) \vdash (q_0, \varepsilon, abbaZ)$$

³Пример взят из [18].

или

$$(q_0, abba, Z) \vdash (q_0, bba, aZ) \vdash (q_0, ba, baZ) \vdash \\ (q_1, a, aZ) \vdash (q_1, \varepsilon, Z) \vdash (q_2, \varepsilon, \varepsilon).$$

Так как вторая последовательность оканчивается заключительным состоянием q_2 , то МП-автомат R допускает входное слово $abba$.

В этом примере ясно видна недетерминированная природа МП-автомата R . Для любой конфигурации вида $(q_0, a\alpha, a\beta)$ автомат R может сделать один из двух тактов: либо поместить в стек еще один символ a , либо устранить из стека верхний символ a .

Можно показать, что языки, определяемые МП-автоматами – это в точности КС-языки [18], об этом, в частности, говорилось в утверждении 2.2. Справедливы и следующие утверждения.

Утверждение 2.5. Пусть Γ – КС-грамматика. По грамматике Γ можно построить такой МП-автомат R , что $L(R) = L(\Gamma)$.

Утверждение 2.6. Пусть R – МП-автомат. Можно построить такую КС-грамматику Γ , что $L(\Gamma) = L(R)$.

Таким образом, для каждой КС-грамматики Γ можно построить МП-автомат, распознающий $L(\Gamma)$. Однако построенные МП-автоматы могут оказаться недетерминированными. В практических приложениях больший интерес представляют детерминированные МП-автоматы, то есть такие, которые в каждой конфигурации могут сделать не более одного очередного такта. К сожалению, детерминированные МП-автоматы не так мощны по своей распознавательной способности, как недетерминированные МП-автоматы. *Существуют КС-языки, которые нельзя определить детерминированными МП-автоматами.*

В заключении параграфа определим линейно-ограниченные автоматы, используемые для построения КЗ-языков. Автомат называется **линейно-ограниченным по памяти**, если для обработки входного слова длины n он

использует не более $p \cdot n$ ячеек памяти. Автомат называется **линейно-ограниченным по времени**, если для обработки входного слова длины n он требует не более $c \cdot n$ шагов по времени. Здесь $p, c \geq 1$ – некоторые константы.

И, наконец, как было отмечено ранее, язык без ограничений определяется машиной Тьюринга, которой мы посвятим следующую главу.

Глава 3

Машина Тьюринга

Прежде всего еще раз напомним, что *машина Тьюринга* не является физическим объектом и принадлежит области абстрактной математики. Это понятие в 1935–1936 годах ввел английский математик и кибернетик Алан Тьюринг.

Как отмечалось в § 2.1, *машина Тьюринга общего вида* – это распознаватель, способный не только читать, но и записывать входные символы. Такой распознаватель называется преобразователем. В этой главе мы пойдем от простого к сложному и построим *машину Тьюринга* как обобщение *концепции Тьюринга*, в основе которой лежит понятие *алгоритма*. Таким образом, к определению машины Тьюринга возможен подход и со стороны теории формальных языков, и со стороны теории алгоритмов¹.

¹Хотелось бы еще раз обратить внимание читателя на подобное «пересечение» различных областей математики. Так, например, конечный автомат можно рассматривать как модель распознавателя, эквивалентного автоматной грамматике (§ 2.2), или как абстрактный последовательный автомат с конечными множествами состояний и сигналов [3].

3.1. Концепция Тьюринга

В основе логических построений данной главы лежит понятие алгоритма. Под **алгоритмом** мы будем понимать точное предписание о выполнении в определенном порядке некоторой системы операций для решения всех задач некоторого данного типа [16].

Теперь сформулируем **проблему алгоритмической разрешимости**, которая была поставлена еще Д. Гильбертом в 1928 г. и над которой работал А. Тьюринг: *существует ли некая универсальная алгоритмическая процедура, позволяющая, в принципе, решить все математические задачи (из некоторого вполне определенного класса) одну за другой* [11]?

Вслед за Тьюрингом, попытаемся построить устройство для выполнения подобной алгоритмической процедуры. Допустим, что это устройство способно находиться в одном из возможных *внутренних состояний*, которые принадлежат *конечному* множеству $\{q_0, q_1, \dots, q_M\}$. Но, несмотря на конечность числа внутренних состояний, наше устройство должно иметь возможность работать с *неограниченным объемом входных данных*. Кроме того, устройство должно иметь доступ к *внешней памяти, также неограниченного объема*, и уметь выдавать *решение любого размера*.

В силу конечности числа внутренних состояний, устройство может обращаться только к тем данным, с которыми оно непосредственно работает в текущий момент времени. Над этими величинами устройство может выполнять некоторые операции. Далее, оно способно записывать результаты во внешнюю память. На этом один шаг работы устройства заканчивается.

Тьюринг предложил изображать входные данные и внешнюю память в виде «ленты» с нанесенными на нее метками. Устройство может «считывать» с ленты информацию, перемещаться по ленте вперед или назад в ходе выполнения операций, а также ставить на ленту новые метки и стирать старые. Таким образом, одну и ту же ленту можно использовать и как источник входных данных, и как внешнюю память. Обосно-

ванием такого «отождествления» понятий является тот факт, что во многих операциях промежуточные результаты вычислений могут играть роль новых исходных данных.

Устройство двигается по ленте до тех пор, пока выполняются вычисления. Когда вычисления закончены, устройство останавливается. При этом результат вычисления должен быть отображен на части ленты, лежащей по одну сторону от устройства. *Для определенности будем считать, что в начале работы все исходные числовые данные и условия задачи расположены на части ленты, находящейся справа от устройства, а ответ (после остановки) – на части ленты, расположенной слева от него.*

Лента представляет собой бесконечную в обоих направлениях последовательность ячеек. Каждая ячейка либо пуста, либо помечена символом из конечного *внешнего алфавита* $\{s_1, \dots, s_F\}$. Несмотря на то, что мы допускаем бесконечность ленты, в каждом конкретном случае входные данные, промежуточные вычисления и окончательный результат должны быть конечными – на бесконечной ленте должно быть конечное число помеченных ячеек. Отсюда следует, что слева и справа от устройства найдутся ячейки, после которых лента будет абсолютно пустой. Отметим, что удобнее пустую ячейку также обозначать некоторым специальным символом s_0 . Устройство считывает по одной ячейке за раз и смещается на одну ячейку влево или вправо.

Итак, поведение нашего устройства полностью определяется его внутренним состоянием и входными данными. При заданном начальном состоянии устройство работает следующим образом:

- переходит в новое состояние или остается в прежнем;
- заменяет считанный символ тем же или другим символом внешнего алфавита (включая «пустой» символ);
- передвигается на одну ячейку вправо или влево;
- решает, продолжить вычисления или остановиться.

Описанное нами устройство, включая бесконечную ленту, называется *машиной Тьюринга* (МТ).

Для задания направления движения устройства вдоль ленты, введем следующие обозначения:

L – движение устройства влево;

R – движение устройства вправо;

STOP – остановка ².

Теперь каждый шаг или такт работы машины Тьюринга можно описать одной из следующих *команд*:

$$q_i s_m \rightarrow q_j s_k L \quad (3.1)$$

$$q_i s_m \rightarrow q_j s_k R \quad (3.2)$$

$$q_i s_m \rightarrow q_j s_k \text{STOP} \quad (3.3)$$

Здесь, слева от стрелки, записываются текущее состояние устройства q_i и символ на ленте s_m , который устройство в данный момент считывает. Оно заменяет этот символ символом s_k , и переходит в состояние q_j , расположенные справа от стрелки. Буквы L или R соответствуют перемещению устройства на одну ячейку влево или вправо. Слово STOP означает смещение устройства на одну ячейку вправо (так как окончательный ответ, включающий в себя результат последнего шага вычислений, должен находиться слева от устройства) и остановку.

Все множество команд, которые описывают действия машины Тьюринга, удобно изображать в виде прямоугольной таблицы, столбцы которой отмечены знаками внутренних состояний, а строки – символами внешнего алфавита. В клетках таблицы записывается соответствующая столбцу q_i и строке s_m выходная тройка знаков, например $q_j s_k L$ (см. табл. 3.1).

²В этой главе мы намеренно сделали некоторые отступления от определений, данных для распознавателей в целом, чтобы показать несколько вариантов задания МТ. В частности, заметим, что вместо остановки (STOP) можно разрешить устройству оставаться на месте (N) не только на последнем такте, как это сделано в § 2.1. Но в этом случае необходимо указать заключительное состояние (или состояния), то есть определить множество F . После того, как устройство переходит в заключительное состояние, МТ останавливается.

Такая таблица называется **функциональной схемой** машины Тьюринга [16]. Отметим, что функциональную схему машины можно задать и простым перечислением всех команд.

Таблица 3.1. Функциональная схема машины Тьюринга

| | q_0 | q_1 | \dots | q_i | \dots | q_M |
|---------|-------|-------|---------|-------------|---------|-------|
| s_0 | | | | | | |
| s_1 | | | | | | |
| \dots | | | | | | |
| s_m | | | | $q_j s_k L$ | | |
| \dots | | | | | | |
| s_F | | | | | | |

Не ограничивая общности дальнейших рассуждений, упростим внешний алфавит до двух символов: «0» и «1». При этом пустым ячейкам будет соответствовать символ «0», а помеченным – символ «1». Внутренним состояниям устройства поставим в соответствие их порядковые номера в двоичной системе счисления: $\{0, 1, 2, 3, 4, 5, \dots\} \rightarrow \{0, 1, 10, 11, 100, 101, \dots\}$.

Пример 3.1. Пусть некоторому вычислительному процессу соответствует следующий участок ленты:

| | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|------------|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 |
| | | | | | | | | | | | | | | | ▲ | | | | | | | | | | | |
| | | | | | | | | | | | | | | | (11010010) | | | | | | | | | | | |

При этом устройство находится в состоянии (11010010). Тогда команда

$$(11010010)1 \rightarrow (11)0 L$$

означает, что символ «1», который в данный момент считывается (на ленте под этим символом расположен указатель) замещается символом «0», внутреннее состояние (11010010) меняется на состояние (11) и устройство перемещается на одну ячейку влево:

$$\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|}
 \hline
 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\
 \hline
 \end{array}$$

▲
(11)

В дальнейшем будем считать, что машина Тьюринга всегда начинает свою работу, находясь во внутреннем состоянии (0).

3.2. Расширенная двоичная форма записи

Теперь обсудим вопрос о том, как представлять в машине Тьюринга числовые данные. Если машина оперирует только натуральными числами, то самый простой способ – унарная система счисления: $\{1, 2, 3, 4, 5, \dots\} \rightarrow \{1, 11, 111, 1111, 11111, \dots\}$. В этом случае символ «0» можно использовать в качестве пробела.

Пример 3.2. Построим функциональную схему машины Тьюринга $\mathbf{UN} + 1$, которая прибавляет единицу к числу в унарном представлении.

Функционирование такой машины описывается следующим набором команд:

$$\begin{array}{ll}
 (0)0 & \rightarrow (0)0 \text{ R} \\
 (0)1 & \rightarrow (1)1 \text{ R} \\
 (1)0 & \rightarrow (0)1 \text{ STOP} \\
 (1)1 & \rightarrow (1)1 \text{ R}
 \end{array}$$

или таблицей 3.2.

Таблица 3.2. Функциональная схема машины Тьюринга $\mathbf{UN} + 1$

| | (0) | (1) |
|---|--------|-----------|
| 0 | (0)0 R | (0)1 STOP |
| 1 | (1)1 R | (1)1 R |

Очевидно, что для записи больших натуральных чисел, унарная система неэффективна. Поэтому, естественно попытаться перейти к двоичной системе. Но сделать это напрямую мы не сможем, так как:

- во-первых, непонятно, когда заканчивается двоичное представление числа и начинается бесконечная последовательность нулей справа, соответствующая пустой ленте;
- во-вторых, невозможно отличить пробелы между числами от нулей, входящих в записи самих чисел;
- в-третьих, помимо чисел нам может понадобиться запись на ленте различных инструкций.

Чтобы преодолеть эти трудности, воспользуемся двумя взаимнообратными процедурами *сокращения* и *расширения* [11].

В процессе считывания любая последовательность из нулей и единиц (с конечным числом единиц) согласно **процедуре сокращения** заменяется последовательностью из нулей, единиц, двоек, троек и т. д. таким образом, чтобы каждое число в получившейся последовательности соответствовало числу единиц между соседними нулями.

Подпоследовательности из нулей и единиц – это по-прежнему числа, записанные в двоичной системе счисления. Числа 2, 3, 4 и т. д. мы можем интерпретировать как метки или некоторые инструкции. Например, пусть 2 – это запятая, указывающая на пробел, 3 – минус, 4 – плюс и т. д.

Пример 3.3. Согласно процедуре сокращения, последовательность на ленте

010001011010101101000111010101111010

при считывании превратится в последовательность

1001211210031141

Далее эта строка (при замене двоек запятыми и переводе чисел в десятичную систему) примет вид:

$9,3,4 + 3 - 1$

В процессе записи на ленту, согласно *процедуре расширения*, в последовательности натуральных чисел, представленных в двоичной системе счисления (и разделенных запятыми), проводятся следующие замены:

$$\begin{array}{lcl} 0 & \Rightarrow & 0 \\ 1 & \Rightarrow & 10 \\ , & \Rightarrow & 110 \end{array}$$

и после этого добавляются бесконечные последовательности нулей с обеих сторон вновь полученной записи.

Отметим, что такая процедура дает возможность отделять последнее число на ленте от бесконечной полосы пустой ленты справа, просто используя «запятую» (110) или какой-либо другой «символ», например (111110), в конце этой записи.

Способ представления чисел при помощи процедуры расширения будем называть *расширенной двоичной формой записи*.

Пример 3.4. Пусть дана последовательность натуральных чисел

$$5, 13, 0, 1, 1, 4$$

В двоичном представлении она принимает вид:

$$101, 1101, 0, 1, 1, 100$$

Согласно процедуре расширения на ленте должна быть записана последовательность:

$$0000\ 10\ 0\ 10\ 110\ 10\ 10\ 0\ 10\ 110\ 0\ 110\ 10\ 110\ 10\ 0\ 0\ 110\ 000$$

При этом, к примеру, расширенная двоичная форма записи числа 13 выглядит как 1010010.

Пример 3.5. Построим функциональную схему машины Тьюринга $\mathbf{XN} \times \mathbf{2}$, которая умножает на два натуральное число, представленное в расширенной двоичной форме.

Очевидно, что для умножения на 2 в двоичной системе счисления достаточно все цифры числа переместить на разряд влево, а в разряд единиц дописать 0.

В процессе чтения числа будем смещать его цифры на ячейку вправо, а перед запятой, которая отделяет число от бесконечной последовательности нулей справа, припишем символ «0»:

| | | |
|-------|---|-----------|
| (0)0 | → | (0)0 R |
| (0)1 | → | (1)0 R |
| (1)0 | → | (0)1 R |
| (1)1 | → | (10)0 R |
| (10)0 | → | (11)1 R |
| (11)0 | → | (0)1 STOP |

Упражнение 3.1. Задайте функциональную схему этой машины таблично. Постройте машину Тьюринга, выполняющую аналогичную процедуру для натуральных чисел, заданных в унарной форме.

3.3. Основная гипотеза теории алгоритмов

По-прежнему рассматривая множество неотрицательных целых чисел, легко убедиться, что все основные математические операции (сложение, умножение, возведение в степень) могут быть выполнены соответствующими машинами Тьюринга. Существуют машины, выполняющие операции, результат которых выражается парой чисел (например, деление с остатком) или даже конечным множеством чисел. Можно сконструировать и такие машины Тьюринга, которые выполняют несколько арифметических операций в зависимости от исходных данных и результатов промежуточных вычислений. В конечном итоге можно вполне обоснованно предположить, что *если некоторая задача четко определена и по природе своей алгоритмична, то найдется машина Тьюринга, способная ее выполнить.*

С другой стороны, может показаться, что принципы построения машин Тьюринга содержат излишние ограничения. Но ни одно из «усовершенствований» машины Тьюринга не

влияет на то, что может быть достигнуто с ее помощью. Хотя некоторая «модернизация» машины и отразилась бы на ее эффективности, но класс решаемых на машине Тьюринга задач остался бы прежним [11].

Вместе с тем, мы не ответили на вопрос о том, действительно ли понятие машины Тьюринга охватывает все математические и логические операции, которые можно назвать *алгоритмическими*.

Независимо от Тьюринга американский логик Алонзо Черч, работая над проблемой алгоритмической разрешимости Гильберта, предложил свою *схему лямбда-исчисления*, оказавшуюся эквивалентной машине Тьюринга.

В итоге повсеместно был принят **тезис Черча-Тьюринга**, который утверждает, что *машина Тьюринга (или ее эквивалент) определяет алгоритмическую процедуру*.

Перефразировав этот тезис следующим образом: *любой алгоритм может быть задан тьюринговой функциональной схемой и реализован в соответствующей машине Тьюринга*, мы получаем **основную гипотезу теории алгоритмов**.

Заметим, что эта гипотеза не может быть строго доказана, подобно математической теореме, так как она представляет собой утверждение об общем понятии алгоритма, которое не является точным математическим понятием. Уверенность в справедливости гипотезы основана главным образом на опытных данных. Значение гипотезы заключается в том, что она уточняет общее и расплывчатое понятие «любого алгоритма» через совершенно точное математическое понятие функциональной схемы машины Тьюринга.

3.4. Универсальная машина Тьюринга

Идея, лежащая в основе *универсальной машины Тьюринга*, состоит в том, чтобы закодировать команды для произвольной машины Тьюринга T в виде последовательности нулей и еди-

ниц, которую затем можно записать на ленте. Эта запись будет использоваться как начальная часть входных данных для некоторой *универсальной машины Тьюринга* U , которая обрабатывает оставшуюся часть ленты точно так, как это сделала бы машина T .

Иными словами, **универсальная машина Тьюринга** – это универсальный имитатор. Начальная часть ленты дает универсальной машине U всю необходимую информацию для точной имитации любой машины T .

Чтобы реализовать универсальную машину, нам потребуется система нумерации машин Тьюринга. Для этого мы должны в соответствии с четкими правилами представить наборы команд каждой машины в виде последовательностей нулей и единиц. Напомним, что общий вид команды машины Тьюринга имеет вид: $q_i s_m \rightarrow q_j s_k R$ (или L, или STOP). Перечислим основные правила такого кодирования.

1. Будем использовать процедуру сокращения или расширенную двоичную форму записи:

$$\begin{array}{llll} 0 & \Rightarrow & 0 & R \Rightarrow 110 \\ 1 & \Rightarrow & 10 & L \Rightarrow 1110 \\ & & & STOP \Rightarrow 11110 \end{array}$$

При этом символов R, L и STOP вполне достаточно для отделения команд друг от друга (они выполняют роль запятой или пробела между командами).

2. Нет необходимости как-то разделять код состояния машины и текущий символ на ленте (ранее состояние заключалось в круглые скобки), так как расположение символа в конце двоичного кода является достаточным отличительным признаком.

Например, фрагмент команды (110)1 будет отображаться на ленте как 1010010.

3. Если предварительно упорядочить команды по возрастанию двоичных чисел, стоящих слева от стрелки (состояние и символ на ленте), то стрелки и числа, непосредственно предшествующие им, можно не кодировать. Но для этого надо убе-

даться в отсутствии «дырок» в получившемся порядке и добавить, где это необходимо, «немые» команды³.

Например, если машина не имеет команды, левая часть которой представляется как $(110)0$, то мы должны добавить в список команд немую команду, например $(110)0 \rightarrow (0)0 R$, которая не вызовет изменений в работе машины.

Пример 3.6. Выпишем в строку «усеченные» команды машины Тьюринга $\mathbf{XN} \times \mathbf{2}$ из примера 3.5 (с дополнительной немой командой $(10)1 \rightarrow (0)0 R$), предварительно упорядочив их указанным способом:

$(0)0R(1)0R(0)1R(10)0R(11)1R(0)0R(0)1STOP$

Эта строка на ленте запишется как последовательность символов:

0011010011001011010001101010101100011001011110

Итак, итогом такого кодирования команд машины Тьюринга является двоичное число, записанное на ленте, которое мы назовем *номером* данной машины Тьюринга. Машину с номером n будем обозначать T_n .

Пусть машина Тьюринга T_n действует на некоторую конечную строку нулей и единиц на ленте, которая подается в устройство справа. Будем интерпретировать эту строку как двоичное представление некоторого числа m . После определенного числа шагов машина T_n останавливается (доходит до команды $STOP$). Последовательность двоичных цифр, которую машина выписала к этому моменту на левой части ленты, и будет результатом. Эту последовательность также можно интерпретировать как некоторое число p . Тогда выражение вида

$$T_n(m) = p \tag{3.4}$$

³На самом деле, есть еще несколько способов «сэкономить» [11], но мы на них останавливаться не будем.

означает, что результатом действия n -й машины Тьюринга T_n на число m является число p .

С другой стороны, это выражение описывает некоторую операцию: для заданных двух чисел n и m мы можем найти число p , если введем m в n -ю машину Тьюринга. Эта операция алгоритмична, следовательно (по основной гипотезе теории алгоритмов) может быть выполнена машиной Тьюринга U : U совершает действия над парой чисел n и m , записанных на ленте, и выдает в результате число p .

Отметим, что при записи на ленту чисел n и m в качестве разделителя можно использовать, например, последовательность 111110.

Список команд машины U должен содержать правила для чтения подходящей команды из «списка», закодированного в числе n , на каждом шаге обработки цифр из числа m . При этом машина U будет совершать значительное количество прыжков влево-вправо по ленте между цифрами, составляющими числа n и m . Такая машина является **универсальной машиной Тьюринга**. Обозначая ее действие на пару чисел (n, m) через $U(n, m)$, получаем:

$$U(n, m) = T_n(m) \quad (3.5)$$

для любых (n, m) таких, что машина T_n корректно определена.

Так как U – также машина Тьюринга, то она сама будет иметь номер: $\exists u$ такое, что $U = T_u$. Вид числа u можно найти в [11].

3.5. Алгоритмически неразрешимые проблемы

Напомним, что цель теории, которую разрабатывал Тьюринг, – решение проблемы алгоритмической разрешимости: существует ли некоторая алгоритмическая процедура для решения всех математических задач, принадлежащих определенному

классу? В силу тезиса Черча-Тьюринга (или основной гипотезы теории алгоритмов) эту проблему можно перефразировать следующим образом: *существует ли машина Тьюринга, решающая все математические задачи из данного класса?* В свою очередь, Тьюринг свел данную проблему к поиску ответа на вопрос: остановится ли в действительности n -я машина Тьюринга, если на ее вход поступит некоторое число m ? Эта задача получила название **проблемы остановки**.

Утверждение 3.1. *Не существует универсального алгоритма для решения вопроса об остановке произвольной машины Тьюринга.*

Доказательство проведем методом от противного – предположим, что указанный алгоритм существует. Тогда существует соответствующая машина Тьюринга S , которая дает ответ на вопрос: остановится ли n -я машина Тьюринга, действуя на число m . Пусть результатом действия машины S будет 0, если n -я машина не остановилась ($T_n(m) = \square$), и 1 – в противном случае ($T_n(m) = p$):

$$S(n; m) = \begin{cases} 0, & \text{если } T_n(m) = \square, \\ 1, & \text{если } T_n(m) \text{ останавливается.} \end{cases} \quad (3.6)$$

Представим теперь бесконечную таблицу, в которую включены окончательные результаты действий всех возможных машин Тьюринга на все возможные входные данные. В этой таблице n -я строка представляет собой результаты вычислений n -й машины Тьюринга при ее работе с $m = 0, 1, 2, 3, 4, \dots$ Как будет следовать из дальнейших рассуждений, на самом деле порядок следования машин в таблице не важен [11]. Поэтому, не ограничивая общности, можно считать, что данная таблица примет вид, представленный таблицей 3.3.

Определим новую процедуру $T_n(m) \circ S(n; m)$, согласно которой машина T_n производит соответствующие действия только если $S(n; m) = 1$. В случае, когда $S(n; m) = 0$, будем записывать в качестве результата также 0. В итоге получаем таблицу 3.4.

Определим **вычислимую последовательность** как бесконечную последовательность, элементы которой могут

Таблица 3.3. Результаты работы машин Тьюринга $T_n(m)$

| $m \rightarrow$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | ... |
|-----------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----|
| n | | | | | | | | | | |
| \downarrow | | | | | | | | | | |
| 0 | \square | \square | \square | \square | \square | \square | \square | \square | \square | ... |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... |
| 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | ... |
| 3 | 0 | 2 | 0 | 2 | 0 | 2 | 0 | 2 | 0 | ... |
| 4 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | ... |
| 5 | 0 | \square | 0 | \square | 0 | \square | 0 | \square | 0 | ... |
| 6 | 0 | \square | 1 | \square | 2 | \square | 3 | \square | 4 | ... |
| 7 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | ... |
| 8 | \square | 1 | \square | \square | 1 | \square | \square | \square | 1 | ... |
| \vdots | \vdots | | | | | | | | \vdots | |
| 197 | 2 | 3 | 5 | 7 | 11 | 13 | 17 | 19 | 23 | ... |
| \vdots | \vdots | | | | | | | | \vdots | |

Таблица 3.4. Результаты работы процедуры $T_n(m) \circ S(n; m)$

| $m \rightarrow$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | ... |
|-----------------|----------|----------|----------|----------|----------|----------|----------|----------|----------|-----|
| n | | | | | | | | | | |
| \downarrow | | | | | | | | | | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... |
| 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | ... |
| 3 | 0 | 2 | 0 | 2 | 0 | 2 | 0 | 2 | 0 | ... |
| 4 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | ... |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... |
| 6 | 0 | 0 | 1 | 0 | 2 | 0 | 3 | 0 | 4 | ... |
| 7 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | ... |
| 8 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | ... |
| \vdots | \vdots | | | | | | | | \vdots | |

быть найдены один за другим при помощи некоторого алгоритма. Это означает, что существует машина Тьюринга, которая производит элементы этой последовательности, будучи поочередно применена к числам $m = 0, 1, 2, 3, 4, 5, \dots$

Исходя из существования машины S , получаем, что все строки таблицы 3.4 представляют собой вычислимые последовательности.

Обратно, любая вычислимая последовательность целых неотрицательных чисел должна появиться среди строк таблицы 3.4 (это свойство выполнялось и для исходной таблицы, содержащей \square).

Считая, что машина Тьюринга S существует, мы получили таблицу вычислительным путем: при помощи процедуры $T_n(m) \circ S(n; m)$. Следовательно, существует некоторая машина Тьюринга Q , применение которой к паре чисел n, m дает значение соответствующей клетки таблицы:

$$Q(n, m) = T_n(m) \circ S(n, m). \quad (3.7)$$

Рассмотрим значения в клетках, расположенных на главной диагонали таблицы⁴ (они выделены жирным шрифтом). Эти элементы образуют некоторую последовательность:

$$0, 0, 1, 2, 1, 0, 3, 7, 1, \dots$$

К каждому элементу последовательности прибавим единицу:

$$1, 1, 2, 3, 2, 1, 4, 8, 2, \dots$$

Очевидно, что процедура получения этих чисел алгоритмична, следовательно, мы получили новую вычислимую последовательность

$$1 + Q(n, n) = 1 + T_n(n) \circ S(n, n), \quad (3.8)$$

с учетом того, что для диагональных элементов $n = m$.

⁴Подобный прием доказательства называется *диагональным процессом* Георга Кантора.

Так как наша таблица содержит в себе *все* вычислимые последовательности, то она должна содержать и новую последовательность. Но это невозможно, так как наша новая последовательность отличается от первой строки первым элементом, от второй строки – вторым, от третьей – третьим и т. д. Получаем противоречие, которое и доказывает утверждение.

Вопрос о том, останавливается ли машина Тьюринга или нет, представляет собой математическую задачу. И наоборот, различные математические задачи могут быть сведены к вопросу об остановке машины Тьюринга⁵. Таким образом, доказав, что не существует алгоритма для решения вопроса об остановке произвольной машины на произвольных входных данных, получаем, что не может быть и общего алгоритма для решения *всех* математических задач. Следовательно, *проблема разрешимости Гильберта не имеет решения*.

Существование алгоритмически неразрешимых классов задач приводит к тому, что стремясь построить требуемый алгоритм, мы вынуждены считаться с тем, что такого алгоритма может и не существовать. Это не означает, что в каждом конкретном случае мы не можем решить определенную математическую задачу или установить остановится ли данная машина Тьюринга либо нет. Но не существует ни одного алгоритма, который позволял бы решить *любую* математическую задачу или давал бы ответ на вопрос об остановке *любой* машины Тьюринга при *любой* входных данных.

В заключении еще раз остановимся на вопросах отображения числовых данных в машине Тьюринга. Ранее мы рассматривали действия над целыми неотрицательными числами. Перейдем теперь к операциям с отрицательными целыми числами, обыкновенными дробями и бесконечными десятичными дробями. Первые две категории легко поддаются обработке

⁵Например, задача разрешимости уравнения $x^{w+2} + y^{w+2} = z^{w+2}$ в натуральных числах – знаменитая «последняя теорема Ферма». Будем просматривать всевозможные четверки x, y, z, w и проверять, выполняется или нет это равенство. Соответствующая МТ останавливается в том случае, когда равенство выполняется.

машинами Тьюринга, причем и числитель, и знаменатель дроби могут быть сколь угодно большими. Все, что нам потребуется – это какой-нибудь подходящий код для знаков «-» и «/», который легко выбрать, используя расширенную двоичную запись (например, «11110» – это «-» и «111110» – это «/»). Тем самым отрицательные целые числа и обыкновенные дроби рассматриваются как конечные наборы натуральных чисел.

То же можно сказать и о конечных или бесконечных периодических десятичных дробях, так как они являются частным случаем обыкновенных дробей. Однако, бесконечные непериодические десятичные выражения, такие как полная запись числа $\pi = 3.14159265358979\dots$, представляют определенные трудности – мы не можем позволить машине Тьюринга работать бесконечно. Но мы можем построить машину, которая в зависимости от входных данных, будет рассчитывать определенный знак дробной части. Подобные замечания относятся и ко многим другим иррациональным числам. Однако оказывается, что некоторые иррациональные числа принципиально не могут быть вычислены с помощью машины Тьюринга [11].

Глава 4

Сети Петри

4.1. События и условия

Поставим перед собой задачу построения модели некоторой дискретной системы. Компоненты этой системы будем представлять абстрактными *событиями*.

Пример 4.1. Событием можно считать выполнение оператора программы, переход триггера из одного состояния в другое, прерывание в операционной системе и т. п.

Реализацию конкретного события назовем *действием*. Набор действий, который необходим для функционирования дискретной системы, образует *процесс*, порождаемый этой системой. Заметим, что в общем случае одна и та же система может функционировать *недетерминированно*, то есть в одних и тех же условиях вести себя по-разному, порождая целое множество процессов.

Если рассматривать традиционные *последовательные модели* дискретных систем, то события необходимо связать с определенными моментами или интервалами времени, в которые происходит одновременное изменение состояний всех компонент системы (изменение *общего состояния системы*). Смена

состояний системы осуществляется последовательно. Такой подход имеет ряд недостатков:

1. Учет состояний *всех* компонент при каждой смене общего состояния делает модель громоздкой.

2. Имеет место потеря причинно-следственных связей между событиями. Например, если два события произошли одновременно, то не известно, случайность это или результат непредусмотренного варианта функционирования модели. Такие понятия, как конфликты между событиями (из-за ресурсов) или ожидание одним событием результатов работы других событий, трудно выражаются в терминах смены состояний системы.

3. В *асинхронных* системах временной интервал, связанный с каким-либо событием, может быть неопределенно большим (напомним, что в синхронных системах этот интервал постоянен). В связи с этим, бывает трудно указать точное время начала и окончания события, а также его продолжительность.

Чтобы избавиться от этих недостатков, откажемся от временных связей – от времени и от тактирования последовательности изменений состояний. Заменим эти понятия причинно-следственными связями между событиями. При этом, если возникнет необходимость осуществить привязку ко времени, моменты или интервалы времени также будем представлять в виде событий.

Взаимодействие событий в больших системах имеет, как правило, сложную динамическую структуру. При описании этих взаимодействий будем указывать не непосредственные связи между событиями, а *ситуации*, при которых данное событие может реализоваться. При этом глобальные ситуации в системе будут формироваться с помощью локальных, называемых **условиями** реализации событий. Каждое условие характеризуется **емкостью**:

- емкость равна 0, если условие не выполнено;
- емкость равна 1, если условие выполнено;
- емкость равна n ($n \in \mathbb{N}$), если условие выполнено с n -кратным запасом.

Очевидно, что определенные условия разрешают реализоваться некоторому событию – **предусловия события**, в свою очередь, реализация события изменяет некоторые условия – **постусловия события**. Таким образом, условия и события взаимодействуют друг с другом.

Подводя итог, получаем, что *дискретную систему можно представить в виде структуры, образованной из элементов двух типов – событий и условий. Сети Петри – это один из вариантов такого представления дискретной системы [7].*

В сетях Петри события и условия представлены абстрактными символами из двух непересекающихся алфавитов. События описываются **множеством переходов**, а условия – **множеством мест**. В графическом представлении сетей переходы изображаются «барьерами», а места – *кружками*. События-переходы и условия-места связаны отношением непосредственной зависимости (или непосредственной причинно-следственной связи), которая изображается при помощи *направленных дуг*. Дуги могут вести как из переходов в места, так и наоборот. **Входные места** – это места, из которых ведут дуги на данный переход. **Выходные места** – это места на которые ведут дуги из данного перехода.

Выполнение условия изображается **разметкой** соответствующего места-кружка *флишками* (маркерами). Число маркеров равно емкости условия (см. рис. 4.1).

| | |
|--------------------|-------------------------------|
| p ○ | условие p не выполнено |
| p ⊙ | условие p выполнено |
| p ⊕ | условие p имеет кратность 2 |
| p ⊕ ⁵ | условие p имеет кратность 5 |

Рис. 4.1. Варианты разметки условий-мест

Пример 4.2. Пример сети Петри представлен на рисунке 4.2 (1). Здесь p_i ($1 \leq i \leq 6$) – условия-места, t_j ($1 \leq j \leq 4$) – события-переходы.

Подводя итог, еще раз сопоставим способы описания основных элементов сети Петри:

события \Rightarrow переходы (барьеры)
 условия \Rightarrow места (кружки)
 причинно-следственные связи \Rightarrow направленные дуги
 предусловия события \Rightarrow входные места
 постусловия события \Rightarrow выходные места
 емкость условий \Rightarrow разметка мест (фишки или маркеры)

Динамика поведения моделируемой дискретной системы описывается функционированием или работой сети Петри. Работа сети – это набор **срабатываний переходов**. Каждое срабатывание перехода соответствует реализации события (действию) и приводит к изменению разметки мест, то есть к локальному изменению условий в системе. Переход может сработать, если выполнены все условия реализации соответствующего события (все предусловия).

Срабатывание перехода – это неделимое действие, которое изменяет разметку входных и выходных мест перехода по следующему правилу: *из каждого входного места изымается по одной фишке (емкость предусловий уменьшается), а в каждое выходное место добавляется по одной фишке (емкость постусловий увеличивается).*

Пример 4.3. Так как оба входных места p_1 и p_2 перехода t_1 (см. рис. 4.2 (1)) содержат фишки, то этот переход может сработать. Разметка входных и выходных мест перехода t_1 после его срабатывания представлена на рисунке 4.2 (2).

Если два (и более) перехода могут сработать и они не имеют общих входных мест, то их срабатывания являются независимыми действиями, осуществляемыми в любой последовательности или параллельно.

Если несколько переходов могут сработать и имеют общее

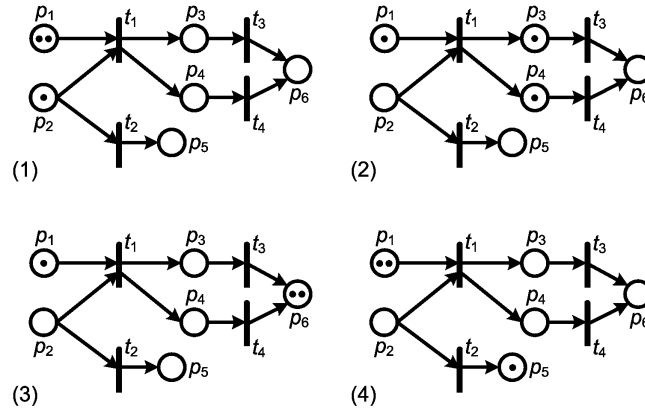


Рис. 4.2. Функционирование сети Петри

входное место (см. переходы t_1 и t_2 на рис. 4.2 (1)), то сработает любой из них, *но только один*. Таким образом, в сети моделируется *конфликт* между событиями, когда реализация одного события может исключить возможность реализации других. В сети не указывается, каким образом конфликт следует разрешить (какое из конфликтующих событий следует реализовать) – в этом проявляется недетерминированность сетей Петри.

Аналогичный конфликт возникает в том случае, когда несколько переходов могут сработать и они имеют общие выходные места (см. переходы t_3 и t_4 на рис. 4.2 (2)).

Сеть останавливается, если ни один из ее переходов не может сработать (см. рис. 4.2 (3) и (4)).

Таким образом, *сети Петри формализуют понятие дискретной динамической системы из событий и условий*. Недетерминированный характер функционирования сети Петри приводит к тому, что система может порождать несколько *параллельных* процессов.

4.2. Формальное определение сетей Петри

Формализуем понятия, введенные в предыдущем параграфе. При этом мы не будем подробно останавливаться на терминологии, относящейся к *теории графов*.

Сетью назовем тройку (P, T, F) , где P – непустое множество элементов сети, называемых *местами*, T – непустое множество элементов сети, называемых *переходами*, $F \subseteq P \times T \cup T \times P$ – отношение **инцидентности**, и для (P, T, F) выполнены следующие условия:

A₁) множества мест и переходов не пересекаются:

$$P \cap T = \emptyset; \quad (4.1)$$

A₂) любой элемент сети инцидентен хотя бы одному элементу другого типа:

$$(F \neq \emptyset) \wedge (\forall x \in P \cup T \exists y \in P \cup T : xFy \vee yFx); \quad (4.2)$$

A₃) сеть не содержит пары мест, которые инцидентны одному и тому же множеству переходов, то есть, если для произвольного элемента сети $x \in X$ ($X = P \cup T$) обозначить через $*x$ множество его **входных элементов** $\{y | yFx\}$, а через x^* – множество его **выходных элементов** $\{y | xFy\}$, то

$$\forall p_1, p_2 \in P : (*p_1 = *p_2) \wedge (p_1^* = p_2^*) \Rightarrow (p_1 = p_2). \quad (4.3)$$

Графически сеть представляется в виде *двудольного ориентированного графа* (в общем случае бесконечного) с двумя типами вершин. Как отмечалось ранее, вершины-места изображаются кружками, вершины-переходы – барьерами (или квадратами). Из вершины x в вершину y ведет дуга тогда и только тогда, когда имеет место xFy .

Сеть описывает только статику дискретной системы. Для привнесения динамики вводится разметка мест, которая моделирует выполнение условий.

Сеть Петри – это набор

$$N = \{P, T, F, W, M_0\}, \quad (4.4)$$

где (P, T, F) – конечная сеть (множество $X = P \cup T$ конечно), $W : F \rightarrow \mathbb{N}$ – функция, называемая **кратностью дуг**, $M_0 : P \rightarrow \mathbb{N} \cup \{0\}$ – функция, называемая **начальной разметкой**.

Функция W сопоставляет каждой дуге число $n > 0$ – **кратность дуги**. Если $n > 1$, то в графическом представлении сети число n выписывается рядом с короткой чертой, пересекающей дугу (можно такую дугу заменить пучком из n дуг, соединяющих соответствующие элементы сети). Если кратность всех дуг равна 1, то такая сеть называется **ординарной**.

Функция M_0 сопоставляет каждому месту $p \in P$ некоторое число $M_0(p) \in \mathbb{N} \cup \{0\}$ – **начальная разметка места**. Еще раз повторим, что на графе сети разметка места p изображается помещением в вершину-кружок числа $M_0(p)$ или, если это число невелико, соответствующего числа фишек (точек) (см. рис. 4.1).

4.3. Функционирование сетей Петри

Функционирование сети Петри описывается при помощи *множества последовательностей срабатываний* и *множества достижимых в сети разметок*. Эти понятия определяются через правила срабатывания переходов сети.

Разметка сети N – это функция $M : P \rightarrow \mathbb{N} \cup \{0\}$. Если предположить, что все места сети N строго упорядочены: $P = (p_1, \dots, p_n)$, то разметку M сети (в том числе и M_0) можно задать как вектор чисел $M = (m_1, \dots, m_n)$ такой, что для любого i ($1 \leq i \leq n$) $m_i = M(p_i)$.

На основе отношения инцидентности F и функции кратности дуг W определим **функцию инцидентности** (которую обозначим тем же символом F) $F : P \times T \cup T \times P \rightarrow \mathbb{N} \cup \{0\}$

такую, что

$$F(x, y) = \begin{cases} n, & \text{если } (xFy) \wedge (W(x, y) = n); \\ 0, & \text{если } \neg(xFy). \end{cases} \quad (4.5)$$

Если места сети упорядочены, то можно каждому переходу t сопоставить два целочисленных вектора ${}^*F(t)$ и $F^*(t)$ длиной n , где $n = |P|$:

$${}^*F(t) = (b_1, \dots, b_n), \text{ где } b_i = F(p_i, t) \quad (4.6)$$

$$F^*(t) = (b_1, \dots, b_n), \text{ где } b_i = F(t, p_i) \quad (4.7)$$

Переход t может сработать при некоторой разметке M сети N , если

$$\forall p \in {}^*t : M(p) \geq F(p, t), \quad (4.8)$$

то есть каждое входное место p перехода t имеет разметку, не меньшую, чем кратность дуги, соединяющей p и t . В векторной форме условие (4.8) принимает вид:

$$M \geq {}^*F(t). \quad (4.9)$$

Срабатывание перехода t при разметке M порождает разметку M' по правилу:

$$\forall p \in P : M'(p) = M(p) - F(p, t) + F(t, p) \quad (4.10)$$

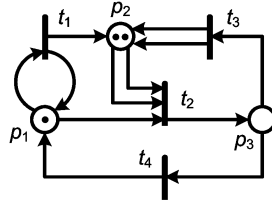
или в векторном виде:

$$M' = M - {}^*F(t) + F^*(t). \quad (4.11)$$

На множестве разметок можно ввести отношение **непосредственного следования** разметок (\mapsto):

$$M \mapsto M' \iff \exists t \in T : (M \geq {}^*F(t)) \wedge (M' = M - {}^*F(t) + F^*(t)). \quad (4.12)$$

Будем также записывать $M \xrightarrow{t} M'$, если M' непосредственно следует после M в результате срабатывания перехода t .

Рис. 4.3. Сеть Петри N_1

Пример 4.4. В сети Петри N_1 , изображенной на рисунке 4.3, $P = (p_1, p_2, p_3)$, $T = (t_1, t_2, t_3, t_4)$.

Функция инцидентности F задается таблицами 4.1, 4.2. В этих таблицах на пересечении строки x и столбца y стоит число $F(x, y)$. Заметим, что в таблице 4.1 строки представляют собой векторы $F^*(t_i)$, а столбцы таблицы 4.2 – это векторы ${}^*F(t_i)$.

Таблица 4.1. Функция инцидентности $F(x, y)$, при условии, что $x \in T$, $y \in P$

| | p_1 | p_2 | p_3 |
|-------|-------|-------|-------|
| t_1 | 1 | 1 | 0 |
| t_2 | 0 | 0 | 1 |
| t_3 | 0 | 2 | 0 |
| t_4 | 1 | 0 | 0 |

Таблица 4.2. Функция инцидентности $F(x, y)$, при условии, что $x \in P$, $y \in T$

| | t_1 | t_2 | t_3 | t_4 |
|-------|-------|-------|-------|-------|
| p_1 | 1 | 1 | 0 | 0 |
| p_2 | 0 | 2 | 0 | 0 |
| p_3 | 0 | 0 | 1 | 1 |

Начальная разметка M_0 задается следующим образом: $M_0(p_1) = 1$, $M_0(p_2) = 2$, $M_0(p_3) = 0$ или, в векторной форме: $M_0 = (1, 2, 0)$.

При разметке M_0 могут сработать переходы t_1 и t_2 , так как $M_0 = (1, 2, 0) \geq {}^*F(t_1) = (1, 0, 0)$ и $M_0 \geq {}^*F(t_2) = (1, 2, 0)$. Переходы t_3 и t_4 сработать не могут, так как $M_0 \not\geq {}^*F(t_3) = (0, 0, 1)$ и $M_0 \not\geq {}^*F(t_4) = (0, 0, 1)$.

В результате срабатывания перехода t_1 разметка M_0 изменится на разметку $(1,3,0)$, а в результате срабатывания перехода t_2 разметка M_0 изменится на разметку $(0,0,1)$.

Будем говорить, что **разметка M' достижима от разметки M** , если существует последовательность разметок M, M_1, M_2, \dots, M' и слово $\tau = t_{i_1} t_{i_2} t_{i_3} \dots t_{i_k}$ в алфавите T такое, что

$$M \xrightarrow{t_{i_1}} M_1 \xrightarrow{t_{i_2}} M_2 \xrightarrow{t_{i_3}} \dots \xrightarrow{t_{i_k}} M'. \quad (4.13)$$

Слово τ называется **последовательностью срабатываний**, ведущих от M к M' , и имеет место запись $M \xrightarrow{\tau} M'$.

Множество разметок, достижимых в сети N от разметки M , обозначим через $R(N, M)$ ($R(N, M) = \{M' | M \xrightarrow{\tau} M'\}$). Множество $R(N) = R(N, M_0)$ (множество разметок, достижимых в N от начальной разметки M_0) называется **множеством достижимых разметок** сети N . Заметим, что $M \in R(N, M)$ и $M_0 \in R(N)$.

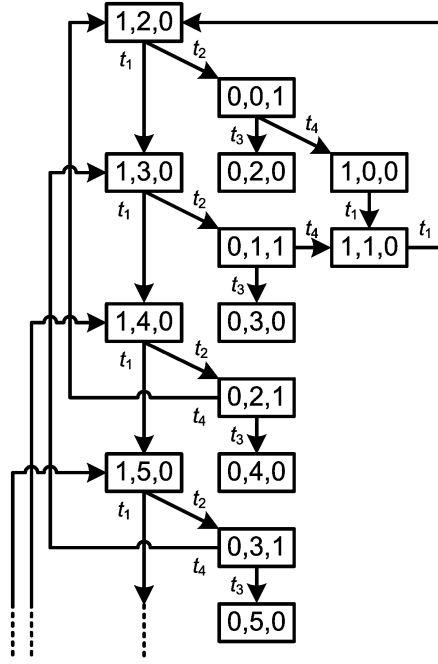
Множеством последовательностей срабатываний сети N или **свободным языком сети N** называется множество

$$L(N) = \{\tau \in T^* | \exists M \in R(N) : M_0 \xrightarrow{\tau} M\}, \quad (4.14)$$

то есть множество всех последовательностей срабатываний, ведущих от M_0 к каждой достижимой в N разметке.

Возможные изменения разметок сети N , происходящие в результате срабатывания ее переходов, представим в виде **графа разметок** – ориентированного графа, множество вершин которого образовано множеством $R(N)$ достижимых в N разметок. При этом из вершины M в вершину M' ведет дуга, отмеченная символом перехода t , тогда и только тогда, когда $M \xrightarrow{t} M'$. На рисунке 4.4 представлен начальный фрагмент графа разметок сети N_1 . Этот граф бесконечен, так как для рассматриваемой сети бесконечно множество достижимых разметок $R(N_1)$.

Разметка $M \in R(N)$ называется **тупиковой**, если в сети N не существует ни одного перехода, который может срабо-



4.4. Свойства сетей Петри

Заметим, что при работе сети Петри, некоторые ее места могут накапливать неограниченное число фишек. (Например, место p_2 на рис. 4.3). Если интерпртировать места как накопители (буферы) данных, то естественно потребовать, чтобы при функционировании сети не происходило бы переполнение этих накопителей, которые в реальных условиях имеют ограниченный объем. Итак, место p называется **ограниченным**, если существует число n такое, что для любой достижимой в сети разметки M справедливо неравенство $M(p) \leq n$. **Сеть ограничена**, если любое ее место ограничено.

Место p называется **безопасным**, если оно ограничено и $n \leq 1$, иными словами, $\forall M \in R(N) : M(p) \leq 1$. **Сеть безопасна**, если все ее места безопасны.

В **консервативной сети** сумма фишек во всех ее местах остается постоянной при функционировании сети, то есть $\forall M_1, M_2 \in R(N) : \sum_{p \in P} M_1(p) = \sum_{p \in P} M_2(p)$. В консервативной сети для любого перехода t справедливо равенство: $|*t| = |t*|$.

Переход t в сети Петри называется **потенциально живым при разметке** $M \in R(N)$, если

$$\exists M' \in R(N, M) : M' \geq *F(t), \quad (4.15)$$

то есть существует достижимая от M разметка M' , при которой переход t может сработать. Если $M = M_0$, то t называется **потенциально живым в сети** N . Переход t в сети Петри называется **живым**, если

$$\forall M \in R(N) \exists M' \in R(N, M) : M' \geq *F(t), \quad (4.16)$$

то есть переход t является потенциально живым при любой достижимой в сети разметке. Обратите внимание, что переход может быть потенциально живым в сети, но не являться живым. **Живая сеть** – это сеть, в которой все переходы живы.

Переход t – **мертвый при разметке** M , если он не является потенциально живым при M . Переход t **мертвый**, ес-

ли он мертв при любой достижимой в сети разметке. Переход t – **потенциально мертвый**, если существует разметка $M \in R(N)$ такая, что при любой разметке $M' \in R(N, M)$ переход t не может сработать¹. Разметка M в этом случае называется t -**тупиковой**. Напомним, что если разметка t -тупиковая для всех $t \in T$, то она является **тупиковой**. Если M – тупиковая разметка, то $R(N, M) = \{M\}$.

Пример 4.5. В сети на рисунке 4.2 все переходы потенциально живы и все переходы потенциально мертвы, так как в ней достижима тупиковая разметка (см. рис. 4.2 (3), (4)). Эта сеть не может быть живой, так как в ней достижима тупиковая разметка. Сеть на рисунке 4.3 также не является живой, так как в ней достижимы тупиковые разметки вида $(0, n, 0)$ ($n \geq 2$).

Переход t называется **устойчивым**, если $\forall M \in R(N), \forall t' \in T \setminus \{t\}$:

$$(M \geq {}^*F(t)) \wedge (M \geq {}^*F(t')) \Rightarrow (M \geq {}^*F(t) + {}^*F(t')), \quad (4.17)$$

то есть, если переход t может сработать, то никакой другой переход не может, сработав, лишить его этой возможности.

Сеть устойчива, если все ее переходы устойчивы.

Конечная цель специальной теории сетей Петри – построение алгоритмов анализа, синтеза и преобразования дискретных систем, моделируемых сетями. В частности, полезно найти алгоритмы, с помощью которых для любой предъявленной сети можно определить, обладает ли она интересующим нас свойством – является ли она ограниченной, живой, устойчивой и т. п. Эти вопросы формулируются как **массовые** алгоритмические проблемы для сетей: *существует ли требуемый алгоритм (разрешима ли данная массовая задача)*. Следующие

¹Заметим, что условия срабатываеия потенциально мертвого перехода t никогда не выполняются после того, как в сети была достигнута разметка M . Это означает, что в моделируемых системах могут появиться ситуации, тупиковые для некоторых событий. Например, в операционных системах подобные случаи имеют место при взаимных блокировках процессов (deadlocks, см. § 4.6).

теоремы, доказательство которых можно найти в [7], решают ряд подобных проблем.

Теорема 4.1. *Проблема ограниченности сети Петри разрешима.*

Теорема 4.2. *Проблема безопасности сети Петри разрешима.*

Теорема 4.3. *Проблема потенциальной живости переходов разрешима.*

Теорема 4.4. *Существует алгоритм, с помощью которого можно узнать, получит ли данное место в сети хотя бы одну фишку.*

Теорема 4.5. *Существует алгоритм, с помощью которого можно узнать, может ли данный переход сработать сколь угодно большое число раз.*

Пусть задан класс сетей \mathcal{N} , которые имеют одно и то же множество мест (или их множества мест *изоморфны*). Если для $N_1, N_2 \in \mathcal{N}$: $R(N_1) \subseteq R(N_2)$, то говорят, что имеет место *R-включение*. В случае $R(N_1) = R(N_2)$ имеет место *R-эквивалентность*.

Теорема 4.6. *Проблемы R-включения и R-эквивалентности не являются разрешимыми.*

Теорема 4.7. *Проблема живости сети Петри и проблема достижимости в ней произвольной разметки эквивалентны.*

Гипотеза о разрешимости проблемы достижимости имела неоднократные попытки доказательства, но во всех из них были найдены ошибки (последней работой в этом ряду была статья Майера, и наличие в ней ошибок пока не установлено) [7].

4.5. Языки сетей Петри

Напомним, что функционирование сети Петри описывается при помощи множества последовательностей срабатываний переходов $L(N)$. Если T^* – множество всех слов, составленных из символов переходов сети, то, очевидно, $L(N) \subseteq T^*$ – язык в алфавите T . Как было определено ранее, этот язык называется свободным языком сети Петри.

В сети все символы переходов различны. Но в систеах, моделируемых сетями, часто удобно считать разные события одинаковыми (например, повторение одного и того же оператора в программе). Для этого можно использовать специальные пометки, для выделения «одинаковых» и «разных» переходов сети. Эти пометки представляют собой символы, принадлежащие некоторому алфавиту A , в общем случае отличному от T . Такие сети называются *помеченными*. Если символы переходов в последовательностях срабатываний заменить на помечающие символы, свободный язык сети преобразуется в некоторый другой язык, порождаемый этой же сетью. Теперь дадим более строгие определения.

Помеченная сеть Петри – это пара (N, Σ) , где N – сеть Петри, $\Sigma : T \rightarrow A$ – **помечающая функция** над некоторым алфавитом A . Если Σ – частичная функция, то есть некоторым переходам не сопоставляется никакой символ из A , то эти непомеченные переходы называются ε -**переходами** и помечаются одним и тем же «пустым» символом ε . На практике ε -переходам соответствуют вспомогательные переходы, не связанные непосредственно с событиями системы.

Расширение помечающей функции на последовательности срабатываний определяется следующим образом:

$$\Sigma(\tau t) = \begin{cases} \Sigma(\tau)\Sigma(t), & \text{если } \Sigma(t) \text{ определено;} \\ \Sigma(\tau), & \text{в противном случае.} \end{cases} \quad (4.17)$$

Здесь $\tau t \in T^*$. При этом $\Sigma(\varepsilon) = \varepsilon$.

Если $\tau \in T^*$ – последовательность срабатываний сети Петри N , а (N, Σ) – помеченная сеть, то $\Sigma(\tau) \in A^*$ – **помечающая последовательность**. Если $L(N)$ – свободный язык сети N , то множество

$$\{\Sigma(\tau) | \tau \in L(N)\} \quad (4.18)$$

образует (**префиксный**) **язык помеченной сети** (N, Σ) .

В некоторых приложениях рассматривается не свободный язык сети Петри, включающий все ее последовательности срабатываний, а его подмножество **терминальных последовательностей**, которое состоит из всех последовательностей,

ведущих от начальной разметки M_0 к некоторой фиксированной **терминальной разметке** M_t , то есть множество

$$L(N, M_t) = \{\tau \in T^* \mid M_0 \xrightarrow{\tau} M_t\}. \quad (4.19)$$

Множество $L(N, M_t)$ образует **свободный терминальный язык сети** N . Множество

$$\{\Sigma(\tau) \mid \tau \in L(N, M_t)\} \quad (4.20)$$

образует **терминальный язык помеченной сети** (N, Σ) .

В теории алгоритмов и автоматов рассматриваются различные абстрактные системы (машины, автоматы), предназначенные для моделирования функционирования дискретных систем. Их способность адекватно описывать сложное поведение моделируемых систем часто характеризуется классами порождаемых ими языков. Эти языки, как и в случае языков сетей Петри, определенным образом кодируют разные возможные способы функционирования систем.

Пусть системы из класса S_1 порождают класс языков \mathcal{L}_1 , а системы из класса S_2 порождают класс языков \mathcal{L}_2 . Если $\mathcal{L}_1 \supseteq \mathcal{L}_2$, то говорят, что класс систем S_1 мощнее класса систем S_2 . Если же $\mathcal{L}_1 \supset \mathcal{L}_2$, то класс систем S_1 строго мощнее класса систем S_2 . Классы S_1 и S_2 равномощны, если $\mathcal{L}_1 = \mathcal{L}_2$. Введем ряд обозначений.

1. \mathcal{L}^ε – класс префиксных языков помеченных сетей Петри.
2. $\mathcal{L} \subset \mathcal{L}^\varepsilon$ – класс префиксных языков помеченных сетей Петри, которые не содержат ε -переходов.
3. $\mathcal{L}^f \subset \mathcal{L}$ – класс свободных языков сетей Петри $(\Sigma : T \rightarrow T)$.
4. $\mathcal{L}_0^\varepsilon$ – класс терминальных языков помеченных сетей Петри.
5. $\mathcal{L}_0 \subset \mathcal{L}_0^\varepsilon$ – класс терминальных языков помеченных сетей Петри, которые не содержат ε -переходов.
6. $\mathcal{L}_0^f \subset \mathcal{L}_0$ – класс свободных терминальных языков сетей Петри $(\Sigma : T \rightarrow T)$.

Имеют место следующие включения:

$$\begin{array}{ccccc} \mathcal{L}^f & \subset & \mathcal{L} & \subset & \mathcal{L}^\varepsilon \\ \cap & & \cap & & \cap \\ \mathcal{L}_0^f & \subset & \mathcal{L}_0 & \subset & \mathcal{L}_0^\varepsilon \end{array}$$

В заключении параграфа приведем теорему, связывающую сети Петри и абстрактные системы, рассмотренные ранее [7].

Теорема 4.8. *Класс помеченных сетей Петри строго мощнее класса конечных автоматов, не сравним с классом магазинных автоматов и строго менее мощен, чем класс машин Тьюринга.*

4.6. Сети Петри и программирование

Среди приложений сетей Петри к задачам моделирования дискретных систем наибольшее развитие получило направление, связанное с использованием аппарата сетей Петри для изучения и описания структурной динамики программ, и в первую очередь – для параллельного программирования.

Под **структурой управления** программы будем понимать набор базовых структурных единиц – *операторов* (а также модулей, подмодулей) и *управляющих примитивов* (условных операторов, операторов перехода, операторов цикла, специальных разделителей, указывающих порядок следования операторов, и т. п.).

Обычная для алгоритмических языков **последовательная структура управления** представляет собой совокупность операторов различных типов, которые связаны некоторым отношением следования. Начальный оператор единственен в программе и не следует ни за каким другим оператором. После безусловного оператора следует ровно один оператор. После условного оператора следует два оператора. После оператора выбора следует некоторая совокупность операторов. После заключительного оператора не следует никакой другой оператор.

Последовательная структура управления может быть представлена как связанная автоматная сеть Петри с определенным топологическим ограничением, элементы которой интерпретируются специальным образом. Сеть Петри называется **автоматной**, если $\forall t \in T : |*t| = |t^*| = 1$, то есть каждый переход автоматной сети имеет ровно одно входное и ровно одно выходное место. На эту сеть накладываются следующие ограничения: имеются единственные *головное* и *заключительное* места, все остальные места делятся на *безусловные* с одним выходным переходом и *условные* с двумя выходными переходами. Если в последовательной программе оператор a' следует за оператором a , то в сети Петри этим операторам соответствуют переходы, связанные дугами с общим местом, которое является выходным для a и входным для a' .

Пример 4.6. Рассмотрим программу вычисления факториала:

```
begin integer x, y;
input(x);
y:=1;
while (x<>0) do begin
    y:=y × x;
    x:=x-1;
end;
output(y)
end.
```

Сеть Петри, интерпретирующая структуру управления этой программы, представлена на рисунке 4.5.

Эта сеть описывает только структуру последовательного управления, но не моделирует сам механизм управления. Например, когда место p_4 получает фишку, то эта фишка сообщает лишь о том, что оператор выполнен, но не несет информацию о вычисленном значении условия. В результате выбор последующего оператора моделируется недетерминированной альтернативой (оператор $\text{output}(y)$ или оператор $y:=y \times x$).

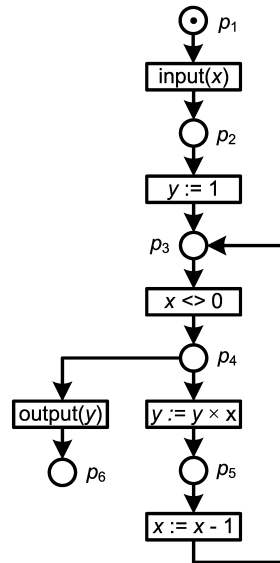


Рис. 4.5. Сеть Петри последовательной структуры

Для того, чтобы иметь средства моделирования условного управления, можно выбрать, например, *ингибиторную сеть* – сеть Петри, дополненную специальной функцией инцидентности $F_I : P \times T \rightarrow \{0, 1\}$, которая определяет дополнительные ингибиторные дуги. Подробное описание ингибиторных сетей можно найти, например, в [7].

Последовательно-параллельные структуры управления представляют собой обобщение последовательных за счет включения в набор управляющих примитивов специальных операторов или указателей, которые выделяют *параллельные ветви* программы, исполняемые независимо друг от друга. Ветви имеют общие начало – точку расхождения ветвей – и конец – точку схождения ветвей. Совокупность ветвей с общим началом и концом образуют *сегмент* параллельной структуры. При исполнении программы процесс вычислений продви-

гается до начала первого сегмента, после чего «расщепляется» на столько копий, сколько ветвей содержит сегмент. Каждый из параллельных процессов вычислений ветвей протекает независимо от других и, достигая конца ветви, останавливается, ожидая, пока все остальные процессы в сегменте не достигнут конца сегмента. В конце сегмента все копии процесса «сливаются» в один. Параллельные ветви сегментов могут, в свою очередь, содержать сегменты.

Пример 4.7. Пример сети Петри последовательно-параллельной структуры представлен на рисунке 4.6.

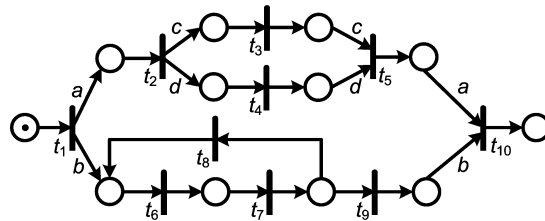


Рис. 4.6. Сеть Петри последовательно-параллельной структуры

Характерным свойством «чистой» последовательно-параллельной структуры управления является отсутствие каких-либо взаимодействий между параллельными ветвями, переходов-скачков из одной ветви в другую. В реальных условиях существует необходимость организации взаимодействий параллельно протекающих процессов исполнения ветвей.

Для организации взаимодействий требуются дополнительные программные средства, обеспечивающие само взаимодействие. В большинстве случаев нужны средства, которые позволяли бы приостанавливать процесс исполнения некоторой ветви, пока не придут данные другой ветви или пока другие ветви не освободят общий ресурс. Тот отрезок ветви, на котором требуется «захват» общего ресурса, называется **критическим интервалом** ветви.

В настоящее время для этих целей в языках программирования широко используется **механизм семафоров**, включающий в себя специальные переменные нового типа (**семафры**) и **две операции** P и V , аргументами которых могут быть только переменные типа семафоров. Область значений семафора – целые неотрицательные числа. Если область значений сужена до $\{0, 1\}$, то семафор называется бинарным. Операция V изменяет значение s семафора на $s + 1$. Действие операции P определяется следующим образом:

- если $s \neq 0$, то P уменьшает значение s на 1;
- если $s = 0$, то P не изменяет значение s и не завершается до тех пор, пока некоторая другая ветвь не изменит значение s с помощью операции V .

Существенным является тот факт, что операции P и V считаются «неделимыми». По отношению к V это означает следующее. Операция V состоит из трех частей:

- считывание значения семафора из памяти;
- увеличение значения семафора;
- помещение нового значения в память.

Неделимость $V(s)$ заключается в том, что с самого начала выполнения этой операции до ее завершения доступ к переменной-семафору s запрещен для всех других операций. Аналогично дело обстоит с операцией P (для случая $s \neq 0$).

Обычно P предшествует критическому интервалу ветви, а V завершает его. В каждый момент времени, когда значение семафора s изменяется с 0 на 1, только одна из операций P может завершиться и разрешить вход в критический интервал только одному процессу исполнения ветви.

Пример 4.8. Приведем пример параллельной программы на паскалеподобном языке. Скобки **parbegin...parend** выделяют сегмент с двумя параллельными ветвями *process 1* и *process 2*, разделенными запятой, и семафором s . Каждая из параллельных ветвей представляет собой последовательный цикл, содержащий критический интервал *critical section* и остальную часть цикла *remainder of cycle*:


```

begin
  semaphore s;
  s:=1;
parbegin
  process 1:
  while (1 = 1) do begin
    P(s);
    critical section 1;
    V(s);
    remainder of cycle 1;
  end,
  process 2:
  while (1 = 1) do begin
    P(s);
    critical section 2;
    V(s);
    remainder of cycle 2;
  end
end
end.

```

В этом примере решается задача взаимного исключения исполнения критических интервалов двух циклических процессов параллельных ветвей. Данная последовательно-параллельная структура управления наглядно изображается с помощью сети Петри, изображенной на рисунке 4.7.

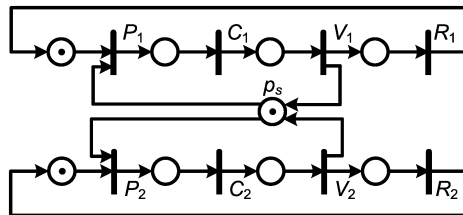


Рис. 4.7. Сеть Петри последовательно-параллельной структуры и механизм семафоров

Здесь переход P_i соответствует операции P в ветви i , переход C_i – критическому интервалу в ветви i , переход V_i – операции V в ветви i , переход R_i – остатку цикла в ветви i ($i = 1, 2$). Место p_s соответствует семафору s .

К сожалению, при программировании сложных взаимодействий в параллельных программах число потенциальных ошибочных ситуаций возрастает по сравнению с последовательными программами. Наряду с обычным заикливанием наиболее частыми ошибками являются взаимные блокировки и отталкивания параллельных процессов.

Пример 4.9. *Взаимная блокировка (deadlock)* возникает, например, в следующей ситуации. Пусть две ветви A и B требуют при своей работе доступа к двум общим ресурсам. Типичная ошибочная ситуация, в которой может возникнуть блокировка, показана при помощи сети Петри, изображенной на рисунке 4.8 (1). Пусть процесс исполнения ветви A (переходы t_1, t_2, t_3) после выполнения действия t_1 захватил один из ресурсов, изображенный фишкой в месте p_7 , после чего место p_7 имеет разметку 0. Процесс исполнения ветви B (переходы t_4, t_5, t_6) после выполнения действия t_4 захватил другой ресурс (место p_8 имеет разметку 0). Оба процесса остановились: A – перед t_2 , B – перед t_5 , так как для выполнения следующих действий каждому из них нужен второй ресурс, захваченный партнером (см. рис. 4.8 (2)). При этом ни один из процессов не может вернуть захваченный ресурс – возникла блокировка.

Следующий подход, противоположный последовательно-параллельной структуре управления, называется методом **асинхронного программирования**. Он состоит в том, что все операторы программы изначально считаются независимыми, параллельно исполняемыми, а формирование вычислительных процессов организуется с помощью ограничений, накладываемых на порядок исполнения операторов. Ограничения представляют собой явно указываемые (или неявно подразумеваемые) индивидуальные *условия готовности*, связанные с каждым оператором. Условия готовности динамически

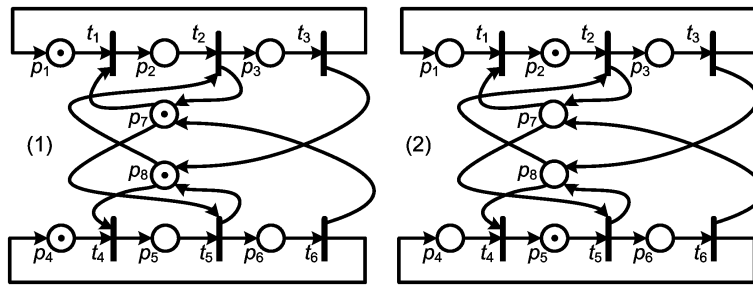


Рис. 4.8. Сеть Петри, иллюстрирующая возникновение ситуации deadlock

проверяются и разрешают или не разрешают (но не предписывают) начать выполнение оператора, с которым данные условия связаны. В зависимости от того, в каких терминах формулируются условия готовности, выделяются различные типы асинхронного управления.

Событийное управление основано на том, что условия готовности формулируются как логические функции от некоторых *событий*. Событием может быть инициирование или завершение какого-либо оператора программы (программные события), прерывание в системе или сигнал об освобождении некоторого ее ресурса (системные события).

При **потокном управлении** действие (оператор или операция) программы может выполняться, если готовы все необходимые для него аргументы (операнды). Условие готовности в этом случае носит стандартный характер и не выписывается явно в программе, а неявно подразумевается. В **обратном потокном управлении** действие может выполняться, если его результат необходим в качестве аргумента для некоторого другого действия. В этом случае второе действие как бы вызывает первое в качестве процедуры.

Оба типа асинхронного управления удобно и наглядно описываются сетями Петри [7]. Примеры событийной и потоковой сетей Петри представлены на рисунках 4.9, 4.10.

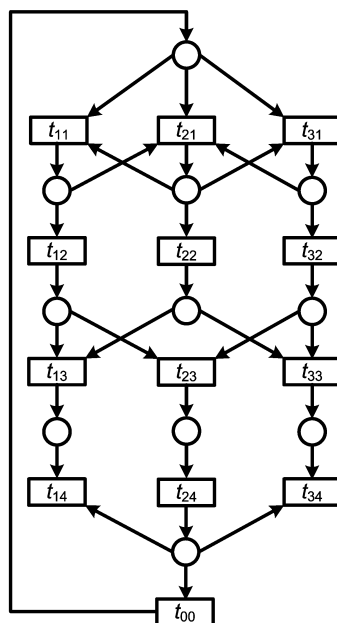


Рис. 4.9. Событийная сеть Петри

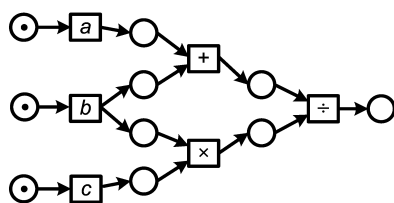


Рис. 4.10. Потокосная сеть Петри

Упражнение 4.1. Докажите, что структуру управления, представленную на рисунке 4.9, не удастся запрограммировать с помощью обычных семафоров без использования дополнительных условных операторов (*указание*: используйте тот факт, что неделимая операция P применима только к одному семафору).

Глава 5

Клеточные автоматы

5.1. Задание клеточного автомата

Клеточные автоматы являются дискретными динамическими системами, поведение которых полностью определяется в терминах локальных зависимостей ¹.

Клеточный автомат можно представлять в виде некоторого «стилизованного мира». Пространством этого мира является равномерная сетка, каждая ячейка которой (или *клетка*), содержит несколько битов данных. Время в этом мире дискретно. Законы мира выражаются единственным набором правил, например, справочной таблицей, по которой любая клетка на каждом временном шаге вычисляет свое новое состояние по состояниям ее близких соседей. Таким образом, законы системы являются *локальными* и *повсюду одинаковыми*. *Локальность* означает следующее: для того чтобы узнать, что произойдет здесь мгновение спустя, достаточно посмотреть на состояние ближайшего окружения, никакое дальное действие не допускается. *Одинаковость* проявляется в том смысле, что отличить одно место от другого можно только по

¹В этом смысле клеточные автоматы в информатике являются аналогом физического понятия «поля».

форме «ландшафта», но не по какой-то разнице в законах [15].

Что нужно для клеточного автомата?

1. *Задать начальные условия.* К примеру, это может быть простая «картинка», занимающая несколько клеток сетки рядом с центром.

2. *Задать законы изменения.* В качестве примера рассмотрим алгоритм закраски клеток одной краской A1:

Выберите клетку и посмотрите на область 3×3 , центром которой она является, – ее «окрестность». Если в этой области расположено в точности три закрашенных клетки, то выбранная центральная клетка отмечается; эта клетка отмечается и в том случае, если она расположена непосредственно над закрашенной клеткой. Описанная процедура повторяется для каждой клетки решетки. После этого все отмеченные клетки закрашиваются. Этот алгоритм моделирует процесс растекания чернильного пятна на промокашке.

Отметим, что в общем случае «окрестность» и число используемых красок могут быть и другими. Но число соседей и число возможных *состояний клетки* (то есть число цветов) должны быть конечными.

3. *Задать форму сетки.* Чаще всего используется квадратная сетка. Но она может быть и иной формы, например, гексагональной или, даже, трехмерной.

Заметим, что общность и гибкость клеточно-автоматного подхода к синтезу систем достигается за счет большого количества переменных (по одной на клетку). Правда, взаимодействия они только локально и единообразно.

Чтобы синтезировать структуры значительной сложности, а также чтобы моделировать их взаимодействие и эволюционирование, необходимо не только использовать большое количество клеток, но и позволить автомату работать на протяжении большого количества шагов.

Событием назовем процесс обновления одной клетки. Для элементарных научных проблем может потребоваться вычислить несколько миллиардов событий, для более сложных приложений – 10^{12} – 10^{15} событий, в действительности преде-

лы устанавливаются тем, сколько мы *можем* выполнить, а не тем, сколько хотим. В связи с этим обычные (последовательные) компьютеры здесь, мягко говоря, не эффективны.

С другой стороны, структура клеточного автомата идеально пригодна для реализации на ЭВМ, обладающей высокой степенью параллелизма, локальными и единообразными взаимодействиями (на вычислительных машинах с так называемой «не-фон-неймановской» архитектурой²). Подходящая архитектура позволяет при моделировании клеточных автоматов за приемлемую цену достигнуть эффективности на несколько порядков выше, чем для обычного компьютера [15]. Так называемые *машины клеточных автоматов* представляют собой лабораторные установки, в которых идеи и принципы теории клеточных автоматов могут быть применены на практике.

В чем же принципиальное отличие клеточных автоматов от рассмотренных ранее абстрактных систем? В последовательных моделях вычислителей, например таких, как машина Тьюринга, различают *структурную часть*, которая фиксирована, и *данные*, которыми вычислитель оперирует – они являются переменными. Вычислитель не может оперировать своей собственной «материальной частью», он не может себя расширять или модифицировать, строить другие вычислители.

Чтобы обеспечить более реалистические модели поведения сложных систем можно использовать клеточно-автоматный подход. В клеточном автомате и «пассивные данные», и «вычислительные устройства» собираются из одного типа структурных элементов и подчиняются одним и тем же локально действующим законам. Вычисление и конструирование – это просто две возможные формы активности.

Клеточные автоматы дают полезные модели для различных исследований, в частности, они представляют естественный путь изучения эволюции больших физических систем. Кроме того, клеточные автоматы образуют общую парадиг-

²Как не парадоксально, клеточные автоматы ввел Дж. фон Нейман в конце сороковых годов двадцатого века.

му параллельных вычислений, подобно тому как это делают машины Тьюринга для последовательных вычислений.

5.2. Игра «Жизнь»

В 1970 году математик Джон Конвей придумал игру «Жизнь», являющуюся по сути клеточным автоматом, которая привлекла внимание не только любителей, но и профессионалов. «Жизнь» описывает популяцию стилизованных организмов, развивающуюся во времени под действием противоборствующих тенденций размножения и вымирания.

«Житель» этой популяции представлен клеткой в состоянии 1 («*живая*» или закрашенная (черная) клетка), клетка в состоянии 0 («*мертвая*» или незакрашенная (белая) клетка) – это пустое пространство. На каждом шаге любая клетка реагирует на состояние ее непосредственного окружения (восьми ближайших соседей) согласно правилам алгоритма A2:

1. Смерть. Живая клетка остается живой, только когда она окружена двумя или тремя живыми соседями, в противном случае она будет чувствовать или «перенаселенность», или «одиночество» и умрет.

2. Рождение. Мертвая клетка обретет жизнь, если будет окружена в точности тремя живыми соседями. Таким образом, рождение вызывается встречей трех рдителей.

Зададим в качестве начальных условий игры «жизнь» экран с «первичным бульоном», в котором живые и мертвые клетки распределены случайно и в равных количествах. После нескольких десятков шагов «бешеной» активности популяция заметно поредет, затем большая часть экрана станет неподвижной, за исключением нескольких мест, где жизнь будет «тлеть» и иногда будут происходить «бои местного значения». В конце концов вся активность может угаснуть, кроме нескольких изолированных «мигалок», развивающихся циклически с коротким периодом.

Продолжая наблюдение за развитием игры, можно заметить, что активные области время от времени извергают

небольшой пульсирующий объект, именуемый *глайдером*, который постоянно убегает прочь по диагональному пути, пока не наткнется на что-либо еще [15].

5.3. Правила клеточного автомата

Посмотрим на законы функционирования или *правила* клеточного автомата с точки зрения их формата, а не содержания. Правила клеточного автомата – это функция конечного множества. Следовательно, они могут быть полностью заданы справочной таблицей, в которой каждому входному значению поставлено в соответствие выходное значение.

Для простоты будем рассматривать клеточные автоматы, которые используют клетки с двумя состояниями и окрестностью из четырех соседей. Правила должны определять новое состояние центральной клетки для каждой окрестности. Состояния центральной клетки и ее четырех соседей запишем в строку согласно последовательности: east, west, south, north, center (сокращенно: «ewsnс»). Очевидно, что для задания клеточного автомата достаточно заполнить таблицу, записывая 0 или 1 в каждой из $32 = 2^5$ позиций.

Отметим, что подобную таблицу можно заполнить 2^{32} способами. Для формата игры «Жизнь» это число возрастает до $2^{2^9} = 2^{512}$, что равно квадрату примерного числа элементарных частиц во Вселенной!

Упражнение 5.1. Напишите программу, моделирующую клеточный автомат, заданный таблицей 5.1. Проследите его поведение для различных начальных конфигураций, в частности, для «островка нулей в море единиц».

Если говорить о том, в каком виде удобнее формулировать правила для нашего восприятия, то естественно, что громоздкие таблицы теряют наглядность. В некоторых случаях удается найти более компактное словесное описание или функциональную зависимость, которые позволяют задать правила клеточного автомата.

Таблица 5.1. Правила клеточного автомата A3 [15]

| ewsnс | c _{new} | ewsnс | c _{new} | ewsnс | c _{new} | ewsnс | c _{new} |
|-------|------------------|-------|------------------|-------|------------------|-------|------------------|
| 00000 | 0 | 01000 | 0 | 10000 | 0 | 11000 | 0 |
| 00001 | 1 | 01001 | 0 | 10001 | 0 | 11001 | 1 |
| 00010 | 1 | 01010 | 0 | 10010 | 0 | 11010 | 0 |
| 00011 | 1 | 01011 | 1 | 10011 | 0 | 11011 | 0 |
| 00100 | 0 | 01100 | 0 | 10100 | 0 | 11100 | 0 |
| 00101 | 0 | 01101 | 0 | 10101 | 1 | 11101 | 1 |
| 00110 | 0 | 01110 | 0 | 10110 | 0 | 11110 | 1 |
| 00111 | 0 | 01111 | 0 | 10111 | 0 | 11111 | 1 |

Упражнение 5.2. Задайте нижеследующие правила клеточного автомата A4 таблично: клетка станет живой или мертвой в зависимости от того, содержит ее окрестность нечетное или четное количество живых клеток. Более формально:

$$\text{center}_{\text{new}} = \text{east} \oplus \text{west} \oplus \text{south} \oplus \text{north} \oplus \text{center}. \quad (5.1)$$

Напишите программу, моделирующую этот клеточный автомат. Проследите его поведение для начальной конфигурации в виде небольшого черного (состоящего из единиц) квадрата в середине белого (заполненного нулями) экрана.

Заметим, что из равенства (5.1) следует *линейность* данного клеточного автомата. Под линейностью будем понимать следующее. Возьмем любые две начальные конфигурации и проследим их эволюцию в двух отдельных экспериментах (например, до ста шагов), а затем сложим (по mod 2) две конечные конфигурации. Результат будет таким же, как если бы две начальные конфигурации были вначале сложены, а затем прошли сто шагов в одном эксперименте.

Другое свойство, которое может быть выведено из формулы (5.1), состоит в том, что для любой исходной фигуры на однородном фоне эта фигура будет обнаружена воспроизведенной в пяти копиях после соответствующего промежутка времени (а позже в двадцати пяти копиях и так далее) [15].

Замечание 5.1. Правила, которые мы описывали до сих пор, являются *детерминированными*, то есть новое состояние клетки однозначно определяется текущим состоянием ее соседей. Отсюда следует, что если запускать детерминированный клеточный автомат несколько раз, начиная с одной и той же конфигурации, то мы неизбежно получим одну и ту же эволюцию. В случае *вероятностного* правила одна и та же текущая ситуация может привести к нескольким различным результатам с заданной вероятностью каждого из них.

Замечание 5.2. Перемещение единиц (живых клеток) в пространстве клеточного автомата можно рассматривать как *движение информации*. *Световой конус события* (обновления клетки) состоит из всех событий в прошлом, которые могут влиять на конечный результат, и всех событий в будущем, которые могут подвергнуться воздействию этого результата. В клеточном автомате *скорость света* (максимальная скорость распространения информации) может быть различной в различных направлениях.

Пример 5.1. Можно ли построить компьютер на клеточном автомате? Это был один из первых вопросов, которые поставил фон Нейман в поисках математической «вселенной», способной поддерживать наиболее существенные черты жизни. Фон Нейман сумел сконструировать универсальный вычисляющий и конструирующий «робот», живущий внутри клеточного автомата. Его конструкция использовала клетки с двадцатью девятью состояниями и пятью соседями и занимала несколько сотен тысяч клеток. Бэнкс предложил более простые решения, используя компромисы между числом состояний, числом соседей, компактностью конструкции и текстурой среды.

Рассмотрим простейшие *правила Бэнкса A5*, которые определяются тремя элементами таблицы 5.2 (с учетом того, что повернутые по часовой стрелке варианты этих элементов приводят к тому же результату).

Для всех других возможных элементов правила не меняют центральную клетку. Заметим, что с учетом расположения

Таблица 5.2. Правила Бэнкса

| | | |
|-----------------------|-----------------------|-----------------------|
| 1 | 0 | 1 |
| 0 1 1 \rightarrow 0 | 1 0 1 \rightarrow 1 | 1 0 1 \rightarrow 1 |
| 0 | 1 | 1 |

единиц в поле нулей правила можно перефразировать так: «заполняйте впадины, выгрызайте углы».

Если мы запустим этот алгоритм, начиная со случайных начальных условий, то получим приятную, но непонятную текстуру, в которой наблюдаются небольшие очаги активности сдвигающимися вперед и назад «*сигналами*» – белыми (или черными) клетками [15]. Можем ли мы «приручить» эту активность и направить на более полезные задачи?

Оказывается, что «сигналы» можно заставить распространяться по «проводам». Вырежем наклонную ступеньку на краю сплошной черной линии. За каждый шаг она будет перемещаться на одно положение в направлении наклона. Для поддержания сигнала черная область должна быть глубиной в три клетки – это и будет наш провод. Достигнув конца провода, сигнал исчезает, чтобы это не происходило, углы на свободном конце провода должны быть защищены небольшим зубцом. Несколько тактов распространения сигнала по проводу представлено на рисунке 5.1.

Оказывается, подправляя правила и конфигурацию, можно заставить сигналы заворачивать за угол, пересекаться и разветвляться. Более того, можно построить *вентиль* с двумя входами и одним выходом, который вычисляет логическую функцию \overline{ab} и т.д.

В заключении главы отметим, что когда мы используем клеточный автомат в качестве модели реальной системы, то наша задача состоит в следующем: подобрать законы функционирования (а также начальную конфигурацию и форму сетки) так, чтобы все, что мы хотим от модели, следовало бы

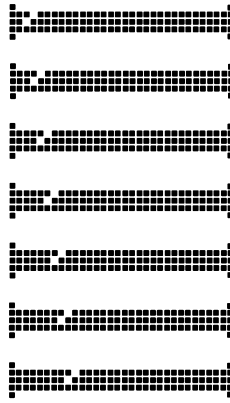


Рис. 5.1. Распространение «сигнала» по «проводу»

из одного этого выбора. Если справедлива гипотеза «Мир в основе своей прост – только его очень много», то клеточные автоматы могут представлять полезный инструмент в попытках понять и описать природу.

Клеточные автоматы успешно используются в моделях диффузии и равновесия, в таких областях как гидродинамика, волновая оптика, при моделировании коллективных явлений (в синергетике) и т.д. [15].

Кроме того, клеточные автоматы обладают двумя фундаментальными достоинствами, которые могут привести к практичным архитектурам компьютеров – это параллелизм и локальность.

Глава 6

Криптосистемы с открытым ключом

В данной главе рассмотрим *криптосистемы с открытым ключом* (или *асимметричные шифры*), в основе которых лежат задачи, принадлежащие теории формальных языков, грамматик и автоматов. Обсуждение этих вопросов будет носить обзорный характер, что объясняет отсутствие строгой терминологии, затрагивающей раздел криптографии. За рамками обзора останутся и оценки сложности предлагаемых криптосистем. Заинтересованный читатель может обратиться, например, к [1, 6, 10, 13, 14, 19].

6.1. Криптосистемы, основанные на грамматиках

Рассмотрим криптосистему с открытым ключом, основанную на *грамматиках*. Для предлагаемого шифра *криптоанализ* и *легальное расшифрование* представляют собой грамматический разбор (распознавание) *криптотекста*. Этот разбор является существенно более легким, если известен *секретный*

ключ.

Будем полагать, что шифруемые тексты являются двоичными последовательностями. Пусть Γ_0 и Γ_1 – произвольные грамматики с одним и тем же терминальным алфавитом T , и пусть A_0 – конечный детерминированный автомат над T (T – множество входных символов). Переведем все заключительные состояния автомата A_0 в незаключительные и наоборот – получим автомат A_1 .

Построим две таких грамматики Γ'_0 и Γ'_1 , что

$$L(\Gamma'_i) = L(\Gamma_i) \cap L(A_i), \quad (6.1)$$

где $L(\Gamma_i)$, $L(A_i)$ – порождаемые языки, $i = 0, 1$. Построение пересечения языков – стандартное [18]. Отметим лишь, что в общем случае языки не замкнуты относительно операции пересечения, то есть тип результирующего языка может быть произвольным.

Пара (Γ'_0, Γ'_1) открывается в качестве *ключа зашифрования*. Используется *зашифрование с помощью «раскрашивания»*¹ – бит i шифруется как произвольное слово в $L(\Gamma'_i)$ ($i = 0, 1$). Автоматы A_0 и A_1 хранятся в качестве *секретного ключа*.

Перехватчик для расшифрования должен определить принадлежность каждого слова криптотекста языку $L(\Gamma'_i)$ ($i = 0, 1$), то есть решить задачу распознавания, являющуюся «трудной» для произвольного контекстного языка, а для языка типа 0 – в общем случае неразрешимой.

¹«Краска связывается с каждой буквой исходного текста. Так как мы полагаем, что исходные тексты являются двоичными последовательностями, то мы используем только две краски – белую (бит 0) и черную (бит 1). Открытый ключ зашифрования дает метод, который порождает произвольно много элементов, раскрашенных белой краской, и произвольно много элементов, раскрашенных черной краской. В обсуждаемых здесь криптосистемах такими элементами являются слова. Биты шифруются как слова, раскрашенные соответствующим образом. Конечно, для различных появлений бита 0 (соответственно 1) должны выбираться различные слова, раскрашенные белой (соответственно черной) краской. В идеальной ситуации задача определения краски для заданного слова является труднорешаемой, в то время как знание *секретной лазейки* позволяет очень легко решать эту задачу [13].»

Легальный получатель решает «легкую» задачу принадлежности слова криптотекста одному из автоматных языков $L(A_0)$ или $L(A_1)$ ².

В силу построения автоматов A_0 и A_1 , языки $L(A_0)$ и $L(A_1)$ не пересекаются (они являются дополнениями друг друга). Тем самым *расшифрование всегда будет однозначным*.

Перейдем к более общему случаю. Отображение $h : \Sigma^* \rightarrow \Delta^*$ такое, что $h(\alpha\beta) = h(\alpha)h(\beta) \forall \alpha, \beta \in \Sigma^*$, называется **морфизмом**. Здесь Σ и Δ – некоторые алфавиты, Σ^* и Δ^* – множества слов над этими алфавитами. Из определения следует, что морфизм полностью определен значениями на буквах алфавита Σ и $h(\varepsilon) = \varepsilon$, где ε – пустое слово в Σ^* .

Пример 6.1. Примером морфизма является следующее отображение $y(a) = a$, $y(b) = bb$, где $\Sigma = \Delta = \{a, b\}$; очевидно, что $y(ab) = abb$.

Пусть Γ – грамматика и h – морфизм, отображающий каждый терминальный символ Γ в терминальный символ или пустое слово ε , и каждый нетерминальный символ Γ – в нетерминальный символ. **Морфический образ** $h(\Gamma)$ грамматики Γ состоит из всех продукций $h(\alpha) \rightarrow h(\beta)$, где $\alpha \rightarrow \beta$ – продукция в Γ . Начальный символ в $h(\Gamma)$ – это $h(S)$, где S – начальный символ в Γ .

Запись вида $\Gamma_1 \subseteq \Gamma_2$ означает, что множество продукций грамматики Γ_1 является подмножеством множества продукций грамматики Γ_2 .

Пусть T_1 – алфавит, намного больший по мощности, чем T . Определим две грамматики Γ_0'' и Γ_1'' с терминальным алфавитом T_1 так, что

$$h(\Gamma_i'') \subseteq \Gamma_i', \quad (6.2)$$

где $i = 0, 1$ и h – описанный ранее морфизм.

В полной версии рассмотренной ранее криптосистемы открытый ключ зашифрования составляет пара (Γ_0'', Γ_1'') . Способ зашифрования остается прежним. К секретному ключу добав-

²Вопрос о «трудных» и «легких» задачах будет обсуждаться в Гл. 7.

ляется морфизм h . Для расшифрования легальный получатель применяет морфизм h к каждому слову криптотекста ξ . Как и в предыдущем случае *расшифрование однозначно*, и если $h(\xi) = \psi$ – результирующее слово, то соответствующий бит исходного текста равен 0 тогда и только тогда, когда автомат A_0 допускает слово ψ (или, что равносильно, $\psi \in L(A_0)$).

Упражнение 6.1.* Оцените трудоемкость описанной криптосистемы. Представляется возможным доказать, что поиск по открытому ключу (Γ_0'', Γ_1'') соответствующих h и A_0 будет \mathcal{NP} -полным³.

6.2. Криптосистемы, основанные на композиции морфизмов

Теперь рассмотрим криптосистему с открытым ключом, в основе которой лежит так называемая *композиция морфизмов*. Как и в предыдущем параграфе, криптоанализ и легальное расшифрование сводятся к грамматическому разбору криптотекста.

Пусть $h_0, h_1 : \Sigma^* \rightarrow \Sigma^*$ – морфизмы и $\omega \in \Sigma^*$, $\omega \neq \varepsilon$.

$$G = (\Sigma, h_0, h_1, \omega) \quad (6.3)$$

называется *обратно детерминированной четверкой*, если из условия на *композицию морфизмов*

$$h_{i_n} \dots h_{i_1}(\omega) = h_{j_m} \dots h_{j_1}(\omega) \quad (6.4)$$

всегда следует, что

$$i_1 \dots i_n = j_1 \dots j_m. \quad (6.5)$$

Все индексы $i_1, \dots, i_n, j_1, \dots, j_m$ принадлежат множеству $\{0, 1\}$. Таким образом, для любой композиции морфизмов выход однозначно определяет порядок их применения. Невозможен случай, когда две различные последовательности морфизмов приводят к одинаковому результату.

³Определение \mathcal{NP} -полноты см. в Гл. 7.

Пример 6.2. Рассмотрим морфизмы, определенные по формулам:

$$h_0(a) = ab, \quad h_0(b) = b, \quad h_1(a) = a, \quad h_1(b) = ba$$

Если мы в качестве ω выберем a , то четверка не будет обратно детерминированной, потому что выход a получается применением любого числа морфизмов h_1 .

С другой стороны, четверка $(\{a, b\}, h_0, h_1, ab)$ уже является обратно детерминированной.

Эта четверка обладает полезным свойством: «последняя буква слова открывает последний применяемый морфизм». Данный принцип позволяет по конечному слову ω' получить исходное ω и найти последовательность применяемых морфизмов.

Рассмотрим *классическую (функциональную) криптосистему*, использующую обратно детерминированную четверку $G = (\Sigma, h_0, h_1, \omega)$. Пусть, как и прежде, шифруемый текст является двоичным. Битовая последовательность $i_1 \dots i_n$ шифруется словом $h_{i_n} \dots h_{i_1}(\omega)$.

Взаимнооднозначность *расшифрования* обеспечивается обратной детерминированностью. Естественно, G должна храниться в секрете. Отметим, что для данной криптосистемы нет различий между криптоанализом и легальным расшифрованием. Чтобы получить исходный текст, необходимо для каждого слова криптотекста провести грамматический разбор – в нашей ситуации это определение применяемой композиции морфизмов.

Пример 6.3. Пусть дана обратно детерминированная четверка $G = (\{a, b\}, h_0, h_1, ab)$,

$$h_0(a) = ab, \quad h_0(b) = b, \quad h_1(a) = a, \quad h_1(b) = ba.$$

Тогда зашифрование исходных текстов можно представить в виде таблицы 6.1.

Таблица 6.1. Схема зашифрования

| Исходный текст | Криптотекст |
|----------------|-------------|
| 0 | abb |
| 1 | aba |
| 00 | abbb |
| 01 | ababa |
| 10 | abbab |
| 11 | abaa |
| 011 | abaabaa |
| и т. д. | |

Следует заметить, что для шифрования текстов, содержащих более двух различных символов, необходимо увеличить число используемых морфизмов.

Преобразуем теперь функциональную криптосистему в криптосистему с открытым ключом. Для этого нам необходимо найти ключ зашифрования, который, в отличие от секретного ключа, приводит к ситуациям с трудным (желательно неоднозначным) грамматическим разбором.

Прежде всего заметим, что областью значений морфизмов может быть и множество подмножеств. В этом случае морфизм называется *конечной подстановкой*.

Пример 6.4. Рассмотрим морфизм σ , определенный следующим образом: $\sigma(a) = \{a, ab\}$, $\sigma(b) = \{b, bb\}$, $\sigma(ab) = \{ab, abb, abbb\}$. Очевидно, что σ является конечной подстановкой.

Пусть по-прежнему $G = (\Sigma, h_0, h_1, \omega)$ – обратнo детерминированная четверка и пусть алфавит Δ такой, что $|\Delta| \gg |\Sigma|$. (Например, $|\Sigma| = 5$, $|\Delta| = 200$.)

Рассмотрим морфизм $g : \Delta^* \rightarrow \Sigma^*$ такой, что $g^{-1}(a)$ непусто для любой буквы $a \in \Sigma$. Буква $d \in \Delta$ является *потомком* буквы $a \in \Sigma$, если $g(d) = a$ (из определения g следует, что каждая буква из Σ имеет по крайней мере одного потомка). Буква

$d \in \Delta$ является *пустышкой*, если $g(d) = \varepsilon$.

Рассмотрим четверку $H = (\Delta, \sigma_0, \sigma_1, \nu)$, где

1) $\sigma_0 : \Delta^* \rightarrow \Delta^*$ – конечная подстановка такая, что $\forall d \in \Delta$ $g(\sigma_0(d)) = h_0(g(d))$, то есть $\sigma_0(d)$ – конечное непустое подмножество множества $g^{-1}(h_0(g(d)))$;

2) конечная подстановка σ_1 определяется аналогично;

3) ν – слово из Δ^* такое, что $g(\nu) = \omega$, то есть ν содержится в $g^{-1}(\omega)$.

В общем случае слово ν и конечные подстановки σ_0, σ_1 не единственны в силу существования пустышек.

Четверка $H = (\Delta, \sigma_0, \sigma_1, \nu)$ открывается в качестве ключа зашифрования. Битовая последовательность (или ее блоки) $i_1 \dots i_n$ шифруется произвольным словом из конечного множества $\sigma_{i_n} \dots \sigma_{i_1}(\nu)$. Четверка $G = (\Sigma, h_0, h_1, \omega)$ и морфизм g образуют секретный ключ. Можно показать, что *расшифрование единственно и если битовая последовательность $i_1 \dots i_n$ зашифровывается как ξ по четверке H , то $i_1 \dots i_n$ расшифровывается как $g(\xi)$ по четверке G [13].*

Пример 6.5. Преобразуем криптосистему из примера 6.3 в криптосистему с открытым ключом. Пусть $\Delta = \{c_1, c_2, c_3, c_4, c_5\}$, морфизм g определим следующим образом:

$$g(c_1) = b, \quad g(c_2) = g(c_4) = a, \quad g(c_3) = g(c_5) = \varepsilon.$$

Выберем конечные подстановки σ_0, σ_1 :

$$\begin{array}{ll} \sigma_0 : & c_1 \rightarrow c_1, c_3 c_1 \\ & c_2 \rightarrow c_4 c_1, c_2 c_1 c_5 \\ & c_3 \rightarrow c_5, c_3 c_3 \\ & c_4 \rightarrow c_4 c_1, c_2 c_5 c_1, c_4 c_1 c_3 \\ & c_5 \rightarrow c_5, c_3 c_5 c_3 \\ \sigma_1 : & c_1 \rightarrow c_1 c_2, c_3 c_1 c_4 \\ & c_2 \rightarrow c_2, c_3 c_5 c_4 \\ & c_3 \rightarrow c_3, c_5 c_5 \\ & c_4 \rightarrow c_2, c_4 c_3 \\ & c_5 \rightarrow c_3, c_5 c_3 \end{array}$$

Чтобы показать, что конечные подстановки σ_0, σ_1 удовлетворяют налагаемым на них условиям, достаточно применить к ним морфизм g и проверить, что они преобразуются в морфизмы h_0, h_1 :

$$\begin{array}{ll}
h_0 : & b \rightarrow b, b \\
& a \rightarrow ab, ab \\
& \varepsilon \rightarrow \varepsilon, \varepsilon \\
& a \rightarrow ab, ab, ab \\
& \varepsilon \rightarrow \varepsilon, \varepsilon \\
h_1 : & b \rightarrow ba, ba \\
& a \rightarrow a, a \\
& \varepsilon \rightarrow \varepsilon, \varepsilon \\
& a \rightarrow a, a \\
& \varepsilon \rightarrow \varepsilon, \varepsilon
\end{array}$$

В качестве ν возьмем слово $c_4c_3c_1$.

Зашифруем текст 011 с помощью открытого ключа H . Из $\sigma_0(\nu)$ выберем, к примеру, слово $c_4c_1c_5c_1 = \xi_1$, затем из $\sigma_1(\xi_1)$ – слово $c_2c_3c_1c_4c_3c_1c_2 = \xi_2$ и, выбирая из $\sigma_1(\xi_2)$ слово ξ_3 , получим криптотекст

$$c_2c_5c_5c_1c_2c_2c_3c_1c_2c_2 = \xi_3.$$

Легальный получатель, зная морфизм g , вычисляет

$$g(\xi_3) = abaabaa,$$

а затем, используя отмеченное выше свойство морфизмов h_0 и h_1 (см. пример 6.2), «легко» вычисляет исходный текст 011.

В заключении отметим, что обратная детерминированность не гарантирует простоту грамматического разбора. Для решения этой проблемы используется сильная обратная детерминированность: четверка $G = (\Sigma, h_0, h_1, \omega)$ – **сильно обратно детерминированная** тогда и только тогда, когда из условия $h_{i_n} \dots h_{i_1}(\omega) = h_t(\xi)$ следует $t = i_n$ и $\xi = h_{i_{n-1}} \dots h_{i_1}(\omega)$. Можно показать, что *расшифрование (грамматический разбор) слова в сильно обратно детерминированной четверке можно провести справа налево без предварительного просмотра* [13].

Упражнение 6.2.* Оцените трудоемкость описанной криптосистемы.

6.3. Криптосистемы, основанные на последовательных автоматах

Следующая криптосистема с открытым ключом является приложением теории конечных автоматов и основана на *автоматах с памятью*⁴. Рассмотрим последовательный автомат (автомат с памятью) M . Очевидно, что он переводит любое входное слово ξ в выходное слово ψ той же длины. *Инверсия* M^{-1} переводит выходное слово ψ обратно во входное ξ .

Пусть k – натуральное число. Определим *инверсию с задержкой* $k - M^{-1}(k)$ следующим образом. Пусть входному слову ξ автомата M соответствует выходное слово ψ . После получения на входе слова $\psi\mu$, где μ – произвольное слово длины k , автомат $M^{-1}(k)$ порождает на выходе слово $\nu\xi$, где ν – также слово длины k . В качестве ключа зашифрования открываются k и $M^{-1}(k)$, а M хранится как секретный ключ расшифрования.

Зашифрование исходного текста ψ происходит с помощью выбора произвольного слова μ длины k и применения автомата $M^{-1}(k)$ к слову $\psi\mu$. При расшифровании криптотекста легальный получатель игнорирует первые k букв и применяет автомат M к оставшемуся слову.

Упражнение 6.3.* Существуют ли алгоритмы нахождения инверсии и инверсии с задержкой для произвольного последовательного автомата?

Упражнение 6.4.* Оцените трудоемкость описанной криптосистемы.

Упражнение 6.5.* Перспективной основой для криптосистем с открытым ключом являются *клеточные автоматы*⁵. Этот подход еще как следует не изучен. К примеру, по заданному обратимому клеточному автомату очень трудно найти его инверсию. Открытый ключ зашифрования составля-

⁴Определение автомата с памятью (последовательного автомата) см. в [3].

⁵Определение клеточного автомата см. в Гл. 5.

ют обратимый клеточный автомат C и натуральное число k . Исходный текст кодируется как конфигурация клеточного автомата C и шифруется применением этого автомата k раз к данной конфигурации. Результирующая конфигурация образует криптотекст. Инверсия C^{-1} образует секретный ключ. Для расшифрования легальный получатель k раз применяет к криптотексту клеточный автомат C^{-1} [13].

Глава 7

Элементы теории сложности

В главах 3 и 4 мы уже обсуждали такие понятия, как массовая задача, алгоритм, разрешимость и т. п. Здесь мы еще раз вернемся к этим терминам, остановимся на вопросах оценки качества алгоритмов и поговорим о классификации алгоритмов и задач¹.

7.1. Временная сложность алгоритмов

Массовой задачей (или просто задачей либо проблемой) будем называть некоторый общий вопрос, на который следует дать ответ. Как правило, задача содержит несколько параметров или переменных. Если всем параметрам присвоить конкретные значения, то из массовой задачи M получим *индивидуальную задачу* I .

¹Мы опускаем ряд классических определений и постановок задач, относящихся к теории графов и комбинаторике. Заинтересованный читатель может обратиться, например, к [5].

Под **алгоритмом**, как и прежде (см. § 3.1), будем понимать общую, выполняемую шаг за шагом процедуру решения задачи. Алгоритм **решает** массовую задачу M , если он применим к любой индивидуальной задаче I из M и обязательно дает решение задачи I .

«Наиболее эффективным» алгоритмом будем считать самый быстрый алгоритм, так как чаще всего именно ограничение по времени является доминирующим фактором, определяющим пригодность конкретного алгоритма для практики.

Время работы алгоритма удобно выражать в виде функции, зависящей от одной переменной. Этой переменной является «размер» (или **входная длина**) индивидуальной задачи, то есть объем входных данных, требуемых для описания этой задачи (например, число символов в слове, которое кодирует индивидуальную задачу). Сама функция называется **временной сложностью** алгоритма. Она каждой входной длине n ставит в соответствие максимальное (по всем индивидуальным задачам длины n) время, затрачиваемое алгоритмом на решение индивидуальных задач этой длины. Исходя из этого определения, получаем, что временная сложность определена нами как мера поведения алгоритма в *наихудшем случае*.

Более формально временная сложность алгоритма определяется следующим образом. Пусть q – выбранная схема кодирования входных данных, $t(I)$ – время, затрачиваемое алгоритмом на решение индивидуальной задачи I . Тогда временная сложность алгоритма решения массовой задачи M вычисляется по формуле:

$$t(n) = \max_{I: |q(I)|=n} t(I). \quad (7.1)$$

Будем говорить, что функция $f(n)$ есть $O(g(n))$, если существует константа C такая, что $|f(n)| \leq C|g(n)|$ для любого $n \geq 0$. **Полиномиальный** алгоритм – это алгоритм, у которого временная сложность равна $O(p(n))$, где $p(n)$ – некоторая полиномиальная функция, а n – входная длина. Алгоритмы, временная сложность которых не поддается подобной оценке, называются **экспоненциальными** (следует отметить, что

это определение включает и такие функции, как $n^{\log n}$, хотя они не являются ни полиномиальными, ни экспоненциальными). В силу того, что при росте размера задачи полиномиальные алгоритмы существенно *быстрее* (требуют меньше временных затрат), чем экспоненциальные, они являются более предпочтительными.

Заметим, что большинство экспоненциальных алгоритмов – это просто варианты *полного перебора*, тогда как полиномиальные алгоритмы обычно можно построить лишь тогда, когда удастся более детально изучить особенности решаемой проблемы.

Задача считается *хорошо решаемой* (или «*легкой*»), если для нее найден полиномиальный алгоритм. Задача называется *труднорешаемой* (или «*трудной*»), если для ее решения не существует полиномиального алгоритма [5]. Задача является *неразрешимой*, если вообще не существует алгоритма ее решения (см. §§ 3.5, 4.4).

7.2. Классы \mathcal{P} и \mathcal{NP}

По степени сложности алгоритмов задачи можно разбить на классы.

Определим первый класс задач – *класс \mathcal{P}* (*Polinomial*). Он состоит из задач, для решения которых существуют полиномиальные алгоритмы. Это так называемые «легкие» задачи. В определении класса \mathcal{P} существенным является фиксация схемы кодирования q (см. формулу 7.1), так как в общем случае существование полиномиального алгоритма зависит от способа кодирования входных данных задачи [5].

Следуя основной гипотезе теории алгоритмов (см. § 3.3), данное определение можно перефразировать следующим образом: задача принадлежит классу \mathcal{P} тогда и только тогда, когда существует машина Тьюринга, решающая ее за полиномиальное время². При этом под временем работы машины

²Это определение можно сформулировать, используя любую другую алгоритмическую модель.

Тьюринга понимается число шагов, которое машина должна совершить для решения задачи.

Пример 7.1. Следующие задачи принадлежат классу \mathcal{P} : умножения матрицы на вектор, поиск минимального *остовного* дерева (*алгоритмы Прима и Краскала*), поиск кратчайшего пути в орграфе (*алгоритм Дейкстры*) и т. д.

Упражнение 7.1. Покажите, что *метод Гаусса* не является полиномиальным.

Обсудим теперь задачи «трудные» и вопросы сравнения их сложности. В дальнейшем ограничимся рассмотрением **задач распознавания свойств**: такие задачи имеют только два возможных решения – «да» или «нет»³.

Пример 7.2. Чтобы из оптимизационной задачи получить задачу распознавания свойств, достаточно перефразировать ее следующим образом: существует ли допустимое решение, стоимость которого меньше (больше) наперед заданного числа C ?

Оказывается, что каждой задаче распознавания свойств можно сопоставить некоторый язык [5]. Напомним, что если A – алфавит, A^* – множество всех слов, образованных этим алфавитом, то любое подмножество множества A^* называется *языком*.

Соответствие между задачей распознавания свойств и языком устанавливается с помощью стандартных схем кодирования q , которые обычно используются для представления индивидуальной задачи при ее решении [5]. Слова, являющиеся кодами индивидуальных задач из M с положительным ответом на вопрос, составляют требуемый язык. Таким образом, задача распознавания свойств может быть заменена задачей определения принадлежности некоторого слова α заданному языку L (это задача разрешимости – см. § 1.8).

³Именно задачи распознавания свойств соединяют пройденные нами ранее понятия: языки, грамматики, машины Тьюринга и т. д.

Теперь выделим класс задач распознавания свойств, для которых время определения принадлежности слова соответствующему языку будет полиномиальным. То есть, если у нас есть решение α , то существует машина Тьюринга, которая допускает (распознает) слово α за полиномиальное время. Такой класс задач называется **классом \mathcal{NP}** (*Nondeterministically polynomial*)⁴.

Одно из основных понятий, которое нам понадобится в дальнейшем, это *полиномиальная сводимость* языков (или задач). Пусть $L_1, L_2 \in A^*$ – два языка. Говорят, что язык L_1 **сводится** к языку L_2 , если существует функция f , отображающая A^* в A^* , такая, что слово α принадлежит языку L_1 тогда и только тогда, когда $f(\alpha)$ принадлежит языку L_2 .

Пример 7.3. Пусть A^* – множество всех слов конечной длины в двоичном алфавите, L_1 – множество слов из A^* с четным числом единиц, L_2 – с нечетным. Рассмотрим следующую функцию: $f(a_1 \dots a_k) = a_1 \dots a_k 1$. Эта функция осуществляет сводимость языка L_1 к L_2 .

Сводимость называется *полиномиальной*, если для функции f существует вычисляющая полиномиальная (в смысле временной сложности) программа.

Полиномиальная эквивалентность двух языков L_1 и L_2 определяется как двусторонняя полиномиальная сводимость.

После введения понятия сводимости очевидными стали следующие вопросы. Какие языки (задачи) в \mathcal{NP} полиномиально эквивалентны? Какова иерархия сложности задач в \mathcal{NP} ?

Основным результатом, после которого введение всех этих понятий приобрело практический смысл, была теорема С. Кука:

⁴Название определяется понятием *недетерминированного полиномиального алгоритма*, который состоит из стадии угадывания (поиск слова α) и стадии проверки (решение задачи разрешимости). Вычислительная сложность стадии угадывания не оговаривается, а сложность стадии проверки – полиномиальная.

Теорема 7.1 (Кук). *Любая задача из класса \mathcal{NP} может быть полиномиально сведена к некоторой, вполне конкретной задаче из этого класса, так называемой задаче «Выполнимость».*

Сформулируем проблему «**Выполнимость**» [9]. Рассмотрим булеву функцию $f(x_1, \dots, x_n)$ в конъюнктивной форме записи:

$$f(x_1, \dots, x_n) = (x_1^{\sigma_{11}} \vee \dots \vee x_n^{\sigma_{1n}}) \dots (x_1^{\sigma_{m1}} \vee \dots \vee x_n^{\sigma_{mn}}) \quad (7.2)$$

где x_i^σ равно x_i при $\sigma = 1$ и равно \bar{x}_i при $\sigma = 0$. Необходимо определить, выполнима ли формула, то есть существует ли набор переменных, при которых функция принимает значение «истина»?

Пример 7.4. Для функции $f(X) = x_1(x_1 \vee x_3)(x_2 \vee x_6 \vee x_3)$ существует много решающих наборов.

Для функции $f(X) = x_1(\bar{x}_1 \vee x_2)\bar{x}_2$ решающего набора нет.

Обратим внимание на то, что в постановке задачи присутствует вопрос «существует ли?». Для каждой единичной задачи в классической постановке это означает, что мы должны указать решающий набор. Но задачу можно понимать и по «инженерному» – пройдет ли сигнал через схему, определяемую булевой функцией $f(X)$? Или даже более конкретно – какова вероятность прохождения сигнала через схему, то есть, сколько всего решающих наборов существует для данной $f(X)$?

Итак, следуя теореме Кука, задача «Выполнимость» считается самой универсальной в классе \mathcal{NP} , так как любая задача из этого класса полиномиально сводится к данной.

Теперь определим третий важный класс задач: любая задача, полиномиально эквивалентная задаче «Выполнимость», называется **\mathcal{NP} -полной** (\mathcal{NP} -complete).

Очевидно, эквивалентность надо доказывать, то есть доказывать полиномиальную сводимость задачи «Выполнимость» к рассматриваемой задаче из класса \mathcal{NP} . Но сначала надо доказать, что сама задача входит в \mathcal{NP} !

Несложно показать, что задача M является \mathcal{NP} -полной, если $M \in \mathcal{NP}$ и любая задача из класса \mathcal{NP} полиномиально сводится к M .

Теперь обсудим вопрос о взаимоотношении классов \mathcal{P} и \mathcal{NP} . Достаточно очевидным является следующее соотношение [5]:

$$\mathcal{P} \subseteq \mathcal{NP}. \quad (7.3)$$

В силу того, что для многих задач из класса \mathcal{NP} не найдено полиномиального алгоритма решения, имеет место гипотеза

$$\mathcal{P} \neq \mathcal{NP}. \quad (7.4)$$

Более того, на сегодняшний день известно более тысячи \mathcal{NP} -полных задач, но ни для одной из них не удалось придумать полиномиального алгоритма решения. Поэтому имеются очень сильные основания считать данную гипотезу справедливой.

Значение этой, пока еще нерешенной проблемы, трудно переоценить. Вдруг окажется, что гипотеза неверна? Тогда, если будет найден хотя бы один полиномиальный алгоритм для какой-нибудь \mathcal{NP} -полной задачи, то *все* задачи из этого класса можно будет решать этим алгоритмом! Данный (гипотетический) математический результат приведет к новой промышленной революции.

Например, что произойдет с криптографией, если вдруг окажется, что $\mathcal{P} = \mathcal{NP}$? Произойдет нечто из ряда вон: криптография, и теоретическая и практическая, по существу прекратят свое существование. «Выживут» шифры типа Вернама с одноразовыми ключами, схемы аутентификации типа Симмонса, также с одноразовыми ключами, схемы разделения секрета. И все.

7.3. \mathcal{NP} -полные задачи

Рассмотрим несколько \mathcal{NP} -полных задач и математических конструкций, которые связывают эти задачи с задачей «Выполнимость» [9].

Сведем полиномиально задачу «Выполнимость» к «*Задаче о разрешимости в $(0,1)$ -числах системы линейных уравнений*». Этот процесс рассмотрим на примере. Пусть

$$f(x_1, x_2, x_3, x_4) = x_1(\bar{x}_1 \vee x_2 \vee x_3)(x_2 \vee \bar{x}_3)(x_2 \vee \bar{x}_4)x_4$$

Условие выполнимости равносильно разрешимости следующей системы булевых уравнений:

$$\begin{cases} x_1 = 1, \\ \bar{x}_1 \vee x_2 \vee x_3 = 1, \\ x_2 \vee \bar{x}_3 = 1, \\ x_2 \vee \bar{x}_4 = 1, \\ x_4 = 1. \end{cases}$$

Заменим дизъюнкцию на знак плюс и \bar{x} — на $1 - x$. Перейдем к обычной системе уравнений и неравенств, где нас интересуют решения в числах 0 и 1:

$$\begin{cases} x_1 = 1, \\ -x_1 + x_2 + x_3 \geq 0, \\ x_2 - x_3 \geq 0, \\ x_2 - x_4 \geq 0, \\ x_4 = 1 \end{cases}$$

или, что то же самое:

$$\begin{cases} x_2 + x_3 \geq 1, \\ x_2 - x_3 \geq 0, \\ x_2 \geq 1. \end{cases}$$

Стандартным образом добавим новые переменные и от системы неравенств перейдем к системе уравнений:

$$\begin{cases} x_2 + x_3 - y_1 = 1, \\ x_2 - x_3 - y_2 = 0, \\ x_2 - y_3 = 1. \end{cases}$$

Отсюда получаем, что решающий набор будет равен:

$$x_1 = x_2 = x_3 = x_4 = 1.$$

То, что представленный способ сведения является полиномиальным – достаточно очевидно. Таким образом, «Задача о разрешимости в $(0,1)$ -числах системы линейных уравнений» является \mathcal{NP} -полной.

Рассмотрим теперь «*Задачу о клике*». *Клик*ой графа G называется максимальное по мощности множество вершин этого графа, любые две из которых являются смежными. «Задача о клике» формулируется следующим образом: существует ли в графе клика из m вершин.

Пример 7.5. Для $m = 3$ – это треугольник, $m = 4$ – четырехугольник с диагоналями. (Отметим, что если в графе есть клика с числом вершин 5, то граф не является планарным.)

Напомним, что доказательство \mathcal{NP} -полноты «Задачи о клике» состоит в полиномиальном сведении к ней задачи «Выполнимость». Рассмотрим пример такого сведения для случая трех переменных. Пусть дана функция

$$f(X) = x_1(x_1 \vee x_2 \vee \bar{x}_3)(\bar{x}_2 \vee x_3)$$

Поставим ей в соответствие граф (см. рис. 7.1).

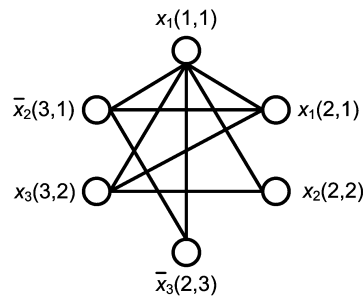


Рис. 7.1. Граф, соответствующий функции $f(X)$

Каждой переменной будут отвечать вершины, маркированные специальным образом: переменной x_r , входящей в скобку

i и занимающей там место j , отвечает вершина графа, маркированная как (i, j) .

Таким образом, первая переменная x_1 – это вершина $(1, 1)$, переменная x_1 второго множителя – вершина $(2, 1)$, переменная $x_2 - (2, 2)$, $x_3 - (3, 2)$, $\bar{x}_2 - (3, 1)$, $\bar{x}_3 - (2, 3)$.

Две вершины $((i, j), (k, l))$ будем считать смежными, если

1) i не равно k , то есть соответствующие переменные находятся в разных скобках.

2) x_{ij} не равно \bar{x}_{kl} , где под x_{ij} понимается переменная, соответствующая паре (i, j) .

Например, вершины $(3, 1)$ и $(2, 2)$ не смежные, так как им отвечают переменные \bar{x}_2 и x_2 .

Теперь задачу «3-Выполнимость» можно переформулировать следующим образом: существуют ли в графе G клика мощности три?

Например, клике $\{(1, 1), (2, 1), (3, 2)\}$, отвечает набор переменных (x_1, x_1, x_3) . Тогда в качестве «выполняющего» набора для функции f можно выбрать следующие значения переменных: $x_1 = 1$, x_2 – произвольно, $x_3 = 1$. Другая клика и другой набор переменных: $\{(1, 1), (2, 2), (3, 2)\}$ и (x_1, x_2, x_3) .

Далее приведем примеры \mathcal{NP} -полных задач (без доказательства полиномиальной сводимости к ним задачи «Выполнимость»):

1. Является ли данный неориентированный граф k -раскрашиваемым?

2. Имеет ли данный неориентированный граф гамильтонов цикл или гамильтонов путь?

3. Существует ли решение в $(0, 1)$ -числах уравнения $\sum_{i=1}^n a_i x_i = b$, где a_i и b заданы?

4. Изоморфен ли данный неориентированный граф G некоторому подграфу данного неориентированного графа G' ?

5. «**Задача коммивояжера**»: даны натуральное число n – количество городов, симметричная матрица d_{ij} ($n \times n$) с неотрицательными целочисленными элементами – матрица расстояний между городами, и целое неотрицательное число L . Требуется определить, существует ли такая циклическая

перестановка длины n (обход), что сумма соответствующих расстояний не больше L .

На данный момент известно около тысячи \mathcal{NP} -полных задач, доказательство \mathcal{NP} -полноты в некоторых случаях представляет большую трудность.

Для некоторых важных задач вопрос об \mathcal{NP} -полноте открыт до сих пор. Это задача проверки изоморфизма графов, задача дискретного логарифмирования, задача факторизации и др.

Поговорим теперь о дальнейшей классификации задач. Ограничиваются классы сложности классами \mathcal{P} и \mathcal{NP} ? По всей видимости нет. Рассмотрим, например, «дополнение» к задаче о существовании гамильтонова цикла в графе: является ли данный граф негамильтоновым? Совсем не очевидно, что эта задача принадлежит классу \mathcal{NP} . Единственный известный на сегодняшний день общий метод доказательства того, что некоторый граф не является гамильтоновым, состоит в том, чтобы систематически выписать все циклы и проверить, что ни один из них не содержит все вершины графа.

Упражнение 7.2. Сформулируйте дополнение к задаче коммивояжера.

В общем случае *дополнением \overline{M} задачи распознавания свойств M* называется задача, в которой коды индивидуальных задач с ответом «да» соответствуют кодам индивидуальных задач из M , не имеющих ответа «да». Класс \mathcal{P} замкнут относительно операции дополнения:

Теорема 7.2. *Если задача M принадлежит классу \mathcal{P} то ее дополнение также принадлежит классу \mathcal{P} .*

Для класса \mathcal{NP} замкнутость относительно операции дополнения не доказана. Если класс задач, являющихся дополнениями к задачам класса \mathcal{NP} , обозначить *со- \mathcal{NP}* , то имеет место следующее утверждение:

Теорема 7.3. *Если дополнение некоторой \mathcal{NP} -полной задачи принадлежит классу \mathcal{NP} , то классы \mathcal{NP} и со- \mathcal{NP} равны.*

Иногда удается доказать, что все задачи из класса \mathcal{NP} полиномиально сводятся к некоторой задаче M , но не удается доказать, что $M \in \mathcal{NP}$. Тогда M не может называться \mathcal{NP} -полной. Однако M несомненно настолько же трудна, как любая задача из класса \mathcal{NP} , и настолько же труднорешаема. Такие задачи называются *\mathcal{NP} -трудными*.

Пример 7.6. Примером \mathcal{NP} -трудной задачи является «Задача поиска K -го по порядку подмножества» [5]: даны целые неотрицательные числа c_1, \dots, c_n , K , L ($K \leq 2^n$ и $L \leq \sum_i c_i$). Спрашивается, существует ли K различных подмножеств $S_1, \dots, S_K \subseteq \{1, 2, \dots, n\}$ таких, что $\sum_{j \in S_i} c_j \geq L$, для $i = 1, 2, \dots, K$?

В заключении определим еще один класс задач. Алгоритм называется *псевдополиномиальным*, если его временная сложность ограничена сверху полиномом от двух переменных: n и C . Здесь n – по-прежнему входная длина задачи, а C – максимальное число в задаче. Очевидно, что любой полиномиальный алгоритм является псевдополиномиальным.

Пример 7.7. Рассмотрим задачу разрешимости в целых числах уравнения $\sum_{i=1}^n c_i x_i = K$. Она \mathcal{NP} -полная. Но каждую индивидуальную задачу можно решить за $O(nK)$ операций. Противоречие? Нет, так как входом в эту индивидуальную задачу является строка бит длиной $n \log_2 K$. То есть, если вход имеет длину nl , то сложность равна $O(n2^l)$.

Псевдополиномиальные алгоритмы обычно довольно успешны. Например, шифросистема с открытым ключом, построенная на основе задачи примера 7.7, была успешно взломана. К сожалению, не для всех задач удастся построить псевдополиномиальные алгоритмы, более того, верно следующее утверждение [5]:

Теорема 7.4. *Если $\mathcal{P} \neq \mathcal{NP}$, то существуют задачи, для которых нельзя построить псевдополиномиальные алгоритмы.*

Такие задачи называют **сильно \mathcal{NP} -полными**.

Подводя итог, еще раз остановимся на классификации алгоритмов. Так по временной сложности алгоритмы подразделяются на:

- 1) полиномиальные;
- 2) недетерминированные полиномиальные;
- 3) экспоненциальные (например, перебор – для малого n вполне эффективен).

Если же классифицировать алгоритмы по точности найденного решения, то можно выделить

- 1) точные алгоритмы;
- 2) вероятностные алгоритмы (бывает, что в асимптотике решение стремится к точному);
- 3) приближенные алгоритмы (здесь мы имеем гарантию того, что наше решение в k раз «не хуже», чем точное);
- 4) *эвристики* (нет строгих математических обоснований, но есть практическая эффективность; об одной из эвристик мы поговорим в следующей главе).

При классификации задач, разумным критерием выступает временная сложность алгоритмов решения:

- 1) «легкие» задачи (существует полиномиальный алгоритм – класс \mathcal{P});
- 2) «трудные» задачи (либо существует недетерминированный полиномиальный алгоритм – класс \mathcal{NP} , либо найден только экспоненциальный алгоритм);
- 3) неразрешимые задачи (доказано, что не существует алгоритма решения).

Глава 8

Эволюционные алгоритмы

Рассматриваемые в этой главе алгоритмы относятся к классу *эвристических*. Поэтому, прежде всего, дадим определение данного термина. **Эвристика (эвристический алгоритм)** – это алгоритм решения поставленной задачи, который, как правило, показывает хорошие результаты на практике, но плохо поддается теоретическому исследованию и обоснованию.

В частности, эвристические методы поиска экстремума характеризуются отсутствием каких-либо существенных ограничений на *целевую функцию* (дифференцируемость, выпуклость, одноэкстремальность и т.д.). Вместе с тем, не всегда удается получить строгие математические результаты относительно экстремальности найденного решения.

Тем не менее, в настоящее время подобные алгоритмы интенсивно развиваются и успешно используются для решения различных прикладных задач.

8.1. Терминология

Относительно новыми, но широко используемыми на практике, являются **эволюционные алгоритмы** (ЭА) – эвристические адаптивные методы поиска, основывающиеся на биологических принципах естественного отбора и выживания сильнейшего [2]. Как известно, биологические популяции развиваются в течение нескольких поколений, подчиняясь законам естественного отбора и принципу «выживает наиболее приспособленный», открытому Чарльзом Дарвином. Эволюционные алгоритмы используют прямую аналогию с таким механизмом, иными словами, они способны «развивать» решения реальных задач, если те соответствующим образом представлены в терминах ЭА.

Эволюционные алгоритмы включают в себя

- 1) эволюционное программирование (*Evolutionary Programming*),
- 2) генетические алгоритмы (*Genetic Algorithms*),
- 3) эволюционные стратегии (*Evolutionary Strategies*).

Эволюционные стратегии работают с переменными, которые в общем случае принадлежат непрерывному пространству поиска [20, 21]. Генетические алгоритмы ориентированы на задачи дискретной оптимизации [23], то есть задачи с дискретным пространством, на котором задана целевая функция.

Эволюционные алгоритмы используются при решении задач поиска максимума (для задач минимизации целевую функцию следует инвертировать $F(x) = -E(x)$ и затем сменить в область положительных значений). Нередко целевая функция

$$F : G \rightarrow \mathbb{R}, \quad (8.1)$$

$G \subseteq \mathbb{R}^k$, исходной задачи заменяется на так называемую **функцию пригодности** или приспособленности (*fitness function*)

$$f(x) = (g \circ F)(x), \quad (8.2)$$

где $g : \mathbb{R} \rightarrow \mathbb{R}^+$ – строго монотонно возрастает. В биологической интерпретации функция f характеризует степень при-

способности конкретного *индивида* к окружающей среде.

Если определена область поиска $G \neq \mathbb{R}^k$ и $x \notin G$, то функция f может действовать аналогично функции штрафа в методах внешних или внутренних штрафных функций [12]. В простейшем случае в качестве функции пригодности выступает сама целевая функция, если она удовлетворяет требуемым условиям неотрицательности. Возможный вид функции f :

$$f(x) = \frac{F(x) - F_{\min}}{F_{av} - F_{\min}}, \quad (8.3)$$

где F_{\min} , F_{av} – минимальное и среднее значения функции F на популяции.

Популяцией $P = \{x^1, \dots, x^N\}$ численности N считается вектор пространства $\mathbb{R}^{k \times N}$. Отметим, что в терминологии ЭА вектор x , принадлежащий популяции, называют также *особью* или *хромосомой*.

На начальной стадии ЭА иницируется исходная популяция особей. Она формируется случайным образом, хотя применяются также и самонаводящиеся способы (если их можно определить заранее). Размер популяции, как правило, пропорционален количеству оптимизируемых параметров. Слишком малая популяция приводит к замыканию в неглубоких локальных экстремумах. Слишком большое количество особей чрезмерно удлиняет вычислительную процедуру и также может не привести к точке глобального экстремума. При случайном выборе векторов x , составляющих исходную популяцию, они статистически независимы и представляют собой начальное погружение в пространство параметров. Одна часть этих особей лучше «приспособлена к существованию» (у них большие значения функции пригодности), а другая часть – хуже. Упорядочение особей в популяции производится в направлении от лучших к худшим.

Шагом ЭА является переход от текущей популяции P_t к новой P_{t+1} под действием **основных операторов** ЭА – *селекции, мутации и рекомбинации*. Два последних (мутацию и рекомбинацию) следует оценивать в соответствии с вероятностью, с которой они находят лучшее решение.

Селекция, рекомбинация и мутация выполняются для подбора таких векторов x , на которых максимизируется величина функции пригодности.

8.2. Основные операторы эволюционных алгоритмов

Оператор *селекции*

$$S: \mathbb{R}^{k \times N} \rightarrow \{1, \dots, N\} \quad (8.4)$$

является аналогом естественного отбора в природе. Он действует на пространстве популяций, выбирая номера «родителей» для порождения новых «потомков». Главное свойство селекции, присущее любому варианту этого оператора, – особи с большим значением функции пригодности получают в среднем большее число потомков в следующем поколении.

Существует огромное количество методов селекции, от полного случайного выбора (как правило, среди наиболее приспособленных особей), через взвешенно-случайный подход и вплоть до так называемой турнирной системы. В последнем случае случайным образом отбирается несколько особей, среди которых определяются наиболее приспособленные. Из победителей последовательно проведенных турниров формируется набор родителей.

Во взвешенно-случайной системе в процессе отбора учитывается информация о текущем значении функции пригодности. Наиболее распространена *«рулеточная» селекция*, при которой каждая особь x^i текущей популяции P_t оказывается родителем при формировании новой особи в популяции P_{t+1} с вероятностью

$$p(x^i) = \frac{f(x^i)}{\sum_{j=1}^N f(x^j)}. \quad (8.5)$$

Тем самым каждая особь принимает участие в построении следующего поколения в соответствии со схемой Бернулли,

где число испытаний равно N [17]. Графическая интерпретация этого вида селекции представлена на рисунке 8.1. Полная

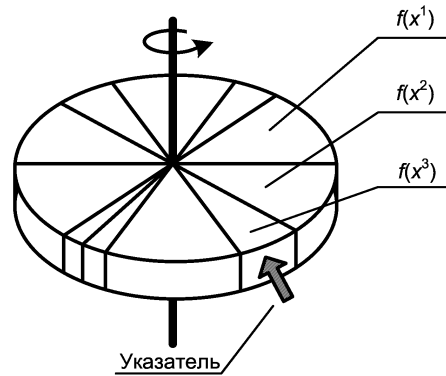


Рис. 8.1. Схема «рулетки», используемая при выборе родителей для будущего поколения

площадь круга соответствует сумме значений функций пригодности всех особей данной популяции. Каждый выделенный сегмент отвечает конкретной особи. Площадь отдельных сегментов пропорциональна значениям соответствующей функции пригодности: наиболее приспособленным особям отводятся большие сегменты колеса рулетки, увеличивающие их шансы попадания в категорию родителей.

Другие варианты оператора селекции могут быть найдены, например, в работе [23].

Особь, отобранная на этапе селекции в качестве родителей, подвергается рекомбинации.

Некоторые варианты ЭА позволяют формировать так называемую *эли́ту* – одна или несколько особей, наилучших с точки зрения функции пригодности. Элита переносится в следующее поколение без изменений. Элитизм может быть внедрен практически в любой стандартный метод селекции.

Рекомбинация (скрещивание, кроссовер (crossover)) – это некоторый поиск, который в принципе ограничен заданной

популяцией. Чаще всего процесс рекомбинации представляет собой расщепление пары особей на две части с последующим обменом этих частей в особях родителей¹. Такой вид кроссовера называется *одноточечной* рекомбинацией $C(x, y) = (x', y')$:

$$(x_1, \dots, x_k), (y_1, \dots, y_k) \rightarrow (x_1, \dots, x_j, y_{j+1}, \dots, y_k), (y_1, \dots, y_j, x_{j+1}, \dots, x_k). \quad (8.6)$$

Место расщепления j выбирается случайным образом равномерно от 1 до n .

Существует множество других вариантов работы оператора рекомбинации [23]. В частности:

Однородная (равномерная) рекомбинация $C(x, y) = z$:

$$x = (x_1, \dots, x_k), y = (y_1, \dots, y_k) \rightarrow z = (z_1, \dots, z_k), \quad (8.7)$$

где $z_i = x_i$ либо $z_i = y_i$. Вероятность выбора x_i или y_i равна 0,5.

Непрерывная рекомбинация $C(x, y) = z$ определяется согласно формуле (8.7), но $z_i = (x_i + y_i)/2$.

Во всех случаях задается вероятность срабатывания оператора рекомбинации, значение этой вероятности зависит либо от размера популяции N , либо от длины вектора k , либо находится экспериментально [23].

Как отмечалось ранее, признается допустимым перенос в очередное поколение некоторых случайно выбранных особей вообще без рекомбинации (элита). Это соответствует ситуации, когда рекомбинация достигает успеха с вероятностью на уровне 0,6 - 1.

Важно помнить, что суммарное количество потомков, полученных в результате рекомбинации, равно количеству отбракованных особей в результате селекции. После добавления новых особей к оставшимся размер популяции остается неизменным.

¹Также может применяться разделение родителей на несколько одинаковых частей с последующим обменом комплементарными компонентами.

Мутация базируется на случайности, шаг мутации почти непредсказуем. В генетических алгоритмах оператор мутации $M_{GA}(x) = x'$ в каждой позиции аргумента с заданной вероятностью заменяет ее содержимое на элемент используемого алфавита A (мутация вектора x происходит по схеме Бернулли). В эволюционных стратегиях мутация осуществляется путем добавления к каждой координате вектора нормально распределенной случайной величины, математическое ожидание которой равно нулю [20].

Мутация обеспечивает защиту как от слишком раннего завершения алгоритма (в случае выравнивания значений функции пригодности на всех векторах популяции), так и от получения одного и того же значения конкретной координаты у всех особей. Однако необходимо иметь в виду, что случайные мутации приводят к повреждению уже частично приспособленных векторов. Обычно мутации подвергается не более 1 – 5 % координат всей популяции векторов.

Элемент, подвергаемый мутации, отбирается случайным образом в соответствии с равномерным распределением [2, 23].

Итак, рекомбинация наиболее приспособленных особей приводит к тому, что исследуются наиболее перспективные участки пространства поиска. В результате новая популяция P_{t+1} содержит как старые особи, перешедшие из предыдущего поколения, так и новые – полученные под воздействием мутации и рекомбинации. Исследованиями доказано, что каждое последующее поколение, сформированное после выполнения селекции, рекомбинации и мутации, имеет статистически лучшие средние показатели пригодности.

Еще раз отметим, что вероятности мутации и рекомбинации являются настраиваемыми параметрами и позволяют регулировать работу ЭА.

В качестве *критерия остановки* используется ограничение по числу итераций, либо условие достижения оптимума (в случае существования алгоритма, позволяющего доказать оптимальность найденного решения, которым является наиболее приспособленная особь текущей популяции).

При реализации эволюционного процесса отслеживается, как правило, не только максимальное значение функции пригодности, но и среднее значение по всей популяции, а также их вариации. Решение об остановке алгоритма может приниматься и в случае отсутствия прогресса оптимизации, определяемого по изменениям названных характеристик.

Замечание 8.1. Возникает вопрос: как совмещать операторы селекции, мутации и рекомбинации для эффективной работы алгоритма? Большинство результатов, полученных в этом направлении, являются эмпирическими и основываются на численных экспериментах и компьютерном моделировании [22, 23]

Замечание 8.2. При решении прикладных задач нередко используются *гибридные алгоритмы*, в которых на первом этапе на протяжении некоторого числа итераций работает один из вариантов эволюционного алгоритма. Затем, на втором этапе, лучшие с точки зрения функции пригодности (элитные) особи выступают в роли начального приближения какого-либо традиционного метода решения задачи поиска экстремума. Данный процесс может повторяться, при этом возможно изменение численности популяции. Если необходимо, каждая новая особь перед добавлением в популяцию подвергается некоторой корректировке, делающей решение допустимым.

В заключении приведем примеры алгоритмов. Как отмечалось ранее, под термином «эволюционные алгоритмы» понимается не одна модель, а достаточно широкий класс алгоритмов, подчас мало похожих друг на друга. Приведем схемы двух из них.

1. $(\mu + \lambda)$ – *эволюционная стратегия* (ЭС).
- 1.0. Представление исходной задачи в терминах ЭС.
- 1.1. Создание начальной популяции размера λ .
- 1.2. Вычисление функции пригодности $f(x^i)$, $i = 1, \dots, \lambda$.
- 1.3. Выбор $\mu \leq \lambda$ лучших особей.
- 1.4. Создание $\frac{\lambda}{\mu}$ потомков из каждой из μ особей с малой

вариацией координат. Возврат на шаг 1.2.

По сути, данный алгоритм представляет собой *случайный поиск*, использующий селекцию и мутацию.

2. Генетический алгоритм (ГА).

2.0. Представление исходной задачи в терминах ГА.

2.1. Создание начальной популяции $P_0 = \{x_0^1, \dots, x_0^N\}$, $t = 0$. Заметим, что нижний индекс здесь означает не координату, а номер популяции.

2.2. Вычисление *средней пригодности текущей популяции*:

$$\bar{f}(t) = \frac{\sum_{i=1}^N f(x_t^i)}{N} \quad (8.8)$$

и «нормализованной» *пригодности каждой особи*:

$$\bar{f}(x_t^i) = \frac{f(x_t^i)}{\bar{f}(t)}. \quad (8.9)$$

2.3. Вычисление для каждой особи вероятности, с которой она может быть выбрана в качестве родителя для создания потомка. Чаще всего эта вероятность выбирается пропорционально ее пригодности (пропорциональная или «рулеточная» селекция), исходя из формулы (8.5):

$$p(x_t^i) = \frac{f(x_t^i)}{\sum_{j=1}^N f(x_t^j)} = \frac{\bar{f}(x_t^i)}{N}. \quad (8.10)$$

Используя это распределение, выбирается N особей из популяции P_t , тем самым создается набор родителей. Отметим, что родители могут повторяться.

2.4. Случайным образом формируется $N/2$ пар. Применяя к каждой паре с определенной вероятностью операторы рекомбинации и мутации, формируется новая популяция P_{t+1} . Возврат на шаг 2.2.

Литература

- [1] Алферов А.П., Зубков А.Ю., Кузьмин А.С., Черемушкин А.В. *Основы криптографии*. М.: Гелиос АРВ, 2001. 480 с.
- [2] Батищев Д. И. *Генетические алгоритмы решения экстремальных задач*. Н.Новгород, 1995. 69 с.
- [3] Богаченко Н.Ф., Файзуллин Р.Т. *Синтез дискретных автоматов*. Учебно-методическое пособие. Омск: Издательство Наследие. Диалог-Сибирь, 2006. 150 с.
- [4] Данилова О.Т., Файзуллин Р.Т. *Проектирование комбинационных схем*. Учебно-методическое пособие. Омск: Издательство Наследие, 2004. 178 с.
- [5] Гэри М., Джонсон Д. *Вычислительные машины и труднорешаемые задачи*. М.: Мир, 1982. 416 с.
- [6] Иванов М.А. *Криптографические методы защиты информации в компьютерных системах и сетях*. М.: КУДИЦ – ОБРАЗ, 2001. 368 с.
- [7] Котов В.Е. *Сети Петри*. М.: Наука, 1984. 160 с.
- [8] Кук Д., Бейз Г. *Компьютерная математика*. М.: Наука, 1990. 384 с.

- [9] Леонтьев В.К. *Комбинаторика: ретроспектива и перспективы.* / Компьютер и задачи выбора. М.: Наука, 1989. С. 49–88.
- [10] Молдовян А.А., Молдовян Н.А., Советов Б.Я. *Криптография.* СПб.: Лань, 2001. 224 с.
- [11] Пенроуз Р. *Новый ум короля. О компьютерах, мышлении и законах физики.* М.: Едиториал УРСС, 2003. 384 с.
- [12] Полак Э. *Численные методы оптимизации.* М.: Мир, 1974. 376 с.
- [13] Саломая А. *Криптография с открытым ключом.* М.: Мир, 1995. 320 с. (Электронная версия.)
- [14] Столлингс В. *Криптография и защита сетей: принципы и практика.* Издательский дом «Вильямс», 2001. 672 с.
- [15] Тоффоли Т., Марголус Н. *Машины клеточных автоматов.* М.: Мир, 1991. 280 с.
- [16] Трахтенброт Б.А. *Алгоритмы и машинное решение задач.* М.: Государственное издательство технико-теоретической литературы, 1957. 95 с.
- [17] Чистяков В. П. *Курс теории вероятностей.* М.: Наука, 1982. 156 с.
- [18] Шамашов М.А. *Теория формальных языков. Грамматики и автоматы.* Учебное пособие. Самара, 1996. (Электронная версия.)
- [19] Шнайер Б. *Прикладная криптография. Протоколы, алгоритмы, исходные тексты на языке Си.* М.: ТРИУМФ, 2003. 816 с.
- [20] Hammel U., Bäck T. *Evolution Strategies on Noisy functions How to Improve Convergence Properties* // Parallel Problem Solving from Nature – PPSN III. Proceedings. Jerusalem. Israel. October. 1994.

-
- [21] Hoffmeyer F., Bäck T. *Genetic Algorithms and Evolution Strategies: Similarities and Differences* // Parallel Problem-Solving from Nature. H-P. Schwefel and R. Männer (eds.). Springer-Verlag, Berlin. 1991.
 - [22] Goldberg D. E. *Sizing Populations for Serial and Parallel Genetic Algorithms.* / Genetic Algorithms. Edited by B.P. Buckles & F.E. Petry. From The Proceedings of the Third International Conference on Genetic Algorithms. Morgan Kaufmann Publishers. San Mateo. California. 1989.
 - [23] Reeves C. R. *Genetic Algorithms for the Operations Researcher* // INFORMS Journal on Computing. V.9. N 3. 1997.

Оглавление

| | | |
|----------|---|-----------|
| 1 | Языки и грамматики | 3 |
| 1.1. | Основные определения | 4 |
| 1.2. | Грамматика | 6 |
| 1.3. | Иерархия Хомского | 10 |
| 1.4. | Построение вывода | 13 |
| 1.4.1. | Синтаксическое дерево | 13 |
| 1.4.2. | Синтаксический разбор | 14 |
| 1.4.3. | Характеристики вывода | 16 |
| 1.5. | Построение КС- и А-грамматик | 17 |
| 1.5.1. | Структура слов и правила грамматики | 18 |
| 1.5.2. | Описание списков | 18 |
| 1.5.3. | Основные конструкции языков программирования | 21 |
| 1.5.4. | Различия между КС- и А-грамматиками | 25 |
| 1.6. | Представление А-грамматики в виде графа состояний | 28 |
| 1.7. | Преобразование КС-грамматик | 31 |
| 1.8. | Разрешимость | 35 |
| 2 | Распознаватели | 36 |
| 2.1. | Автоматы как распознаватели | 36 |
| 2.2. | Конечные автоматы и А-грамматики | 41 |
| 2.3. | Магазинные автоматы и КС-грамматики | 43 |

| | | |
|----------|---|------------|
| 3 | Машина Тьюринга | 50 |
| 3.1. | Концепция Тьюринга | 51 |
| 3.2. | Расширенная двоичная форма записи | 55 |
| 3.3. | Основная гипотеза теории алгоритмов | 58 |
| 3.4. | Универсальная машина Тьюринга | 59 |
| 3.5. | Алгоритмически неразрешимые проблемы | 62 |
| 4 | Сети Петри | 68 |
| 4.1. | События и условия | 68 |
| 4.2. | Формальное определение сетей Петри | 73 |
| 4.3. | Функционирование сетей Петри | 74 |
| 4.4. | Свойства сетей Петри | 79 |
| 4.5. | Языки сетей Петри | 81 |
| 4.6. | Сети Петри и программирование | 84 |
| 5 | Клеточные автоматы | 94 |
| 5.1. | Задание клеточного автомата | 94 |
| 5.2. | Игра «Жизнь» | 97 |
| 5.3. | Правила клеточного автомата | 98 |
| 6 | Криптосистемы с открытым ключом | 103 |
| 6.1. | Криптосистемы, основанные на грамматиках | 103 |
| 6.2. | Криптосистемы, основанные на композиции морфизмов | 106 |
| 6.3. | Криптосистемы, основанные на последовательных автоматах | 111 |
| 7 | Элементы теории сложности | 113 |
| 7.1. | Временная сложность алгоритмов | 113 |
| 7.2. | Классы \mathcal{P} и \mathcal{NP} | 115 |
| 7.3. | \mathcal{NP} -полные задачи | 119 |
| 8 | Эволюционные алгоритмы | 126 |
| 8.1. | Терминология | 127 |
| 8.2. | Основные операторы эволюционных алгоритмов | 129 |
| | Литература | 135 |

Н.Ф. Богаченко, Р.Т. Файзуллин

Автоматы, грамматики, алгоритмы

Учебно-методическое пособие

Авторское редактирование

Подготовлено к печати
ООО «Издательство Наследие. Диалог-Сибирь»
Лицензия ЛР N 071680 от 04.06.98.
Подписано в печать 20.05.2006.
Формат 60 × 84 1/16. Усл.печ.л. 8,75. Уч.-изд.л. 8,60.
Тираж 100 экз.

Полиграфический центр КАН
644050, Омск-50, пр. Мира, 32, к.11
тел. (3812) 65-47-31
Лицензия ПЛД N 58-47 от 21.04.97.