

С. С. Гайсарян  
В. Е. Зайцев

# КУРС ИНФОРМАТИКИ

Издание 12-е, предварительное, переработанное и дополненное  
с приложением хрестоматии на компакт-дисках

Допущено Министерством  
образования Российской  
Федерации в качестве учебного  
пособия для студентов вузов,  
обучающихся по специальности  
«Прикладная математика  
и информатика»

Москва  
Издательство Вузовская книга  
2012

ББК 16.2.12

Г147

УДК 519.3 (075.8)

Рецензенты:

академик РАН Иванников В. П.

д-р. физ.-мат. наук, проф. Жоголев Е. А.

д-р. физ.-мат. наук Цейтин Г. С.

**Гайсарян С. С., Зайцев В. Е.**

Г147 Курс информатики: Учеб. пособие. — М.: Изд-во Вузовская книга,

2012. — 424 с.: ил. — с приложением на CD ROM.

ISBN 5-7035-XXXX-X

Фундаментальные основы современной информатики излагаются на базе надъязыкового математического подхода.

Учебное пособие предназначено для студентов специальностей, предполагающих университетскую подготовку по информатике, и может быть полезно для самообразования с целью углубления теоретических знаний.

Курс сопровождается практикумом, описанном в отдельном пособии.

В прилагаемую к пособию хрестоматию на компакт-диске включены многочисленные оригинальные и свободно-распространяемые системы, статьи, документы и материалы которые могут применяться при изучении курса.

Г **1602120000-10** 6-90  
094(02)-92

**ББК 16.2.12**

ISBN 5-7035-XXXX-X

©С. С. Гайсарян 1983–2012, В. Е. Зайцев 1993–2012

# Оглавление

<b>Введение</b>	<b>7</b>
Предмет курса . . . . .	7
Структура курса . . . . .	8
Рекомендуемая литература . . . . .	9
<b>1 Основные понятия информатики</b>	<b>16</b>
1.1 Информация и сообщение . . . . .	16
1.2 Интерпретация сообщений . . . . .	16
1.3 Знаки и символы . . . . .	18
1.4 Кодирование . . . . .	19
1.5 Системы счисления . . . . .	24
1.6 Обработка сообщений . . . . .	29
1.7 Обработка информации . . . . .	30
1.8 Автоматизация обработки информации . . . . .	31
1.9 Конструктивное описание процесса обработки дискретных сообщений . . . . .	32
1.10 Интерпретация дискретных сообщений . . . . .	37
<b>2 Элементы теории алгоритмов</b>	<b>40</b>
2.1 Необходимость формального определения алгоритма . . . . .	40
2.1.1 Рекурсивные функции . . . . .	41
2.1.2 Машины Тьюринга . . . . .	41
2.1.3 Нормальные алгоритмы Маркова . . . . .	42
2.1.4 Тезис Тьюринга-Черча . . . . .	42
2.2 Машины Тьюринга . . . . .	42
2.2.1 Неформальное описание . . . . .	42
2.2.2 Математически строгое определение . . . . .	44
2.2.3 Диаграммы Тьюринга . . . . .	47
2.2.4 Понятие моделирования . . . . .	53
2.2.5 Эквивалентность диаграмм и программ МТ . . . . .	54
2.3 Нормальные алгоритмы Маркова . . . . .	57
2.4 Исследование алгоритмической модели Тьюринга . . . . .	63
2.4.1 Теоремы Шеннона . . . . .	63
2.4.2 Доказательство I теоремы Шеннона (К. Шенон, 1956) . . . . .	65
2.4.3 Доказательство I теоремы Шеннона (Д. Рисенберг, 2005) . . . . .	69
2.4.4 Доказательство теоремы 2.4.2 . . . . .	75
2.5 Вычислимые функции . . . . .	78

2.5.1	Понятие функции на множестве слов . . . . .	78
2.5.2	Понятие вычислимости . . . . .	79
2.5.3	Функции, вычислимые по Тьюрингу . . . . .	79
2.5.4	Нормированные вычисления . . . . .	79
2.5.5	Предикаты, вычислимые по Тьюрингу . . . . .	81
2.6	Теоремы о сочетаниях алгоритмов . . . . .	82
2.6.1	Теорема о композиции . . . . .	82
2.6.2	Теорема о ветвлении . . . . .	84
2.6.3	Следствие из теоремы о ветвлении . . . . .	85
2.6.4	Пример итеративного алгоритма . . . . .	85
2.6.5	Теорема о цикле . . . . .	87
2.6.6	Обобщенная теорема о цикле . . . . .	88
2.7	Схемы машин Тьюринга . . . . .	91
2.7.1	Конструирование машин Тьюринга «сверху вниз» . . . . .	91
2.7.2	Определение схемы машины Тьюринга . . . . .	95
2.7.3	Эквивалентность схем и диаграмм . . . . .	96
2.7.4	Доказательство теоремы о нормированной вычислимости . . . . .	100
2.7.5	Некоторые фундаментальные результаты теории алгоритмов . . . . .	107
2.8	Развитие алгоритмической модели Тьюринга . . . . .	107
2.8.1	Понятие об универсальной машине Тьюринга . . . . .	107
2.8.2	Линейная запись схем машин Тьюринга . . . . .	109
2.8.3	Язык описания схем машин Тьюринга . . . . .	111
2.9	Модель фон Неймана . . . . .	113
2.9.1	Критика модели вычислений Тьюринга . . . . .	113
2.9.2	Адреса и имена . . . . .	114
2.9.3	Построение процессора фон Неймана . . . . .	118
2.9.4	Согласование процессоров . . . . .	121
2.9.5	Машина фон Неймана . . . . .	121
<b>3</b>	<b>Концепция языка программирования для машины фон Неймана</b>	<b>123</b>
3.1	Элементы дейкстровской нотации . . . . .	123
3.1.1	Требования к структуре программ для машины фон Неймана . . . . .	123
3.1.2	Обобщенная инструкция присваивания . . . . .	126
3.1.3	Обобщенная инструкция композиции . . . . .	131
3.1.4	Охраняемые инструкции . . . . .	133
3.1.5	Обобщенная инструкция ветвления . . . . .	134
3.1.6	Обобщенная инструкция цикла . . . . .	135
3.2	Типы данных . . . . .	138
3.2.1	Определение типа данных . . . . .	138
3.2.2	Тип логический . . . . .	143
3.2.3	Тип целый . . . . .	146
3.2.4	Тип вещественный . . . . .	156
3.2.5	Тип литерный . . . . .	172
3.2.6	Согласование типов . . . . .	174
3.2.7	Небазовые типы данных . . . . .	175
3.2.8	Понятие о структурном типе данных . . . . .	175

3.2.9	Тип массив . . . . .	176
3.2.10	Понятие о записях . . . . .	177
3.2.11	Понятие о файлах . . . . .	179
3.3	Блочная структура программ . . . . .	182
3.4	Процедуры и функции . . . . .	185
3.4.1	Описание и использование . . . . .	186
3.4.2	Вызов функций и процедур. Формальные и фактические параметры	186
3.4.3	Передача параметров . . . . .	187
3.4.4	Побочные эффекты . . . . .	191
3.4.5	Критика алгоритмической модели фон Неймана . . . . .	192
3.5	Критика языка Паскаль . . . . .	194
<b>4</b>	<b>Распределение памяти</b>	<b>196</b>
4.1	Статические и динамические объекты программ . . . . .	196
4.2	Ссылочный тип . . . . .	197
<b>5</b>	<b>Структуры данных</b>	<b>203</b>
5.1	Уровни описания структур данных . . . . .	203
5.2	Файл . . . . .	207
5.2.1	Функциональная спецификация. . . . .	208
5.2.2	Логическое описание и физическое представление . . . . .	209
5.3	Вектор . . . . .	213
5.3.1	Функциональная спецификация . . . . .	213
5.3.2	Логическое описание и физическое представление . . . . .	214
5.4	Очередь . . . . .	216
5.4.1	Функциональная спецификация. . . . .	218
5.4.2	Логическое описание и физическое представление . . . . .	219
5.5	Стек . . . . .	234
5.5.1	Функциональная спецификация . . . . .	236
5.5.2	Логическое описание . . . . .	238
5.6	Линейный список . . . . .	243
5.6.1	Функциональная спецификация . . . . .	245
5.6.2	Логическое описание . . . . .	246
5.6.3	Физическое представление . . . . .	247
5.7	Дек . . . . .	259
5.8	Списки общего вида . . . . .	266
5.9	Понятие о рекурсии . . . . .	266
5.10	Деревья . . . . .	275
5.10.1	Двоичные деревья . . . . .	278
5.10.2	Двоичная интерпретация дерева общего вида . . . . .	279
5.10.3	Функциональная спецификация . . . . .	281
5.10.4	Логическое описание . . . . .	282
5.11	Алгоритмы обработки деревьев . . . . .	283
5.11.1	Понятие обхода дерева . . . . .	283
5.11.2	Алгоритмы обхода двоичных деревьев . . . . .	284
5.11.3	Построение и визуализация дерева . . . . .	286
5.11.4	Деревья выражений. Разнофиксные формы записи выражений . . . . .	288

5.11.5	Обход дерева общего вида . . . . .	303
5.12	Деревья поиска . . . . .	303
5.12.1	Поиск по дереву с включениями . . . . .	305
5.12.2	Исключение из деревьев . . . . .	308
5.12.3	Сбалансированные деревья . . . . .	310
5.12.4	Физическое представление. Отображение на массив.	316
5.13	Графы . . . . .	318
5.13.1	Алгоритмы на графах . . . . .	321
<b>6</b>	<b>Сортировка и поиск</b>	<b>325</b>
6.1	Алгоритмы поиска . . . . .	325
6.1.1	Линейный поиск . . . . .	325
6.1.2	Двоичный поиск . . . . .	326
6.1.3	Поиск в таблице . . . . .	329
6.1.4	Поиск по образцу . . . . .	332
6.1.5	Алгоритм Кнута-Мориса-Пратта . . . . .	334
6.1.6	Алгоритм Бойера-Мура . . . . .	340
6.1.7	Алгоритм Рабина-Карпа . . . . .	344
6.2	Алгоритмы сортировки . . . . .	347
6.2.1	Введение . . . . .	347
6.2.2	Сортировка массивов . . . . .	348
6.2.3	Сравнение методов внутренней сортировки . . . . .	381
6.2.4	Внешние сортировки . . . . .	383
6.3	Таблицы с прямым доступом . . . . .	387
6.3.1	Выбор хеш-функции . . . . .	388
6.3.2	Рехеширование . . . . .	389
<b>7</b>	<b>Методы программирования</b>	<b>393</b>
7.1	Модульное программирование . . . . .	393
7.1.1	Модули в расширениях языка Паскаль . . . . .	394
7.1.2	Экспорт и импорт объектов . . . . .	409
7.2	Программирование в абстрактных типах данных . . . . .	409
7.2.1	Методы абстракции в языках программирования . . . . .	410
7.2.2	Виды абстракции в языках программирования . . . . .	414
7.3	Типовые абстракции . . . . .	419
7.3.1	Типизация языка . . . . .	419
7.3.2	Контроль типов . . . . .	420
7.3.3	Преобразование и передача типов . . . . .	420
7.3.4	Адресный тип . . . . .	421
7.3.5	Родовые модули . . . . .	422
7.3.6	Полиморфизм . . . . .	422
7.3.7	Процедурный тип . . . . .	425
7.3.8	Элементы объектно-ориентированного программирования в расширениях языка Си . . . . .	426
7.4	Объектно-ориентированное программирование . . . . .	435
7.4.1	Наследование . . . . .	435
7.4.2	Построение объектной технологии . . . . .	437

7.4.3	Объектно-ориентированное программирование и структурное про- граммирование . . . . .	438
7.4.4	ООП в фон Неймановских языках . . . . .	439
7.4.5	ООП в абсолютно объектных средах . . . . .	445
<b>8</b>	<b>Языки программирования не фон Неймановских моделей</b>	<b>446</b>
8.1	Языки реляционного типа (SETL, SQL) . . . . .	446
8.2	Языки логического программирования (PROLOG) . . . . .	446
8.3	Языки функционального программирования (LISP, AFP) . . . . .	449

# Введение

## Предмет курса

Электронные вычислительные машины (сокращенно ЭВМ) применяются почти во всех областях человеческой деятельности: в научных исследованиях, на производстве, на транспорте, в торговле, в учебных заведениях, в медицине, в военном деле, в делопроизводстве, в спорте. Этот список нетрудно продолжить, тем более, что он непрерывно пополняется — ЭВМ стремительно захватывают новые сферы применения. Столь широкое распространение ЭВМ объясняется тем, что любой вид человеческой деятельности связан с обработкой информации, а ЭВМ является средством автоматизации обработки информации.

Вместе с ЭВМ развивается «наука об ЭВМ» — **информатика**, которая является предметом нашего изучения. Информатика объединяет несколько более или менее независимых дисциплин. Одной из таких дисциплин является программирование.

Изучению основных принципов и приемов программирования, задач обработки информации для ЭВМ посвящается большая часть курса. Остальные разделы информатики будут изучаться лишь в той мере, которая необходима для сознательного усвоения методов программирования.

Информатику определяют как науку об автоматической обработке информации с помощью ЭВМ: [1]

*Информатика: наука об осуществляющей преимущественно с помощью автоматических средств целесообразной обработке информации, рассматривающей как представление знаний и сообщений в технических, экономических и социальных областях.*

(Французская Академия)

*L'INFORMATIQUE: Science de traitement rationnel, notamment par machines automatiques, de l'information considérée comme le support des connaissances humaines et des communications, dans les domaines techniques, économiques et sociaux*

(Académie Française)

Д. Кнут в своей статье «Информатика и ее связь с математикой» [34] пишет: «Лучший с моей точки зрения, способ определить информатику — это сказать, что она занимается изучением алгоритмов».

## Дональд Кнут



Дональд Э. Кнут (Donald E. Knuth) — выдающийся американский ученый в области информатики, профессор Стенфордского университета, автор монументальной серии монографий «Искусство программирования», создатель системы компьютерной полиграфии TeX, в которой подготовлен оригинал-макет этой книги. В последнее время вышли новые книги Дональда Кнута: «Конкретная математика» (в соавторстве с Грэхемом и Паташником), «Компьютерная типография» и др.

Дональд Кнут является почетным членом многих академий, почетным доктором ряда университетов, в том числе Санкт-Петербургского. Дональд Кнут удостоен национальной научной медали США в 1979 году.



---

Оба эти определения не очень понятны, так как они содержат еще не определенные понятия, такие, как «информация», «алгоритм». С изучения этих понятий мы и начнем.

Отметим, что информатика использует многие математические методы и во многом подобна математике. Так же, как и математика, информатика изучает законы, созданные человеком и поддающиеся «доказательству», в отличие от естественных законов (изучаемых такими науками, как физика, химия, биология и т. д.), знание которых всегда сопровождается некоторой долей неопределенности. Но информатика и математика — две разные науки. Разница между ними заключается в предмете и подходе — математика обычно имеет дело с теоремами, бесконечными процессами и статическими соотношениями, а информатика — с алгоритмами, конечными конструкциями и динамическими соотношениями [1].

Как гласит программистский фольклор, математика делает то что можно так, как нужно, а информатика — то что нужно так, как можно [19].

## Структура курса

На I курсе предусмотрены следующие формы занятий: лекции (204 часа), лабораторные (204 часа) занятия, консультации по курсовым проектам и работам (85 часов на группу), самостоятельная работа (310 часов). Контрольные мероприятия: зачеты с оценкой по лабораторным работам и заданиям курсовых проектов (включая входной контроль знаний), зачеты по курсовым проектам и работам (с оценкой), экзамен (письменный);

во 2-ом семестре — с отладкой программ на ЭВМ). Кроме того, проводятся олимпиады, тестирование знаний, коллоквиумы и репетиционные экзамены, автоматизированное, индивидуальное и дистанционное обучение.

На **лекциях** излагается теоретический материал: основные понятия, теоремы, алгоритмы. Проводимые с помощью мультимедийной компьютерной проекционной аппаратуры лекционные демонстрации предусматривают показ работы алгоритмов и программ и другую иллюстративную поддержку курса.

**Лабораторные работы** посвящены практическому освоению вычислительных машин, систем программного обеспечения, конкретных методов и приемов программирования, осуществляющему, как правило, путем разработки и отладки небольших программ в учебной вычислительной лаборатории.

**Практические занятия** заключаются в закреплении материала лекций и разъяснении его применения к решению практических задач путем групповых занятий в аудиториях. На практических занятиях под руководством преподавателя проводится составление типовых программ по всем основным разделам курса, включая тематику курсовых и лабораторных работ, осуществляются различные контрольные мероприятия.

**Курсовые работы и проекты** состоят из нескольких заданий, которые выполняются студентами самостоятельно в неаудиторное время. Особенностью курсовых проектов по информатике является разработка программной документации в соответствии с правилами ЕСПД и ЕСКД, в то время как основными документами инженерного проектирования являются чертежи. Подробнее о курсовых и лабораторных работах сказано в ежегодно обновляемом пособии по практикуму, издаваемому в бумажном и электронном (CD) виде.

Значение курсовых и лабораторных работ трудно переоценить, так как, чтобы до конца понять суть информатики, *необходимо программировать самому*.

Постановка курса осуществлена в 1976–1983 гг. к. ф.-м. н., доц. С. С. Гайсаряном.

С 1993 г. курс читается к. ф.-м. н., доц. Зайцевым В. Е. с практикумом, описанном в учебном пособии «Информатика. Практикум». — М.: 1993. С 1997 г. курс комплектуется CD-хрестоматией.

Большую помощь в издании курса оказали к. ф.-м. н., доц. Журавлева Т. Э., Сошников Д. В. и Титов В. К., ст. преп. Сеницкий П. А., Дзюба Д. В. и Овечкис А. Г., преп. Перетягин И. А. и студ. Рисенберг Д. В.

## Рекомендуемая литература

Среди изданных к настоящему времени учебников и учебных пособий по информатике и программированию нет ни одного, которое содержало бы все разделы курса. Наиболее близкими по содержанию являются учебные пособия [1] и [2], но они труднодоступны, так как изданы малыми тиражами. При изучении материала курса можно пользоваться учебным пособием [3], методическими разработками [4-6], книгами [7-13]. Отдельные вопросы, рассматриваемые в курсе, изложены в книгах [14-29,31-40]. Практикум по курсу проводится в соответствии с пособием [30].

# Литература

- [1] Ф. Бауэр, Т. Гооз. Информатика. — М.: Мир, 1976, 1990.
- [2] Б. Мейер, К. Бодуэн. Методы программирования. Т. 1 и 2. — М.: Мир, 1982.
- [3] Э. З. Любимский, В. В. Мартынюк, Н. П. Трифонов. Программирование. — М.: Наука, 1980.
- [4] Принципы программирования на абсолютно универсальных вычислительных машинах. Методическая разработка. // С. С. Гайсарян, И. З. Луговая. — М.: 1980.
- [5] С. С. Гайсарян, И. З. Луговая, В. Д. Трасковский. Основные приемы программирования: типы данных и структура программ. — М.: 1981.
- [6] С. С. Гайсарян, И. З. Луговая, В. Д. Трасковский. Средства организации программ сложной структуры. — М.: 1982.
- [7] Г.-Д. Эббинхауз, К. Якобс, Ф.-К. Ман, Г. Хермес. Машины Тьюринга и рекурсивные функции. — М.: Мир, 1972.
- [8] Н. Вирт. Систематическое программирование. Введение. — М.: Мир, 1977.
- [9] К. Йенсен, Н. Вирт. Паскаль. Руководство для пользователя и описание языка. — М.: Финансы и статистика, 1982, 1989.
- [10] П. Грогоно. Программирование на языке Паскаль. — М.: Мир, 1982.
- [11] Э. Дейкстра. Дисциплина программирования. — М.: Мир, 1978.
- [12] Т. Пратт. Языки программирования. Разработка и реализация. — М.: Мир, 1982.
- [13] Г. Майерс. Надежность программного обеспечения. — М.: Мир, 1980.
- [14] Д. Цикритзис, Ф. Бернштайн. Операционные системы. — М.: Мир, 1977.
- [15] У. Дал, Э. Дейкстра, К. Хоор. Структурное программирование. — М.: Мир, 1975.
- [16] Б. А. Трахтенброт. Алгоритмы и вычислительные автоматы. — М.: Сов. радио, 1974. — 200 с., ил.
- [17] Э. З. Любимский, В. В. Мартынюк. Элементы теории алгоритмов и структур данных. — М.: МГУ, 1976.

- [18] К. Шенон. Универсальная машина Тьюринга с двумя внутренними состояниями. — В кн.: Работы по теории информации и кибернетике. — М.: ИЛ, 1963. с. 740-750.
- [19] Е. А. Жоголев. Введение в технологию программирования. Конспект лекций. — М.: Диалог-МГУ, 1995.
- [20] М. Бвой. Информатика. Основополагающее введение. В 4-х частях. — М.: Диалог-МИФИ, 1996.
- [21] В. Ш. Кауфман. Языки программирования. Концепции и принципы. — М.: Радио и связь, 1993.
- [22] Э. Танненбаум. Многоуровневая организация ЭВМ. — М.: Мир, 1979.
- [23] Ю. А. Шрейдер, А. А. Шаров. Системы и модели. — М.: Радио и связь, 1982. — 152 с.
- [24] Т. Тоффоли, Н. Марголус. Машины клеточных автоматов. — М.: Мир, 1991.
- [25] А. А. Марков, Н. М. Нагорный. Теория алгорифмов. — М.: Наука, 1984.
- [26] Лекции лауреатов премии Тьюринга. / Пер. с англ. — М.: Мир, 1993. — 560 с., ил.
- [27] Н. Вирт. Программирование на языке Модула-2. — М.: Мир, 1987.
- [28] В. А. Успенский. Машина Поста. (Популярные лекции по математике. Вып. 54). — М.: Наука, 1979.
- [29] В. А. Успенский. Лекции о вычислимых функциях. — М.: Наука, 1960.
- [30] В. Е. Зайцев и др. Информатика. Практикум. // Учеб. пособие. — М.: 1993.
- [31] Н. К. Косовский. Элементы математической логики и ее приложения к теории субрекурсивных алгоритмов. — Л.: ЛГУ, 1981.
- [32] Ю. А. Шрейдер. Логика знаковых систем. — М.: Знание, 1974.
- [33] А. И. Мальцев. Алгоритмы и рекурсивные функции. — М.: Наука, 1965.
- [34] Д. Кнут. Информатика и ее связь с математикой // Сб. статей «Современные проблемы математики» — М.: Знание, 1977, стр. 4–32.
- [35] С. Янг. Алгоритмические языки реального времени. — М.: Мир, 1985.
- [36] А. Шень. Программирование: теоремы и задачи. — М.: МЦНМО, 1995. — 263 с., ил.
- [37] Д. Бентли. Жемчужины творчества программистов. — М.: Радио и связь, 1990. — 224 с., ил.
- [38] Р. Лингер, Х. Миллс, Б. Уитт. Теория и практика структурного программирования // — М.: Мир, 1982.
- [39] А. П. Ершов. Введение в теоретическое программирование. — М.: Наука, 1976.

- [40] Б. Лисков, Дж. Гатэг. Использование абстракций и спецификаций при разработке программ. — М.: Мир, 1989. — 424 с., ил.
- [41] Языки программирования Ада, Си, Паскаль // Под ред. А. Фьюера, Н. Джехани. — М.: Радио и связь, 1989.
- [42] Э. Йодан. Структурное проектирование и конструирование программ // — М.: Мир, 1979.
- [43] И. З. Луговая, Л. Н. Чернышов, С. М. Юдин. Динамические структуры данных языка Паскаль: Учебное пособие. — М.: 1988. — 57 с., ил.
- [44] Л. Н. Чернышов , С. М. Юдин. Инструментальная система инфорт и работа с динамическими структурами данных: Учебное пособие. — М.: 1984. — 68 с., ил.
- [45] Технология программирования. Языки программирования, операционные системы, базы данных. — М.: МИУ, 1995, N 0, 1, CD приложение.
- [46] Данные в языках программирования. Абстракция и типология: Сб. статей. Пер. с англ. под ред. В. Н. Агафонова. — М.: Мир, 1982.
- [47] Д. Баррон. Рекурсивные методы в программировании. — М.: Мир, 1974.
- [48] Л. Броуди. Начальный курс программирования на языке Форт. — М.: Финансы и статистика, 1990.
- [49] А. Т. Берзтисс. Структуры данных. — М.: Статистика, 1974.
- [50] С. С. Лавров, Г. С. Силагадзе. Язык Лисп и его реализация. — М.: Наука, 1978.
- [51] С. П. Никулин. Регулярные и комбинированные структуры данных / Под ред. В. Е. Зайцева — М.: 1997.
- [52] Г. Лорин. Сортировка и системы сортировки. — М.: Наука, 1983.
- [53] Н. Вирт. Алгоритмы + Структуры данных = Программы. — М.: Мир, 1985.
- [54] Н. Вирт. Алгоритмы и структуры данных. — М.: Мир, 1989.
- [55] Turbo Pascal 7.0. — К.: BHV, 1996.
- [56] GNU Pascal. Шпаргалка пользователя. — М.: 1997.
- [57] К. Клоксин, Б. Меллиш. Программирование на Прологе. — М.: Мир, 1986.
- [58] И. Братко. Программирование на языке Пролог для искусственного интеллекта.
- [59] Д. Райли. Абстракция и структуры данных. — М.: Мир, 1993.
- [60] К. Фути, Н. Судзуки. Языки программирования и схемотехника СБИС. — М.: Мир, 1988.
- [61] Языки программирования Ада, Си, Паскаль. Сравнение и оценка / Под ред. А. Р. Фьюера, Н. Джехани. — М.: Радио и связь, 1989.

- [62] С. Б. Карасев, Т. Е. Кошелева, Л. Н. Чернышов. Машины алгоритмы обработки информации. — М.: 1987.
- [63] Д. Кнут. Искусство программирования для ЭВМ. т. 1. Основные алгоритмы. — М.: Мир, 1976.
- [64] Д. Кнут. Искусство программирования для ЭВМ. т. 2. Получисленные алгоритмы. — М.: Мир, 1977.
- [65] Д. Кнут. Искусство программирования для ЭВМ. т. 3. Сортировка и поиск. — М.: Мир, 1978.
- [66] К. Кристиан, Руководство по программированию на языке Модула-2. — М.: Мир, 1989.
- [67] К. Дейт. Введение в системы баз данных. — М.: Наука, 1981, К.: Бином, 1997.
- [68] О. С. Разумов. Организация данных в вычислительных системах. — М.: Финансы и статистика, 1978.
- [69] Д. Я. Левин. Сетл — язык весьма высокого уровня. — Программирование, 1976, №. 5, с. 3–9.
- [70] Д. Грис. Конструирование компиляторов для цифровых вычислительных машин. — М.: Мир, 1975.
- [71] М. Брой. Информатика. Структуры систем и системное программирование. Ч. 3. — М.: Диалог-МИФИ, 1996. — 224 с.
- [72] Б. Мейер, К. Бодуэн. Методы программирования. Т. 1 — М.: Мир, 1982.
- [73] Г. Буч. Объектно-ориентированное проектирование с примерами применения. — М.: Конкорд, 1992. — 519 с., ил.
- [74] В. Бердж. Методы рекурсивного программирования. — М.: Машиностроение, 1983. — 248 с., ил.
- [75] Генри С. Уоррен. Алгоритмические трюки для программистов: Пер. с англ. — М.: Издательский дом «Вильямс», 2004. — 288 с.: ил.
- [76] Р. Грэхем, Д. Кнут, О. Паташник. Конкретная математика. Основание информатики. — М.: Мир, 1998.
- [77] Ф. А. Новиков. Дискретная математика для программистов. — СПб.: Изд. Питер, 2002.
- [78] И. Ильф, Е. Петров. 12 стульев. Сатирический роман. // Собрание сочинений в пяти томах, т. I. ГИХЛ, Москва, 1961.
- [79] А. С. Кронрод. Беседы о программировании. — М.: УРСС, 2004. -248с., ил.
- [80] С. В. Фомин. Системы счисления. — М.: Наука, 1987. —48с. —(Попул. лекции по мат.)

- [81] В. Ф. Турчин. Феномен науки: Кибернетический подход к эволюции. — М.: ЭТС. — 2000. — 368 с.
- [82] Ильин В. А., Позняк Э. Г. Основы математического анализа. Часть I. — М.: Наука, 1982, с. 35–57.
- [83] Дж. Форсайт, М. Малькольм, К. Моулер. Машинные методы математических вычислений. — М.: Мир, 1980, с. 9–42.
- [84] М. Бен-Ари. Языки программирования. Практический сравнительный анализ: Пер. с англ. — М.: Мир, 2000. — 366 с., ил.
- [85] Т. Кормен, Ч. Лейзерсон, Р. Ривест. Алгоритмы: построение и анализ. Пер. с англ. — М.: МЦНМО, 2000. — 960 с., 263 ил. (ISBN 5-900916-37-5).
- [86] Р. Хэз菲尔д, Л. Кирби и др. Искусство программирования на С. Фундаментальные алгоритмы, структуры данных и примеры приложений. Энциклопедия программиста: Пер. с англ. — К.: Издательство «ДиаСофт», 2001. — 736 с.
- [87] Д. Гасфилд. Строки, деревья и последовательности в алгоритмах: Информатика и вычислительная биология. — СПб.: Невский Диалект, БХВ-Петербург, 2003. — 654 с., ил.
- [88] Р. Седжвик. Фундаментальные алгоритмы на C++. Анализ/Структуры данных/-Сортировка/Поиск: Пер. с англ./Роберт Седжвик. — СПб.: ООО «ДиаСофтЮП», 2002. — 688 с.
- [89] С. Макконнелл. Совершенный код. Мастер-класс / Пер. с англ. — М.: Издательство торговый дом «Русская редакция»; СПб.: Питер, 2005. — 896 с.: ил.
- [90] А. Ахо, Дж. Хопкрофт, Дж. Ульман. Структуры данных и алгоритмы.: Пер. с англ. — М.: Издательский дом «Вильямс», 2003. — 384 с.: ил. — Парал. тит. англ.
- [91] Р. Л. Смелянский. Алгоритмы и алгоритмические языки. Конспект лекций. — М.: ВМиК МГУ, 1998.
- [92] Compaq Pascal Language Reference Manual. Order Number: AA — PWVSC-TK. Version 5.7. Jun 1999. — Houston, Texas.
- [93] Зорич В. А. Математический анализ. Том I. — М.: МЦНМО, 2001.
- [94] <http://dmoz.org/Computers/Programming/Languages/SETL/>
- [95] Стивен С. Скиена, Мигель А. Ревиллья. Олимпиадные задачи по программированию. Руководство по подготовке к соревнованиям. Пер. с исп. — М.: Кудиц-Образ, 2005. — 416 с.
- [96] John Tromp. Kolmogorov Complexity in Combinatory Logic. March 17, 2002.
- [97] J. T. Tromp. <http://homepages.cwi.nl/~tromp/cl/Lambda.lhs>, 2004.
- [98] J. T. Tromp. [www.cwi.nl/~tromp/cl/CL.ps](http://www.cwi.nl/~tromp/cl/CL.ps)

- [99] Левинская М. А. Инstrumentальные средства создания интеллектуальных обучающих систем с визуальным преобразованием, сопоставлением и вычислением формул. Дисс. . . к. ф.-м. н. Научный руководитель к. ф.-м. н., доц. Зайцев В. Е. — М.: 2003.
- [100] Hodges, Andrew. Alan Turing: the enigma / Andrew Hodges; foreword by Douglas Hofstadter. — New York: Simon and Schuster, 1983. ISBN 0-8027-7580-2.
- [101] Ф. Бауэр, Т. Гооз. Информатика. Задачи и решения. — М.: Мир, 1978.
- [102] Н. С. Бахвалов. Численные методы, Т. 1. — М.: Наука, 1973.
- [103] Семантика языков программирования. Сборник статей. Пер. с англ. под ред. В. М. Курочкина. М.: Мир, 1976.
- [104] А. Л. Ластовецкий. Вычислительная ошибка при моделировании ЛА на ЭВМ с плавающей запятой: Учебное пособие. — М.: 1988. —38 с., ил.
- [105] Нефедов В. Н., Осипова В. А. Курс дискретной математики: Учеб. пособие. — М.: 1992. — 264 с.: ил. ISBN 5-7035-0157-X.
- [106] The Wolfram 2,3 Turing Machine Research Prize  
<http://www.wolframsience.com/prizes/tm23/>
- [107] Босс В. Лекции по математике. Т.6 От Диофанта до Тьюринга: Учебное пособие. — М.: КомКнига, 2006. — 208 с. ISBN 5-484-00463-2.
- [108] Petzold, Charles. The Annotated Turing: A Guided Tour through Alan Turing's Historic Paper on Computability and the Turing Machine. John Wiley&Sons, 2008. (ISBN 978-0-470-22905-7).
- [109] <http://lenta.ru/news/2007/10/24/math/>
- [199] Книги изд-ва Мир за 1971–95 год.

# Глава 1

## Основные понятия информатики

### Лекция 2

#### 1.1 Информация и сообщение

**Информация и сообщение** — основные (неопределяемые) понятия информатики (такие понятия есть и в математике, например, «точка», «множество»). Использование этих понятий можно разъяснить на примерах.

Информация передается посредством сообщения, и наоборот, сообщение — то, что несет информацию. Сообщение является материальным носителем информации. Более того, информация может существовать только в форме некоторого сообщения. В философии информация рассматривается также как одно из свойств объектов материального мира (такие как «масса», «энергия» и т. п.), так что любой материальный объект может рассматриваться как сообщение.

Соответствие между информацией и несущим ее сообщением не является взаимно однозначным: 1) одна и та же информация может передаваться с помощью различных сообщений (например, текущая лекция может быть прочитана на разных языках); 2) одно и то же сообщение может передавать совершенно различную информацию (тут можно сослаться на детективы: разведчик покупает в цветочном магазине одну белую и две красные розы; это видит другой разведчик, «случайно» прогуливающийся по улице недалеко от магазина и передает в центр перевод этого сообщения; еще пример — продавец *мясного* отдела гастронома кричит кассирше: «Перестань выбивать мозги!»). Продолжая эту тему, можно вспомнить знаменитые пирожки «с умом», продававшиеся на черноморском побережье Кавказа по 5 коп. В таких случаях всё зависит от **интерпретации сообщений**.

#### 1.2 Интерпретация сообщений

Информация  $i$ , которая передается сообщением  $n$ , устанавливается с помощью правила интерпретации, которое представляет собой отображение рассматриваемого множества сообщений  $\mathbf{v}$ <sup>1</sup> в множество сведений (термин «сведения» мы будем употреблять в качестве

---

<sup>1</sup>Отображение типа «в» обычно отождествляется с инъективным, «разные переходят в разные» [93].

множественного числа для термина «информация»):

$$\varphi: N \rightarrow I.$$

Множество сообщений  $N$  представляет собой язык (естественный или искусственный), т. е. каждое сообщение  $n$  является (правильно построенным) текстом языка  $N$ . Мы будем рассматривать **конечные языки**, содержащие конечное число текстов, и **бесконечные языки**, содержащие бесконечное множество правильно построенных текстов (имеется в виду потенциальная бесконечность).

Примеры конечных языков можно получить, рассматривая регулируемый перекресток со светофором с точки зрения водителя и с точки зрения регулировщика. В первом случае получается конечный язык, содержащий три текста: «красный», «желтый», «зеленый». Во втором случае получается другой конечный язык, число текстов в котором достигает 10–15.

Для того, чтобы задать конечный язык, можно перечислить все его тексты (см. врезку «язык Эллочки-людоедки»). Для задания бесконечного языка вводят правила построения текстов, или **грамматику** языка.

---

#### Пример конечного языка — язык Эллочки-людоедки [78]

1. Хамите.
2. Хо-хо! (Выражает, в зависимости от обстоятельств: иронию, удивление, восторг, ненависть, радость, презрение и удовлетворенность.)
3. Знаменито.
4. Мрачный. (По отношению ко всему. Например, «мрачный Петя пришел», «мрачная погода», «мрачный случай», «мрачный кот» и т. д.)
5. Мрак.
6. Жуть. (Жуткий. Например, при встрече с добrouй знакомой: «жуткая встреча».)
7. Парниша. (По отношению ко всем знакомым мужчинам, независимо от возраста и общественного положения.)
8. Не учите меня жить.
9. Как ребенка. («Я бью его как ребенка», — при игре в карты. «Я срезала его как ребенка», — как видно, в разговоре с ответственным съемщиком.)
10. Кр-р-расота!
11. Толстый и красивый. (Употребляется как характеристика одушевленных и неодушевленных предметов.)
12. Поедем на извозчике. (Говорится мужу.)
13. Поедем в таксо. (Знакомым мужского пола.)
14. У вас вся спина белая. (Шутка.)
15. Подумаешь.

16. Уля. (Ласкательное окончание имен. Например, Мишуля, Зинуля.)
17. Ого! (Ирония, удивление, восторг, ненависть, радость, презрение и удовлетворенность.)

Оставшиеся в небольшом количестве слова и выражения использовались как передаточные звенья между Эллочкой и приказчиками магазинов.

---

Отображение  $\varphi$  сопоставляет каждому (правильно построенному) тексту некоторого языка смысл или **семантику** этого текста. Для конечных языков  $\varphi$  можно задать в виде таблицы, в которой рядом с каждым текстом записан его смысл. Так, для примера со светофорами  $\varphi$  может быть задано с помощью следующей таблицы: «красный» — остановиться, «желтый» — внимание, «зеленый» — начать движение. Для бесконечных языков задать  $\varphi$  гораздо сложнее.

В процессе воспитания и образования человек усваивает свой родной язык и постепенно познает семантику таких текстов, как «я хочу пить», потом «дважды два равняется четыре», а потом и «рецессивная аллель влияет на фенотип только тогда, когда генотип гомозиготен». Иными словами, человек научается применять отображение  $\varphi$  для своего родного языка (обозначим это отображение через  $\varphi_0$ ), но, к сожалению, мы не имеем формального описания  $\varphi_0$  (мы не знаем, как человек интерпретирует тексты на родном языке). Семантику текста на иностранном языке (соответствующую интерпретацию обозначим через  $\varphi_1$ ) обычно выясняют путем перевода этого текста на родной язык и интерпретации перевода на родном языке. Иными словами, отображение  $\varphi_1: N_1 \rightarrow I$  представляется в виде **композиции** (последовательного применения) отображения перевода  $\pi: N_1 \rightarrow N_0$  и интерпретации  $\varphi_0: \varphi_1 = \pi \circ \varphi_0$  ( $\circ$  — знак композиции; композируемые отображения применяются слева направо). Отображение  $\pi$  тоже полностью не формализовано (перевод с иностранного языка — творческий акт), но для небольших подмножеств естественных языков отображение  $\pi$  может быть описано. Это справедливо, например, для разговорников (табличный способ задания  $\pi$ ), а также для подмножеств языка, определяемых небольшим числом предложений.

В информатике рассматриваются в основном искусственные языки (различные исчисления, формальные языки, алгоритмические языки, языки программирования и т. д.), либо специальные подмножества естественных языков (такие, например, как математический жаргон). Для интерпретации сообщений на таких языках можно использовать формальные модели. Мы рассмотрим одну из них немного позднее.

### 1.3 Знаки и символы

Каждое сообщение передается на каком-либо материальном **носителе сообщения**. Для устных сообщений носителем является воздух. Мы будем изучать сообщения, представленные на **долговременных** носителях (таких, как бумага, магнитная лента и т. п.). Такие сообщения называются **письменными**. Процесс нанесения письменных сообщений на носитель называется **записью** или **письмом**, а процесс их воспроизведения — **чтением**.

Письменные языковые сообщения представляют из себя последовательности знаков. Мы будем различать **атомарные** и **составные** знаки.

**Атомарным знаком** или **буквой** называется элемент какого-либо упорядоченного конечного непустого множества графически (или каким-либо иным образом) отличимых друг от друга литер (предметов, сущностей, объектов, членов), называемого **алфавитом**.

В качестве примеров атомарных знаков (букв) можно рассмотреть русскую (латинскую) букву А, арабскую цифру 5, знак сложения и т. п. В качестве экзотических можно представить себе алфавиты, элементы которых различимы не зритально, а с помощью иных органов чувств: по запаху, на ощупь или по весу). Понятие алфавита имеет важное практическое значение. Оно определяет множество допустимых знаков для записи входных и выходных сообщений на клавиатуре и принтере соответственно.

**Составным знаком** или **словом** называется конечная последовательность знаков (неважно, атомарных или составных). Множество слов тоже можно рассматривать как набор знаков (алфавит или *словарь*).

Таким образом, письменное сообщение может рассматриваться как один составной знак, либо как последовательность знаков (простых или составных). Если в состав последовательности знаков входят составные знаки, то необходимо иметь специальный знак, который помещается между соседними составными знаками, отделяя их друг от друга. Этот знак называется **разделителем**. Обычно в качестве разделителя используется знак пробела, который мы будем обозначать  $\lambda$  или пропуском (пустое знакоместо).

Атомарные знаки будем называть **знаками 1-го уровня**. Последовательности знаков 1-го уровня образуют знаки 2-го уровня (слова). Последовательности знаков второго уровня (в их составе разделители) — знаки третьего уровня и т. д. Для записи сообщения, которое содержит знаки  $k$ -того уровня необходимо иметь  $k - 1$  различных типов знаков-разделителей. В естественных языках текст целиком представляет собой знак достаточно высокого уровня. Так, суть письма запорожских казаков турецкому султану может быть выражена всего лишь одним знаком (кукиш). Для разделения слов (знаков второго уровня) используется пробел, для разделения предложений (знаков третьего уровня) используется точка, для разделения групп предложений (знаков четвертого уровня) — абзац и т. д.

С каждым знаком связывается его **смысл** или **семантика**. Знак (некоторого уровня) вместе с сопоставленной ему семантикой называется **символом** (это слово перегружено значением *знака, буквы, литеры*). Отметим, что разные знаки могут изображать один и тот же символ (знаки • и × в разных языках изображают один символ умножения) и что один и тот же знак может изображать разные символы ( $\lambda$  является символом для буквы греческого алфавита, символом пробела в теории алгоритмов, символом аргумента в  $\lambda$ -исчислении и т. д.). Таким образом, отношение между символом и знаком аналогично отношению между информацией и сообщением (знак — это сообщение символа).

## 1.4 Кодирование

**Кодом** называется правило, описывающее отображение  $C: A \rightarrow A'$  алфавита (набора знаков)  $A$  на другой набор знаков  $A'$ . Здесь «на» означает сюръекцию, отображение, при котором каждый элемент из  $A'$  имеет прообраз из  $A$ . Если при этом алфавит  $A'$  содержит меньше знаков, чем  $A$ , то образами некоторых знаков из  $A$  будут комбинации (конечные последовательности) знаков из  $A'$  (слова над алфавитом  $A'$ ), т. е. знаки более высокого уровня. Например, отображение восьмеричных цифр в двоичные триады дает двоичнокодированную запись восьмеричного числа ( $0_8 \rightarrow 000_2, 1_8 \rightarrow 001_2, \dots, 7_8 \rightarrow 111_2$ ).

В отличие от тайнописи, когда скрывается сам факт существования сообщения, при кодировании меняются лишь изображения букв (знаки). Смысл знаков сохраняется. Следовательно, при кодировании сообщения  $n$  получается новое сообщение  $C(n)$ , которое

содержит ту же информацию, что и  $n$ .

При этом изменяется правило интерпретации сообщений. Если до кодирования для интерпретации использовалось отображение  $\varphi_0$ , то после кодирования сообщения будут интерпретироваться с помощью отображения  $\varphi_1 = C^{-1} \circ \varphi_0$ , где  $C^{-1}$  — отображение, обратное отображению  $C$  (отображение **декодирования**). Следовательно, отображение  $C$  должно допускать существование обратного отображения, т. е. быть взаимно однозначным (биективным). Если известно отображение  $C^{-1}$  (или  $C$ ), или известен способ построения отображения  $C^{-1}$  (или  $C$ ), то говорят, что известен **ключ** кода  $C$ .

В качестве примера кодов рассмотрим семейство кодов Юлия Цезаря. В этих кодах знаки алфавита  $A$ , содержащего  $m$  знаков, заменяются другими знаками того же алфавита по следующему правилу: выбирается постоянное целое число  $p \in [0, m]$  и  $k$ -му знаку алфавита  $A$  ставится в соответствие его  $(k + p)$ -й знак, если  $k + p \leq m$ , и  $(k + p - m)$ -й знак, если  $k + p > m$ . Если, например,  $A$  — русский алфавит и  $p = 3$ , то слово *информатика* будет закодировано словом *лрчсупххнг*. Легко видеть, что ключом для кода Цезаря является число  $p$ , определяющее величину сдвига алфавита  $A$  при кодировании. Другими примерами являются азбука Морзе и различные кодировки текстовых данных в ЭВМ (КОИ-8, ASCII и т. п.).

Ниже приводится таблица семибитного кода ASCII в том виде, в котором он был принят в 1966 г.:

0 – NUL	16 – DLE	32 – SP	48 – 0	64 – ©	80 – P	96 – ’	112 – p
1 – SOX	17 – DC1	33 – !	49 – 1	65 – A	81 – Q	97 – a	113 – q
2 – STX	18 – DC2	34 – ‘	50 – 2	66 – B	82 – R	98 – b	114 – r
3 – ETX	19 – DC3	35 – #	51 – 3	67 – C	83 – S	99 – c	115 – s
4 – EOT	20 – DC4	36 – \$	52 – 4	68 – D	84 – T	100 – d	116 – t
5 – ENQ	21 – NAK	37 – %	53 – 5	69 – E	85 – U	101 – e	117 – u
6 – ACK	22 – SYN	38 – &	54 – 6	70 – F	86 – V	102 – f	118 – v
7 – BEL	23 – ETB	39 – ’	55 – 7	71 – G	87 – W	103 – g	119 – w
8 – BS	24 – CAN	40 – (	56 – 8	72 – H	88 – X	104 – h	120 – x
9 – HT	25 – EM	41 – )	57 – 9	73 – I	89 – Y	105 – i	121 – y
10 – LF	26 – SUB	42 – *	58 – :	74 – J	90 – Z	106 – j	122 – z
11 – VT	27 – ESC	43 – +	59 – ;	75 – K	91 – [	107 – k	123 – {
12 – FF	28 – FS	44 – ,	60 – <	76 – L	92 – \	108 – l	124 –
13 – CR	29 – GS	45 – -	61 – =	77 – M	93 – ]	109 – m	125 – }
14 – SO	30 – RS	46 – .	62 – >	78 – N	94 – ^	110 – n	126 – ~
15 – SI	31 – US	47 – /	63 – ?	79 – O	95 – _	111 – o	127 – DEL

Здесь нет места для национального алфавита. В свое время русские буквы были размещены вместо малых латинских (код КОИ-7). Соответствующим образом были переделаны знакогенераторы терминалов и принтеров. Помимо знаков внешнего алфавита в начале кода содержатся служебные кодовые комбинации для управления аппаратурой передачи и отображения данных. Эти знаки, как правило, не отображаются на устройствах ввода/вывода (терминалах, принтерах, клавиатурах и т. п.), но аппаратно интерпретируются ими. Структура знаковой части кода ASCII проста: вслед за пробелом (коды  $40_8, 32_{10}, 20_{16}$ ) идут 15 разделителей в порядке, который легко не запоминается. С 48-й позиции ( $60_8$  или  $30_{16}$ ) идут десятичные цифры в порядке возрастания. Далее после 7

разделителей начиная с кодов  $101_8$ ,  $65_{10}$  и  $41_{16}$  непрерывно в алфавитном порядке следуют 26 заглавных латинских букв. Через 32 позиции ( $141_8$ ,  $97_{10}$  или  $61_{16}$ ) размещается диапазон малых латинских букв. Таким образом, структура кода ASCII вполне регулярна и легко запоминаема. Все распространенные кодировки содержат ASCII в качестве подмножества.

Основная и альтернативная кодировки и кодировка КОИ-8 отличаются от ASCII способами добавления русских букв в расширенную часть таблицы (8-й бит равен 1) и размещением знаков псевдографики. В КОИ-8 русские буквы размещены фонетически (транскрипционно) так же, как и латинские в младшей части таблицы. Гашение старшего бита (происходящее, например, при посылке русских сообщений через многие западные почтовые сервера) дает произношение нижегородского по-французски (что иногда называют volapük или "как передать русскую телеграмму латиницей"): вместо АБВГД вы получите ABVGD. В основной кодировке русские буквы размещены подряд и без разрывов. В альтернативной кодировке заглавные буквы размещены подряд, а малые — разбиты на два поддиапазона, между которыми для совместимости с кодировкой IBM размещена псевдографика.

Первым восьмибитным кодом был IBM-овский EBCDIC. Его русская версия называлась ДКОИ.

Восьмибитные коды удобнее семибитных, там есть место и для больших, и для малых букв, и для одного национального алфавита. Но потребность в локализации (поддержки многоязычия) программного обеспечения и отсутствие простых способов представления текстовых строк привели к переходу на 16-битные кодировки, наиболее известной из которых является **Unicode**.

Unicode разработан Apple и Xerox (!) в 1988 г. С 1991 г. для совершенствования и внедрения Unicode создан консорциум из ведущих компаний. На сайте [www.unicode.org](http://www.unicode.org) опубликован стандарт Unicode.

Unicode представляет собой 16-битную кодировку и позволяет кодировать 65536 знаков вместо 256 8-битных кодов:

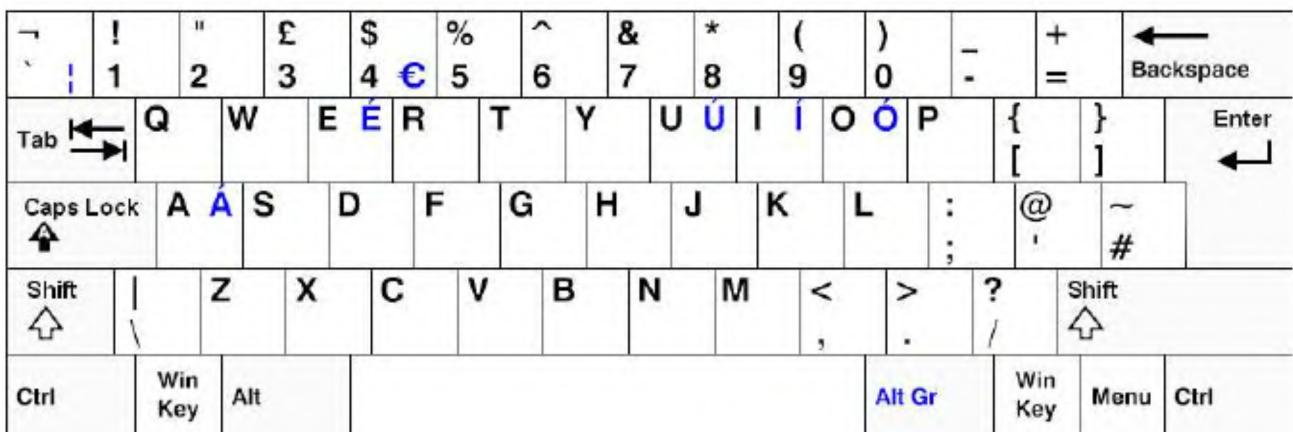
$$A_{101} \xrightarrow{\text{Unicode}} (A_2)^{16}$$

Здесь  $A_{101}$  — входной алфавит (например, определяемый клавиатурой), а  $A_2$  — выходной алфавит (двоичный, машинный).

Структура Unicode такова:

16-бит. код	Символы	16-бит. код	Символы
0000–007F	ASCII	0300–036F	Диакритические
0080–00FF	Символы Latin1	0400–04FF	Кириллица
0100–017F	Европейские латинские	0530–058F	Армянский
0180–01FF	Расширенные латинские	0590–05FF	Еврейский
0250–02AF	Стандартные фонетические	0600–06FF	Арабский
02B0–02FF	Модифицированные буквы	0900–097F	Деванагари

Около 29000 кодовых позиций Unicode не заняты. 6000 — зарезервированы для использования программистами. На Unicode построены ОС MS Windows 2000/XP/2003/CE. Unicode решает проблему национальных алфавитов ценой удвоенного расхода памяти, что по нынешним временам не так страшно. Кроме того, Unicode позволяет единообразно представлять строки, допуская смешение алфавитов.



Такое устройство ввода, как клавиатура, определяет базовый входной алфавит для представления вводимых сообщений. На самом деле клавиатурная раскладка — это не только размещение букв по клавишам (кнопкам) клавиатуры, а ещё и сопоставление этим буквам аппаратных сканкодов, т. е. своеобразное кодирование, преобразование алфавитов. А сканкоды отображаются в целевую часть используемой кодировки программой-драйвером клавиатуры. (Сканкод представляет собой 16-битный служебный клавиатурный код, генерируемый микроконтроллером клавиатуры по нажатию (отжатию) всех клавиш, в том числе тех, которых нет в ASCII. Сканкоды, вообще говоря, отличаются от кодов ASCII даже для знаков этого кода).

Стандартные раскладки клавиатур вовсе не случайны: они имеют частотно-рычажное происхождение. Будучи всё-таки не десяти-, а двухпальцевыми, они распределяют клавиши по убыванию известной частоты встречаемости букв данного языка от середины к краям клавиатуры, от развитого указательного пальца до периферийного мизинца. Рычажность раскладок заключалась в том, что при быстрой печати рычаги механической пишущей машинки раскачивались и задевали друг друга, сбивая припаянные буквы. Для предотвращения этого «частые» буквы перемежали с более редкими с учётом паросочетаемости. Рычагов давно уж нет, а раскладка осталась. Бывают раскладки не qwerty (немецкая, французская).

Кириллические раскладки имеют в своей основе другие частотные распределения букв (QWERTY → ЙЦУКЕН) и могут быть нанесены на латинскую клавиатуру верхним регистром. Другой способ сделать это — так называемая фонетическая раскладка — навеяна кодировкой КОИ-8: русские буквы нанесены на клавиши однотипных по произношению или написанию латинских (за исключением трёх букв).

Наиболее совершенной является раскладка Дворака. Именно на ней установлен мировой рекорд скорости печати на клавиатуре. Причиной быстроты печати по Двораку является не только частотный принцип размещения, но и чередование рук. Была даже предпринята попытка разместить цифры не по порядку, а в соответствии с частотами: 7, 5, 3, 1, 9, 0, 2, 4, 6, 8! В результате набор 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 превращается в играемый двумя руками пассаж.

Усовершенствованные раскладки продолжают появляться до сих пор: в 2006 г. предложена новая раскладка для кириллицы — Diktor. Эта раскладка рассчитана на слепое десятипальцевое мягкое печатание именно на компьютерной клавиатуре и свободна от рычажных атавизмов.

В заключение примеры: украинская и казахская раскладки.

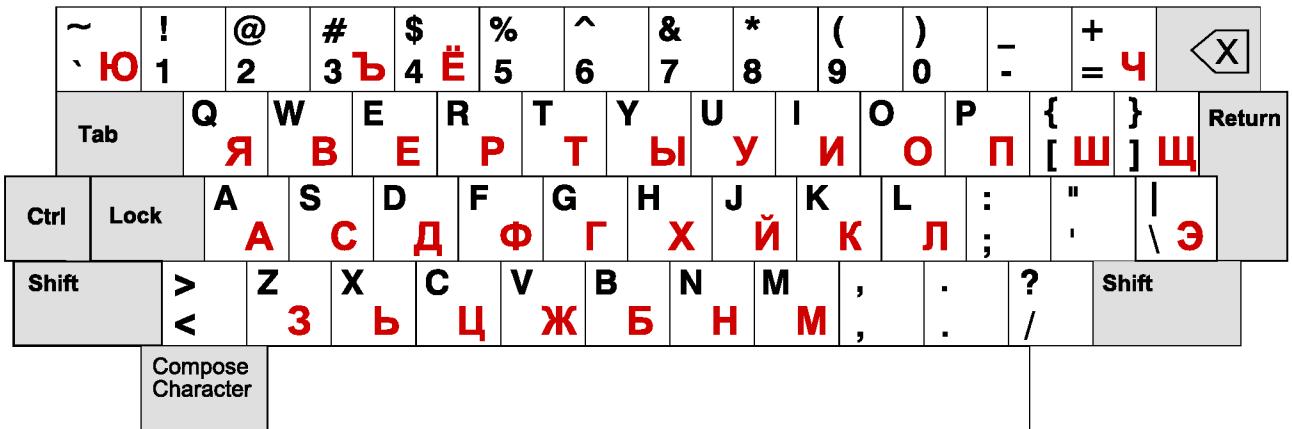


Рис. 1.1: Фонетическая раскладка

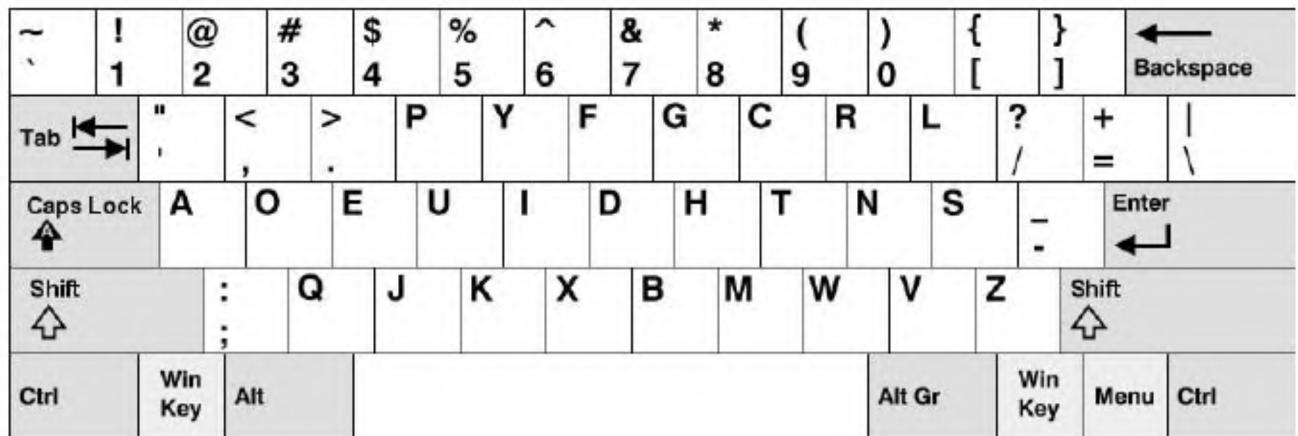


Рис. 1.2: Раскладка Дворака, [https://ru.wikipedia.org/wiki/Клавиатура\\_Дворака](https://ru.wikipedia.org/wiki/Клавиатура_Дворака)

Продолжим рассмотрение различных аспектов кодирования. Кодирование, при котором каждый *образ* является отдельным знаком, называется **шифрованием**. Традиционный шифр — это книга, в которой словам естественного языка сопоставлены группы цифр или букв. Видоизменяя сообщение, шифровка скрывает от непосвященного его смысл.

В рассмотренных ранее (симметричных) системах шифрования операции кодирования и декодирования являются взаимно обратными функциями, т. е. зная способ шифрования можно, действуя наоборот, расшифровать сообщение. Современная теория чисел дает способ создать систему шифрования, в которой используются два ключа — один для кодирования, другой для декодирования (т. н. **ассимметричное шифрование**). Система построена таким образом, чтобы, зная шифрованное сообщение и способ кодирования, было невозможно расшифровать сообщение за приемлемое время (подробнее см. [77]). Современная криптография основана на гипотезе об отсутствии каких-либо закономерностей в распределении простых чисел. В 2004 г. эта гипотеза была опровергнута, опубликовано доказательство, поставившее под сомнение всю просточисленную криптографию.

**Одностороннее шифрование** используется для хранения паролей (например, в Unix), и строго говоря кодированием не является. В этом случае используется неоднозначная (например, периодическая) функция. Для каждого шифрованного пароля получается множество вариантов дешифрованных, что затрудняет проникновение в систему.

После того, как кодирование и шифрование достаточно проиллюстрированы, можно избавиться от ненужного разнообразия алфавитов. Иногда оказывается удобным кодировать знаки рассматриваемого алфавита  $A$  в начальном отрезке счетного множества стандартных знаков  $\omega = b_1, b_2, \dots, b_m, \dots$ . При этом алфавиту  $A$ , содержащему  $m$  знаков, ставится в соответствие стандартный  $m$ -значный алфавит  $\omega = b_1, b_2, \dots, b_m$ , так, что первая буква алфавита  $A$  кодируется стандартным знаком  $b_1$ , вторая буква — знаком  $b_2$ , и т. д. Если, например,  $A$  — русский алфавит, то ему соответствует стандартный алфавит  $\omega$ , в котором слово **информатика** записывается в виде  $b_{10}b_{15}b_{22}b_{16}b_{18}b_{14}b_1b_{20}b_{10}b_{12}b_1$ .

Наряду со стандартным набором знаков  $\omega$  мы будем использовать расширенные стандартные наборы знаков  $\omega_0, \omega_{-1}, \dots$ , которые получаются добавлением к набору  $\omega$  несобственных знаков  $b_0, b_{-1}, \dots$ , которые используются для формирования разделителей, в частности,  $\omega_0 = \omega + b_0$ . Несобственный знак  $b_0$  является обозначением для пробела (' ',  $\lambda$ ).

**(1.4.1) Утверждение.** Произвольный конечный набор знаков (алфавит) может быть закодирован знаками набора  $\omega = (b_0, b_1)$ .

►Укажем способ кодирования. Знаки исходного набора могут быть закодированы знаками стандартного алфавита  $\omega_m$ , где  $m$  — число знаков в исходном наборе. Пусть  $b_i \in \omega_m$ . Сопоставим ему код  $b_0b_1\dots b_1$  ( $i+1$  раз  $b_1$ ) (последний код мы будем для краткости обозначать  $b_0b_1^{i+1}$ ). Утверждение доказано. □

Иногда для большей наглядности будут использоваться другие обозначения знаков набора  $\omega$ :  $b_0$  будет обозначаться знаком  $\lambda$ , а  $b_1$  — знаком | («палочка»).

## Лекция 3

### 1.5 Системы счисления

**Система счисления** — это способ числовой интерпретации цифровых сообщений. При этом удобно каждое число выражать отдельным словом (не используя многословных

бухгалтерских устных «сумм прописью» типа *Одна тысяча девятьсот восемьдесят девять*.

Рассмотренный в утверждении 1.4.1 способ кодирования может быть положен в основу т. н. **натуральной системы счисления**. В этом случае элементам бесконечного (счетного) множества  $\mathbb{N}$  натуральных чисел ставится в соответствие слова в  $\omega^*$  (звездочка над алфавитом обозначает множество всевозможных слов, образованных из его букв, включая пустое слово), при этом числу  $n$  сопоставляется слово из  $n$  палочек. Эта система называется натуральной, поскольку она представляет собой естественную числовую абстракцию, возникшую от пересчёта предметов — категории нуля и отрицательных чисел отсутствуют. Как установили биологи МГУ, натуральной системой счисления в пределах 20 владеют даже вороны. Кроме того, они могут складывать и вычитать числа порядка 3–4.

Знаменитый коллектив французских математиков Бурбаки склонен, для большей общности, не только приписывать 0 к натуральным числам, но и считать его одновременно и положительным, и отрицательным. Во многих случаях действительно полезно и удобно наряду с натуральными числами рассматривать и нуль. Для представления нуля можно слегка изменить способ кодирования — использовать палочку для обозначения нуля, две палочки — для единицы и т. д. Такую систему счисления будем называть **кардиальной**. В этом случае арифметические операции несколько усложняются: например, для сложения необходимо не только объединить две кучки палочек, но и изъять одну лишнюю для правильного представления суммы.

Недостатком этих замечательных по своей простоте систем счисления является линейный рост длины числового слова пропорционально величине представляемого числа. Так для изображения десятичного числа 1000 потребуется половина экрана терминала! То есть линейный рост в данном случае просто катастрофичен. Этот недостаток исправляется в т. н. позиционных системах счисления.

**Позиционная система счисления** — это полиномиальный способ числовой интерпретации слов над цифровым алфавитом. Коэффициентами многочлена являются малые целые числа, означающие уже априори (вручную) проинтерпретированные цифры. Значения этого многочлена вычисляются в точке, равной основанию системы счисления. Позиционная система задается одним натуральным числом  $p \in \mathbb{N}$  — **основанием системы счисления** — которое однозначно определяет алфавит  $A_p = \{0, 1, \dots, p - 1\}$  и функцию интерпретации

$$x = \overline{a_n a_{n-1} \dots a_1 a_0} \xrightarrow{\varphi_p} \sum_{i=0}^n \varphi(a_i)p^i = \sum_{j=0}^m \varphi(b_j)q^j \xleftarrow{\varphi_q} \overline{b_m b_{m-1} \dots b_1 b_0} = x$$

Интерпретация чисел опирается на натуральную интерпретацию цифр и основания системы счисления  $\varphi(a_i) = \underbrace{\| \dots \|}_{i \text{ палочек}}$ .

Если в эту формулу подставить  $p = 1$ , то получится единичная система счисления — аналог натуральной. Натуральные системы счисления принято считать непозиционными, хотя по некоторым соображениям их можно одновременно считать позиционными. Действительно, натуральная система счисления подходит под определение для  $p = 1$  и каждая дополнительная палочка слева увеличивает значение числа на 1 сообразно единичному основанию системы счисления.

Если основание системы счисления больше 1, то длина слова, изображающего число, равна целой части логарифма этого числа по основанию системы счисления  $p$ , увеличенной

на 1. Таким образом, линейный рост длины кодового слова сменяется гораздо более пологим логарифмическим и десятичное число 1000 изображается словом всего лишь из 4 цифробукв.

Из основополагающей формулы позиционной системы счисления следует способ перевода изображения числа в другую систему счисления: надо просто получить коэффициенты  $c_j$  разложения этого же числа по степеням другого основания  $q$ . Это будут остатки целочисленного деления числа  $x$  на основание целевой системы счисления  $q$ . Чтобы составить из них число в позиционной форме, надо всего лишь записать полученные остатки в обратном порядке.

Наиболее распространеными системами счисления являются системы с основанием 2, 3, 8, 10, 16. Системы счисления с нецелыми основаниями практически не используются, хотя наилучшей в некотором смысле является система счисления с основанием  $e$ . Это основание доставляет максимум функции

$$p^{\frac{n}{p}},$$

значения которой равны количеству кодовых слов длины  $n$  в  $p$ -ичном алфавите [75, 76, 80]. Для решения этой элементарной экстремальной задачи достаточно знаний основ математики из школьной программы.

Среди систем счисления с целыми основаниями ближе всего к  $e$  троичная. Она-то и использовалась в паре случаев для практической реализации идеала. (Троичная ЭВМ «Сетунь» и ее заокеанский собрат.) Среди рассматриваемых систем удобно выделить подкласс из двоичной, восьмеричной и шестнадцатиричной. Подставляя в основополагающую формулу вместо  $p$  2,  $2^3$  и  $2^4$  получим после приведения подобных членов двоичные триады и тетрады — кодовые слова (составные буквы), изображающие (кодирующие!) 8- и 16-ричные цифры. Это дает основание реализовать перевод между системами счисления с кратными основаниями как простое перекодирование изображений чисел. В остальных случаях это более трудоемкая арифметическая процедура.

Чем больше основание системы счисления, тем более экономна запись, и тем более сложная аппаратура требуется (клавиатура, кодировка, шрифт). Так, десятичная система счисления в  $\log_2 10$  — более чем в 3 раза экономнее (формула перехода к логарифму по другому основанию).

А как вам нравится система счисления с основанием 256? Удобно, конечно, каждую цифру представить в памяти ЭВМ одним байтом с непосредственной аппаратной адресацией. Однако при этом потребуется 347-кнопочная клавиатура ( $101 + 256 - 10$ ), которая так же удобна, как и 101-кнопочная мышь. Кстати, эта система счисления нашла применение в алгоритме Рабина-Карпа, используемом для быстрого поиска подстроки. А аналогичная идея прямого внешнего представления изображений чисел в ЭВМ была применена в так называемой десятичной арифметике, реализованной во многих универсальных ЭВМ 60-70-х годов; применение для этой цели BCD-кодировки давало экономию на переводе чисел в двоичную систему счисления.

---

Всем известная римская система счисления непозиционная, а точнее позиционная с нерегулярной и немонотонной позиционностью.

---

С позиционными системами счисления также связаны обратная и дополнительная кодировки отрицательных чисел в ЭВМ. Число  $-x$  кодируется дополнением до ближайшей

целой степени  $p$  с числом разрядов, на единицу большим количества цифр в разрядной сетке машинного слова:

$$p^n - x = 1 + \sum_{i=0}^{n-1} (p-1)p^i - \sum_{i=0}^{n-1} a_i p^i = 1 + \sum_{i=0}^{n-1} (p-1-a_i)p^i.$$

Диапазон представимых чисел несимметричен относительно 0:

$$-p^{n-1} \leq x \leq p^{n-1} - 1, \quad (*)$$

но представление 0 однозначно и арифметические операции не требуют изменения.

Таким образом, в дополнительном коде отрицательные числа представляются большими положительными, вдвое сужая диапазон допустимых положительных чисел. При этом, чем больше положительное значение кодового слова отрицательного числа, тем меньше абсолютная величина представляемого им числа. Например, в 64-битной ЭВМ DEC Alpha:

Цифрами обратного кода являются дополнения до максимальной цифры данной системы счисления  $p-1-a_i$ . Обратный код легко получается из прямого, но неудобен неоднозначностью представления 0 и требует изменения способа выполнения арифметических операций. Из формулы (\*) следует способ получения дополнительного кода:

1. Заменить цифры числа дополнениями до максимальной цифры системы счисления (простая и быстрая неарифметическая микрокодная операция параллельного кодирования над текстом изображения числа).
  2. Прибавить 1 к младшему разряду и выполнить возможно возникающие при этом переносы (эта операция кажется арифметической и сугубо последовательной, но наверное для нее существует аналог однотактного сумматора Бессонова — счетчика Шеннона [79]).

Элементарными операциями микрокода любой ЭВМ являются сдвиги: логические, циклические, арифметические. Логический сдвиг представляет собой перемещение всех разрядов машинного слова влево или вправо на число разрядов, заданное вторым операндом. При этом освобождающиеся при таком сдвиге свободные разряды заполняются

нулями, а разряды, выходящие за пределы машинного слова, теряются. При циклическом сдвиге разряды машинного слова считаются закольцованными, образуя т. н. двусторонний список, так что вслед за последним разрядом идет первый, а перед первым — последний. Арифметический сдвиг сложнее: он трактует слово как дополнительный код целого числа, так что сдвиг влево есть целочисленное умножение, а сдвиг вправо — деление. В первом случае «знаковый» разряд распространяется вправо по сдвигаемому слову, а младшие разряды теряются, как и положено при делении нацело. При сдвиге вправо «знаковый» разряд остается неизменным, а подлежащие удалению старшие разряды выпадают непосредственно перед ним. Это свойство сдвига, использующее позиционность системы счисления, позволяет очень просто реализовать довольно сложные операции умножения и деления, причем неарифметическим микрокодным образом.

Знаете ли вы, что 18 марта 2005 г. в 01:58:31 по Гринвичу системные часы UNIX, ведущие свой отсчёт с 0 часов 0 минут 0 секунд 1 января 1970 г. приняли значение  $1111111111_{10}$ .

А в ночь на 19 января 2038 г., когда будет достигнуто значение  $2^{31} - 1 = 2147483647$ , они исчерпают свой ресурс, возникнет переполнение (ошибка Y2038) и, если отсчитать назад полученное  $-2^{31}$ , настанет вечер 13 декабря 1901 г. (PC WEEK).

Представление вещественных чисел в ЭВМ возможно только конечными рациональными приближениями — дробями в позиционной системе счисления с основанием  $p$  (если не рассматривать экзотических случаев систем счисления с трансцендентными основаниями). Так же, как и в случае кодирования знака целого числа, позиционную точку в кодовом машинном слове не ставят, а подразумевают на некотором фиксированном месте. Однако такое разбиение приводит к диапазону представимых чисел, неудовлетворительному с практической точки зрения. Ни массу электрона, ни скорость света в обычном машинном слове представить с фиксированной точкой невозможно. Числа с фиксированной точкой применялись в прошлом только для простых действий с текстовыми или десятично-кодированными изображениями в бухгалтерских программах, когда переводить числа во внутреннее представление невыгодно [1]. (Десятично-кодированное представление чисел (BCD) вдвое экономнее строкового ASCII-шного т. к. каждая цифра занимает не байт, а полубайт. Более того, в крайнюю левую двоичную тетраду, представляющую старшую десятичную цифру числа, можно втиснуть и знак, поскольку не все 16 комбинаций полубайта нужны для представления десяти десятичных цифр.)

Для расширения диапазона представимых чисел немецкий ученый Конрад Цузе еще в 1937 г. предложил вместо фиксированной точки использовать полулогарифмическое композитное представление с *плавающей точкой*:

$$x = m \times p^e, e \in \mathbb{Z}.$$

В современных ЭВМ в качестве основания порядка  $p$  употребляют числа 2, 8 или 16. Полулогарифмичность этого косвенного представления состоит в том, что вместо числа  $x$  в одном машинном слове хранится и мантисса  $m$  — целое число, — и порядок (целая часть логарифма значения по выбранному основанию плюс единица; дробная часть логарифма находится в мантиссе) — тоже целое число. При этом мантисса представляется в прямом коде с явным заданием знака (здесь дополнительный код ничего не дает!). Позиционная точка в мантиссе подразумевается перед первым разрядом. Целое число мантиссы интерпретируется по стандартной формуле для позиционной системы счисления,

но, вообще говоря, с другим основанием системы счисления ( $q$ ). Порядок представляется беззнаковым целым числом  $e$ , смещённым на величину максимального по модулю отрицательного порядка. Такой модифицированный порядок называют *характеристикой*. Поскольку больший порядок дает более широкий диапазон, основание системы счисления для порядка  $p$  иногда делают большим, чем основание системы счисления для мантиссы  $q$ . Точность полулогарифмического представления определяется величиной  $q^{-n}$ , где  $n$  — число разрядов под мантиссой. Эта величина является мерой точности машинного вещественного типа и обозначается *машинным*  $\varepsilon$ . На каждом компьютере вы можете экспериментально определить машинное  $\varepsilon$  с помощью такой программы:

<pre><b>var</b> eps : <b>real</b>; eps := 1.0; <b>while</b> ((1.0 + eps / 2.0) &gt; 1.0) <b>do</b>     eps := eps / 2.0;</pre>	<pre><b>float</b> eps = 1.0; <b>for</b>(; 1.0 + eps / 2.0 &gt; 1.0; eps /=     2.0);</pre>
--	--

Подставив в этот цикл счётчик итераций, можно узнать число разрядов мантиссы.

## 1.6 Обработка сообщений

Обработка сообщений определяется правилом обработки  $\nu$ , которое представляет собой отображение  $\nu: N \rightarrow N'$ , где  $N$  — множество исходных сообщений, а  $N'$  — множество сообщений, получающихся в результате обработки.

Примером обработки сообщений является кодирование (см. п. 1.4). Другие примеры обработки сообщений: перевод с одного языка на другой, чтение вслух, стенографирование, перепечатка текста на машинке, редактирование текста, решение системы уравнений и т. д.

Приведенные примеры показывают, что каждая обработка сообщений состоит в выделении в исходном сообщении знаков некоторого уровня, составляющих это сообщение, и замене каждого выделенного знака другим знаком. При чтении вслух в исходном сообщении выделяются слоги (знаки второго уровня) и каждый слог заменяется фонемой (знаком первого уровня); последовательность фонем составляет результирующее устное сообщение. При решении системы уравнений вся система рассматривается как знак некоторого уровня и заменяется другим знаком (тоже достаточно высокого уровня) — решением системы.

Таким образом, любая обработка сообщений может рассматриваться как кодирование в широком смысле. Это соображение лежит в основе всякой машинной обработки *дискретных* сообщений. Заметим, что обработка сообщений производится человеком, но представляет собой рутинную нетворческую процедуру, ориентированную на технического исполнителя.

Обработка сообщений никогда не осуществляется «мгновенно», а всегда требует определенного времени, которым, как правило, нельзя пренебречь. Зависимость от времени приводит к понятию **эффективности правила обработки** сообщений, которая измеряется скоростью процесса обработки по сравнению со скоростями других процессов.

## 1.7 Обработка информации

Множество сообщений  $N$  представляет интерес только тогда, когда ему соответствует (по крайней мере одно) множество сведений  $I$  и определено соответствующее правило интерпретации  $\varphi: N \rightarrow I$  (см. п. 1.2). Так как множеству сообщений  $N'$  тоже соответствует некоторое множество сведений  $I'$  (и правило интерпретации  $\varphi'$ ), то любое правило обработки сообщений  $\nu: N \rightarrow N'$  (см. п. 1.6) приводит к следующей диаграмме:

$$\begin{array}{ccc} N & \xrightarrow{\varphi} & I \\ \nu \downarrow & & \downarrow \sigma \\ N' & \xrightarrow{\varphi'} & I' \end{array} \quad (*)$$

Эта диаграмма определяет соответствие между множествами  $I$  и  $I'$ . Так как согласно диаграмме  $(*)$  каждому сообщению  $n \in N$  соответствует пара сведений  $i = \varphi(n) \in I$  и  $i' = \varphi(\nu(n)) \in I'$ , построенное соответствие между  $I$  и  $I'$  (обозначим его через  $\sigma$ ), вообще говоря, не является отображением. В самом деле, если правило интерпретации  $\varphi$  не является однозначным (инъективным, когда разные переходят в разные), т. е. если существуют два различных сообщения  $n_1, n_2 \in N$ ,  $n_1 \neq n_2$ , передающих одинаковую информацию  $i = \varphi(n_1) = \varphi(n_2)$ , то может оказаться, что  $\varphi'(\nu(n_1)) \neq \varphi'(\nu(n_2))$  и, следовательно, одной информации  $i \in I$  будут соответствовать (по крайней мере) две различных информации  $i'_1 = \varphi'(\nu(n_1))$  и  $i'_2 = \varphi'(\nu(n_2))$ .

Если отображение  $\varphi$  обратимо, т. е. если существует отображение  $\varphi^{-1}$  (для нас достаточно, чтобы  $\varphi$  было инъективным отображением), то можно построить отображение  $\sigma$ , определяющее обработку информации  $\sigma: I \rightarrow I'$  в виде  $\sigma = \varphi' \circ \nu \circ \varphi^{-1}$  так, что  $i' = \varphi'(\nu(\varphi^{-1}(i)))$ .

Во всех случаях, когда соответствие  $\sigma$  является отображением, правило обработки сообщений  $\nu$  называется **сохраняющим информацию**. Если правило обработки сообщений  $\nu$  сохраняет информацию, то диаграмма

$$\begin{array}{ccc} N & \xrightarrow{\varphi} & I \\ \nu \downarrow & & \downarrow \sigma \\ N' & \xrightarrow{\varphi'} & I' \end{array} \quad (**)$$

коммутативна:  $\nu \circ \varphi' = \varphi \circ \sigma$ . Отображение  $\sigma$  называется в этом случае **правилом обработки информации**.

Обычно обработку информации сводят к обработке сообщений, т. е., исходя из требуемого правила обработки информации  $\sigma$ , пытаются определить отображения  $\nu$ ,  $\varphi$  и  $\varphi'$  таким образом, чтобы диаграмма  $(**)$  была коммутативной.

Если  $\sigma$  — обратимое (взаимно однозначное) отображение, т. е. если информация при обработке по правилу  $\sigma$  не теряется, то соответствующую обработку сообщений  $\nu$  называют **перешифровкой**.

Пусть  $\nu$  — обратимая пересыпка. Тогда по сообщению  $n' = \nu(n)$  можно восстановить не только исходную информацию, но и само исходное сообщение  $n$ . Иными словами, в этом случае  $n'$  кодирует  $n$  (см. п. 1.4). Обратимая пересыпка  $\nu$  называется **перекодировкой**.

Пусть пересыпка  $\nu$  не является обратимой, т. е. пусть несколько сообщений из  $N$  кодируются одним и тем же сообщением из  $N'$ . Но так как при пересыпке информация

не теряется, это означает, что исходное множество сообщений  $N$  является избыточным: некоторые сообщения из  $N$  содержат одну и ту же информацию (дублируют друг друга). В  $N'$  таких дублирующих сообщений меньше, чем в  $N$ , так как при обработке по правилу  $\nu$  некоторые из дублирующих друг друга сообщений «сливаются» в одно сообщение. Перешифровка  $\nu$ , которая не является обратимой, называется **сжимающей**. Сжатию подвергается множество сообщений. То есть в результате необратимой перешифровки сообщений их количество уменьшается, а информация может либо сохраняться, либо теряться.

**Пример 1.7.1.** Пусть сообщения  $(a, b)$ , составленные из пар целых чисел (например, в десятичной позиционной записи), передают информацию «рациональное число  $r$ , представленное дробью  $\frac{a}{b}$ ». Тогда  $N = \mathbb{Z} \times \mathbb{N}$  (где  $\mathbb{Z}$  — множество целых чисел,  $\mathbb{N}$  — множество натуральных чисел),  $I = \mathbb{Q}$  ( $\mathbb{Q}$  — множество рациональных чисел). Отображение  $\varphi: N \rightarrow I$  не является обратимым, так как при любом целом  $n$  парам  $(a, b)$  и  $(na, nb)$  соответствует одно и то же рациональное число  $r$ . Пусть  $N'$  — множество пар  $(p, q)$  взаимно простых целых чисел и пусть  $\nu: N \rightarrow N'$  переводит все  $(np, nq)$  в  $(p, q)$ . Тогда  $\nu$  — сжимающее отображение, а  $\varphi': N' \rightarrow I$  — обратимое отображение (мы считаем  $I' = I$ ). Такое отображение  $\nu$  называется **вполне сжимающей перешифровкой**, поскольку после обработки сообщений соответствие между сообщениями и информацией биективно. Здесь информация не теряется.

Если  $\sigma$  — необратимое отображение, т. е. если разные сведения из  $I$  отображаются в одну и ту же информацию  $i' \in I'$ , то соответствующую обработку сообщений называют **избирательной**. Особенno часто встречается случай, когда  $I' \subset I$ , а  $\sigma$  — тождественное отображение для всех  $i' \in I'$ . В этом случае производится *выбор* из данного множества сведений.

Таким образом, «обработка информации» — это, как правило, сокращение количества информации. Во всяком случае, верно утверждение: обработка информации никогда **не добавляет** информацию, она состоит в том, что **извлекает** интересную информацию из той, которая содержится в сообщении.

## Лекция 4

### 1.8 Автоматизация обработки информации

Вернемся к рассмотрению диаграммы (\*\*). Если заменить на ней отображение  $\varphi$  обратным отображением  $\psi = \varphi^{-1}$ , получим новую диаграмму:

$$\begin{array}{ccc} N & \xleftarrow{\psi} & I \\ \nu \downarrow & & \downarrow \sigma \\ N' & \xrightarrow{\varphi'} & I' \end{array}$$

Автоматизация обработки информации заключается в выполнении  $\sigma$  или  $\varphi^{-1} \circ \nu \circ \varphi'$  при помощи физических устройств. Однако в программировании изучаются методы автоматического выполнения только отображения  $\nu$ , т. е. обработки сообщений. Программно-аппаратная реализация отображений  $\psi, \varphi'$  изучается в другом разделе информатики, который называется «Искусственный интеллект» — и потому выходят за рамки нашего курса.

Для автоматизации обработки сообщений необходимо иметь физическое устройство, работа которого «моделировала» бы выполнение отображения  $\nu$ . Для этого необходимо располагать тремя **физическими представлениями**:

1. физическим представлением для множества исходных сообщений  $N$  — его мы будем обозначать буквой  $D$  и называть *множеством исходных данных*;
2. физическим представлением для множества результирующих сообщений  $N'$  — его мы будем обозначать буквой  $D'$  и называть *множеством результирующих данных* (результатов);
3. физическим представлением правила обработки  $\nu$ , которое должно быть преобразованием, или последовательностью преобразований, оперирующих над  $D$  и дающих в результате элемент  $D'$ , — его мы будем обозначать буквой  $P$ ; преобразование  $P$  должно быть физически реализуемым, т. е. выполняться на некотором физическом устройстве.

Таким образом, для того, чтобы выполнить автоматическую обработку сообщений, необходимо автоматизировать выполнение трех отображений:

1. отображения кодирования  $C$ , которое позволяет представить (абстрактное) сообщение из  $N$  с помощью элемента множества данных  $D$ , т. е., с помощью конкретного состояния некоторого физического устройства;
2. отображения  $P$ , которое переводит элемент  $D$  в элемент  $D'$  — конкретное состояние еще одного физического устройства;
3. отображения декодирования  $Q$ , которое позволяет проинтерпретировать результат — конкретное состояние физического устройства, представляющее элемент из  $D'$ , переведя его в соответствующее сообщение из множества  $N'$ .

Таким образом, автоматическая обработка информации может быть представлена диаграммой

$$\begin{array}{ccccc} D & \xleftarrow{C} & N & \xleftarrow{\psi} & I \\ P \downarrow & & \downarrow \nu & & \downarrow \sigma \\ D' & \xrightarrow{Q} & N' & \xrightarrow{\varphi'} & I' \end{array}$$

Из диаграммы следует  $\nu = C \circ P \circ Q$ , причем все три отображения  $C$ ,  $P$  и  $Q$  должны выполняться автоматом.

## 1.9 Конструктивное описание процесса обработки дискретных сообщений

Обработка (дискретных) сообщений состоит в применении отображения (правила обработки)  $\nu$  (см. п. 1.6), или в последовательном применении отображений  $C$ ,  $P$  и  $Q$  (см. п. 1.8).

Чтобы отображение  $P$  могло служить основой для автоматизации обработки сообщений, оно должно задавать некоторый **способ построения** сообщения  $d' = P(d)$ , исходя

из сообщения  $d \in D$ . Если  $D$  — конечное множество, то отображение  $P$  может быть задано таблицей, в которой перечисляются все сообщения  $d \in D$  и соответствующие им сообщения  $P(d)$ . Примером такой таблицы может служить таблица умножения (или таблица сложения) однозначных чисел в позиционной системе счисления. Если же  $D$  бесконечно или по крайней мере так велико, что задание  $P$  с помощью таблицы оказывается непрактичным, то нужно представить  $P$  в виде последовательности элементарных **шагов обработки** (или **тактов**), каждый из которых состоит в выполнении одного или нескольких достаточно простых отображений, называемых **операциями**:  $P = P_1 \circ P_2 \circ \dots \circ P_k$ . Примеры операций: переписывание (копирование) буквы или слова, приписывание буквы к слову, приписывание слова к другому слову. Отметим, что любое отображение, допускающее описание с помощью достаточно простой таблицы, может быть объявлено операцией. Конечно же, оператор языка программирования высокого уровня, машинная команда, директива или системный вызов операционной системы также удовлетворяют критерию элементарности шагов обработки.

**Пример 1.9.1.** Пусть исходное сообщение — это пара  $(u_1, u_2)$  слов над некоторым (например, латинским) алфавитом, к которому перед буквой  $a$  добавлен пробел, и пусть требуется, чтобы результирующее сообщение  $(w_1, w_2)$  состояло из тех же слов, но расположенных в лексикографическом порядке. Правило обработки может быть задано следующей последовательностью элементарных шагов:

- (1) завести пустые слова  $w_1$  и  $w_2$  (это можно сделать, например, написав  $w_1 = ''$  и  $w_2 = ''$ ), уравнять количество букв в словах  $u_1$  и  $u_2$ , добавив в конец более короткого слова пробелы и перейти к шагу (2);
- (2) сравнить (пользуясь списком букв в алфавитном порядке, или таблицей для отношения алфавитного предшествования  $\prec$ )  $\ell(u_1)$  и  $\ell(u_2)$  ( $\ell(u)$  — обозначение для крайней левой буквы слова  $u$ , отличной от пробела); если  $\ell(u_1) \prec \ell(u_2)$ , перейти к шагу (3); если  $\ell(u_2) \prec \ell(u_1)$ , перейти к шагу (4). Если буквы  $\ell(u_1)$  и  $\ell(u_2)$  совпадают и не являются пробелами, то приписать  $\ell(u_1)$  (или  $\ell(u_2)$ ) к словам  $w_1$  и  $w_2$ . Заменить  $\ell(u_1)$  и  $\ell(u_2)$  пробелами и снова выполнить шаг (2). Наконец, если  $\ell(u_1)$  и  $\ell(u_2)$  — пробелы, закончить обработку сообщения;
- (3) приписать слово  $u_1$  к слову  $w_1$ , слово  $u_2$  к слову  $w_2$  и закончить обработку сообщения;
- (4) приписать слово  $u_1$  к слову  $w_2$ , слово  $u_2$  к слову  $w_1$  и закончить обработку сообщения.

Нетрудно показать, что последовательность шагов (1)–(4) действительно обеспечивает лексикографическое упорядочивание слов в любом исходном сообщении.

Обработка сообщений, описанная в примере (1.9.1), называется **эффективной процедурой** или **алгоритмом**. Более точно, алгоритм — это точно заданная последовательность правил, указывающая, каким образом можно за конечное число шагов получить выходное сообщение определенного вида, используя заданное исходное сообщение.

Пример (1.9.1) позволяет обнаружить некоторые свойства алгоритмов, а именно:

- (1) **массовость**, т. е. потенциальная бесконечность исходных сообщений, предназначенных для переработки алгоритмом;

- (2) **детерминированность** процесса применения алгоритма, заключающегося в последовательном выполнении дискретных действий (шагов), однозначно определяемых результатами каждого предыдущего шага;
- (3) **элементарность каждого шага** обработки, который должен быть достаточно легко выполним;
- (4) **результативность** алгоритма, т. е. точное описание того, что считается результатом применения алгоритма после выполнения всех требуемых шагов.

Чрезвычайно важным с точки зрения практической эффективности алгоритмов является еще одно свойство:

- (5) **сложность** алгоритма определяется количеством простых операций, которые нужно совершить для обработки сообщения. Это количество прямо зависит от длины входного сообщения  $n$  и обычно задается некоторой функцией  $f(n)$ . Таким образом, сложность ограничивает применимость алгоритма. Алгоритм, выполняющийся за постоянное время, можно применять к любому входному сообщению. Если время выполнения пропорционально  $n^2$  или  $n^3$ , то такой алгоритм неприменим к длинным исходным сообщениям, поскольку время выполнения становится неприемлемо большим.

Иногда очевидный алгоритм можно существенно улучшить за счет другого подхода к проблеме. Пусть необходимо вычислить значение многочлена

$$a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$$

Прямое вычисление потребует  $n + n - 1 + n - 2 + \dots + 1 = \frac{n(n+1)}{2}$  операций умножения и  $n$  операций сложения, всего  $\frac{n(n+3)}{2}$ , т. е. его сложность —  $O(n^2)$ . А если переписать задачу в виде

$$(\dots((a_n x + a_{n-1}) x + a_{n-2}) \dots) x + a_0,$$

то можно обойтись  $n$  умножениями и  $n$  сложениями — всего лишь  $2n$  операциями, т. е. порядок сложности этого алгоритма, называемого *схемой Горнера*, —  $O(n)$ .

Сложность алгоритма определяет, как долго придется ждать, пока будет получен результат его работы. Одновременно сложность определяет размер задачи, которая может быть решена данным алгоритмом за приемлемое время. Если иногда можно подождать месяц или даже три, то ждать год или 10 лет почти всегда бессмысленно. Поэтому всегда актуальна задача о поиске наиболее эффективных (т. е. быстро работающих) алгоритмов.

Кроме обычной временной сложности алгоритмов часто рассматривают и пространственную сложность — объем памяти, необходимый алгоритму для переработки входного сообщения длины  $n$ .

Для обозначения класса сложности используется  $O$ -нотация [93, 76]. У всякого алгоритма, как у функции, есть главная часть, которая занимает больше всего времени для выполнения. Анализируя количество выполнений главной части делают вывод о поведении алгоритма в зависимости от длины исходного сообщения [63]. Тогда появляется

возможность сказать, какие задачи могут быть решены за приемлемое время, а какие не будут решены практически никогда.

Все алгоритмы можно разбить на классы по их сложности [88]:

Класс сложности	Описание
$O(1)$	Алгоритмы с постоянным временем выполнения, например доступ к элементу массива. При увеличении размера задачи вдвое время выполнения не меняется.
$O(\log n)$	Алгоритмы с логарифмическим временем выполнения, например бинарный поиск. Время выполнения удваивается при увеличении размера задачи в $n$ раз.
$O(n)$	Алгоритмы с линейным временем выполнения, например последовательный поиск или схема Горнера. При увеличении размера задачи вдвое время выполнения также удваивается.
$O(n \log n)$	Алгоритмы с линеарифмическим временем выполнения, например, быстрая сортировка. Время выполнения увеличивается немногим больше, чем вдвое, при увеличении размера задачи в два раза.
$O(n^2)$	Алгоритмы с квадратичным временем выполнения, например простые алгоритмы сортировки. Время выполнения увеличивается в 4 раза при увеличении размера задачи в 2 раза.
$O(n^3)$	Алгоритмы с кубическим временем выполнения, например умножение матриц. Время выполнения увеличивается в 8 раз при увеличении размера задачи в 2 раза.
$O(2^n)$	Алгоритмы с экспоненциальным временем выполнения, например задача о составлении расписания. Время выполнения увеличивается в $2^n$ раз при увеличении размера задачи в 2 раза.

И сегодня задача построения эффективных алгоритмов остается весьма актуальной. Заведующий кафедрой математической кибернетики факультета ВМиК МГУ профессор Алексеев В. Б. пишет:

---

«Для того, чтобы передать задачу компьютеру, казалось бы достаточно разработать какой-нибудь алгоритм решения. Но оказывается, что и при современных сверхскоростных компьютерах некоторые алгоритмы не подходят, поскольку требуют слишком много времени. Например, если алгоритм требует перебрать все булевые функции от  $n$  переменных, то компьютер не сможет это сделать даже за миллион лет уже при  $n = 7$ . Выходов два — либо ускорять компьютеры, либо придумывать новые нестандартные алгоритмы. Оказывается, что нестандартные алгоритмы существуют для многих задач. Например, умножение в столбик двух  $n$ -разрядных чисел требует  $O(n^2)$  битовых операций. Только в 1962–63 гг. были обнаружены более быстрые алгоритмы со сложностью  $O(n^{1+\varepsilon})$  для любого  $\varepsilon > 0$ . Они основаны на разложении чисел на множители (т. н. китайская теорема об остатках [64]). Оказывается, что и умножение матриц *строка на столбец* — это не самый быстрый способ (он требует  $O(n^3)$  арифметических операций для матриц порядка  $n$ ). Уже более 15 лет известна оценка  $O(n^{2.38})$ , но дальнейшее ее улучшение еще впереди. К сожалению, для большинства задач на практике не существует пока алгоритмов с полиномиальной верхней оценкой сложности, а экспоненциальная сложность — очень быстро растущая величина. С другой стороны, для этих алгоритмов нет и нижней оценки сложности выше, чем линейная. Так что с ситуацией здесь еще предстоит разбираться. Похоже, что для разработки быстрых алгоритмов надо применять хорошо развитые разделы математики, такие, например, как алгебра.»

---

Важность сложностных характеристик алгоритмов особенно велика для таких ответственных применений, как криптография. Защитой от расшифровки может быть высокая сложность алгоритма декодирования. Своебразной шифровке — обfuscации — подвергаются и программы, исходный код которых маскируется от реинженеринга. В последнем случае стойкость скрываемого кода определяется сложностью алгоритма распознавания исходного текста программы.

**Замечание.** Предлагая новый или модифицированный алгоритм, необходимо всегда давать его сложностную оценку, без которой невозможно судить о полезности этого алгоритма. Во всяком случае, алгоритм без такой оценки не считается научно обоснованным. Кроме того, во многих случаях, заказчик налагает весьма жёсткие ограничения на пространственную и временную сложность алгоритма и программы решения задачи. Например, на олимпиадах по программированию предельное время выполнения последовательности тестов жюри программой всегда задаётся таким, чтобы лобовой алгоритм заведомо не уложился бы. Например, степень полинома  $p$  выбирается настолько большой, чтобы без схемы Горнера с её линейной сложностью было невозможно за малое время его табулировать (вычислить значения такого многочлена во многих точках).

В нашем курсе понятие алгоритма и его свойства будут изучены достаточно подробно и достаточно строго. Это обусловлено тем, что алгоритмы образуют сердцевину информатики, являются тем общим знаменателем, который образует ее основу и объединяет различные направления. Здесь уместно повторить мнение Д. Кнута: «Лучший, с моей точки зрения, способ определить информатику — это сказать, что она занимается изучением алгоритмов» [34].

Но прежде чем заняться детальным изучением алгоритмов и их свойств, нам необходимо рассмотреть вопросы, связанные с интерпретацией сообщений, и, в частности, выяснить, как реализуются отображения  $\psi$  и  $\varphi'$ , связывающие сообщения с содержащейся в них информацией. Решив эти вопросы, мы сможем задавать правильные исходные данные для алгоритмов и правильно интерпретировать сообщения, полученные в результате их применения.

## Лекция 5

### 1.10 Интерпретация дискретных сообщений

Сообщения интерпретируются людьми в соответствии с их представлениями о внешнем мире, о среде (естественной или искусственной), информацию о которой содержат эти сообщения. Именно этим обстоятельством объясняется то, что одно и то же сообщение интерпретируется разными людьми по-разному. Прежде, чем заняться уточнением этих несколько туманных утверждений, поясним их смысл на примере.

**Притча.** Однажды седобородый философ и одноглазый вор вели на языке жестов беседу о смысле жизни. Философ (**Ф**) показал одноглазому (**О**) один палец; в ответ **О** показал два пальца; тогда **Ф** показал три пальца, а **О** ответил тем, что показал кулак; после этого **Ф** сделал жест, имитирующий процесс питья из большой чаши; **О** в ответ показал ему луковицу. На этом беседа завершилась и собеседники разошлись, очень довольные собой и друг другом. Один из любопытных свидетелей беседы обратился к философу за разъяснением ее содержания. «Мой собеседник оказался очень мудрым

человеком», — сказал философ. «Сначала я сказал ему, что плохо жить одному, как перст. На что он ответил, что, конечно, вдвоем жить лучше. Тогда я сказал ему, что еще лучше, когда в семье трое, имея в виду ребенка. Он согласился и заметил, что ребенок сплотит семью. И жизнь пойдет полной чашей, отметил я напоследок, а он, показав луковицу, как бы сказал, что тогда даже мелкие неприятности не будут омрачать радости жизни». Свидетель усомнился в интерпретации **Ф** и пошел за разъяснениями к одноглазому.

«Он, ничего, парень смелый, но куда ему против меня», — сказал **О**. «Сначала он говорит мне: Эй ты, одноглазый черт», на что я отвечал: «А ты со своими двумя не очень-то задавайся». А он никак не уимется: «А ведь на двоих-то у нас всего три глаза». Тут я показал ему кулак. Он испугался и решил мириться. «Выпьем, что ли?» — предложил он, на что я ответил: «Иди за бутылкой, закуска найдется». **Мораль** очевидна.

Чтобы не оказаться в положении собеседников из притчи, фиксируют представление о внешнем мире в виде его модели. Мы здесь опишем сущность этого понятия, не уточняя пока полностью определений. Формальное и строгое изложение данного материала отложим до специальных курсов.

Строя математическое понятие модели, мы исходим из того, что внешний мир (среда) состоит из объектов (предметов), которые обладают определенными свойствами (атрибутами) и находятся в определенных отношениях друг с другом. Объектам, их атрибутам и отношениям ставятся в соответствие знаки (имена), из которых строятся сообщения (напомним, что дискретное сообщение — это последовательность знаков). Интерпретация дискретного сообщения при наличии модели сводится к тому, что каждому знаку, входящему в состав сообщения, ставится в соответствие объект, свойство (атрибут) объекта, или отношение между объектами, представляющее этим знаком [23, 32].

**Определение 1.10.1.** Назовем  $k$ -местным отношением на множестве  $M$  совокупность (множество)  $r^k$  упорядоченных наборов из  $k$  элементов множества  $M$  вида  $\langle x_1, x_2, \dots, x_k \rangle$  (иначе говоря, отношение  $r$  — это подмножество декартова произведения  $r^k \subseteq M^k$ ; верхний индекс у  $r$  указывает местность (число мест) отношения).

**Определение 1.10.2. Моделью** (или реляционной системой) называется некоторое множество  $\mathcal{M}$  с заданным на нем набором отношений  $\{r_1^{k_1}, r_2^{k_2}, \dots, r_n^{k_n}\}$ . Иначе говоря, модель  $\mathcal{M}$  — это упорядоченная пара объектов  $\mathcal{M} = \langle M, \{r_1^{k_1}, r_2^{k_2}, \dots, r_n^{k_n}\} \rangle$ , где  $r_1^{k_1}, r_2^{k_2}, \dots, r_n^{k_n}$  — отношения на множестве  $M$ .

**Пример 1.10.3.** Модель  $\mathcal{M}_1 = \langle R, \text{„} < \text{“} \rangle$ , где  $R$  — множество вещественных чисел, „ $<$ “ — отношение «меньше». В математике и физике рассматривается специальный класс моделей, в которых на исходном множестве заданы некоторые числовые функции, а отношения выражаются через значения этих функций (см. Пример 1.10.4).

**Пример 1.10.4.** Модель  $\mathcal{M}_2 = \langle E, \{r_1^2, r_2^2, r_3^2\} \rangle$ . Пусть на множестве  $E$  задана вещественная неотрицательная функция двух переменных  $\rho(x, y)$ , такая, что  $\rho(x, y) = 0$  тогда и только тогда, когда  $x = y$ . По определению,  $x$  и  $y \in E$  находятся в отношении  $r_1^2$  тогда и только тогда, когда  $\rho(x, y) = 0$ ;  $x$  и  $y \in E$  находятся в отношении  $r_2^2$  тогда и только тогда, когда  $\rho(x, y) = \rho(y, x)$ ;  $x$  и  $y \in E$  находятся в отношении  $r_3^2$  тогда и только тогда, когда  $\rho(x, y) \leq \rho(x, z) + \rho(z, y)$ . Например, в качестве множества  $E$  можно взять евклидово пространство, а в качестве функции  $\rho(x, y)$  — расстояние между точками этого пространства.

В определении модели не указано, что же моделируется. Примеры моделей 1.10.3 и 1.10.4 также не дают ответа на этот вопрос: что моделируют модели  $\mathcal{M}_1$  и  $\mathcal{M}_2$ . Для решения этого вопроса введем понятия **теории** и **сигнатуры**.

Пусть  $R^k$  — имя (знак) отношения  $r^k$ . Будем говорить, что модель  $\mathcal{M} = \langle M, \{r_1^{k_1}, r_2^{k_2}, \dots, r_n^{k_n}\} \rangle$  имеет сигнатуру  $\Omega$ , если существует правило интерпретации  $\varphi$ , такое, что  $\varphi(R_i^k) = r_i^{k_i}$  для всех  $i = 1, 2, \dots, m$ . Если задана сигнатура  $\Omega$  и какая-нибудь интерпретация  $\varphi$  этой сигнатуры на множестве  $M$ , то пара  $\langle M, \varphi \rangle$  определяет модель (мы будем называть ее моделью в сигнатуре  $\Omega$ ).

Сигнатура — это перечень знаков (имен) отношений. **Теория** — это упорядоченная пара  $T = (\Omega, A)$ , где  $\Omega$  — сигнатуря, а  $A$  — множество **аксиом**, высказываний о свойствах сигнатуры  $\Omega$  (обычно в качестве аксиом используются формулы специального вида).

Модель  $\mathcal{M}$  называется моделью теории  $T$ , если  $\mathcal{M}$  и  $T$  имеют одинаковую сигнатуру и если после интерпретации  $\varphi$  каждого имени отношения теории, как одноименного отношения в модели, каждая аксиома теории становится истинным высказыванием.

Таким образом, теория — это перечень названий отношений и свойств этих отношений, а модель — это множество, на котором заданы соответствующие отношения и выполнены требуемые свойства.

Одна и та же теория может иметь много разных моделей. Например, моделями теории  $T = \langle \{\triangleleft\}, \{A_1, A_2, A_3\} \rangle$  где

$$A_1 : (\forall x)(\forall y)((x \triangleleft y) \supset \neg(y \triangleleft x));$$

$$A_2 : (\forall x)(\forall y)(\forall z)((x \triangleleft y) \& (y \triangleleft z) \supset (x \triangleleft z));$$

$$A_3 : (\forall x)(\forall y)((x \triangleleft y) \& (y \triangleleft x) \supset (x = y))$$

являются любые множества чисел, если в качестве  $\triangleleft$  взять отношение « $\leq$ » или « $\geq$ », любой алфавит с отношением « $\prec$ » (см. Пример 1.9.1) и т. п.

Возвращаясь к проблеме интерпретации сообщений, отметим, что теория содержит все имена (знаки) и служит основой для составления сообщений. Теория — это синтаксис языка сообщений, это правила построения сообщений. Для того, чтобы истолковать (принтерпретировать) какое-нибудь сообщение, необходимо использовать одну из моделей этой теории, причем различные модели порождают различные интерпретации.

Из сказанного следует, что любая задача обработки информации, которую мы хотим автоматизировать, должна быть поставлена на соответствующей модели. Прежде, чем разрабатывать алгоритм решения задачи обработки сообщений, необходимо **формализовать** задачу, т. е. четко описать модель, на которой ставится задача, и попытаться сформулировать теорию для этой модели. Так поступают, например, при реализации языка программирования высокого уровня [23].

# Глава 2

## Элементы теории алгоритмов

### Лекция 6

#### 2.1 Необходимость формального определения алгоритма

Переход от неформального  
к формальному существенно неформален!

*M. P. Шура-Бура*

В п. 1.9 мы определили **алгоритм** (или эффективную процедуру) как точно заданную последовательность правил, указывающую, каким образом можно за конечное число шагов получить выходное сообщение определенного вида, используя заданное входное сообщение.

При этом подчеркивалось, что действия, предписываемые алгоритмом, должны быть чисто механическими, всем понятными и легко выполнимыми; все исполнители алгоритма (люди и автоматы) должны понимать и выполнять эти действия одинаково.

Основным недостатком этого неформального определения алгоритма является его расплывчатость: непонятно, что значит «понимать и выполнять действия одинаково» и что значит «всем понятные, легко выполнимые действия». В самом деле, 1) алгоритм вычисления производной многочлена  $n$ -й степени прост и ясен тем, кто знает начала анализа, но для прочих он может быть совершенно непонятным; 2) алгоритм или не алгоритм процедура завязывания шнурков на ботинках? Скорее всего, нет (попробуйте описать его хотя бы словесно; точное описание требует знания общей топологии), хотя шнурки завязываются чисто механически всеми людьми старше 5–6 лет (это проверяется на «вступительных экзаменах» в детские сады).

Если бы все поставленные математические задачи могли быть алгоритмически решены, можно было бы не уточнять понятие алгоритма: когда для решения какого-нибудь класса задач предлагался конкретный алгоритм, возникало соглашение считать указанный алгоритм действительно алгоритмом. Но, к сожалению, не все математические задачи алгоритмически разрешимы, а доказательство алгоритмической неразрешимости какого-либо класса задач (т. е., того, что не существует алгоритма решения всех подобных задач) неизбежно содержит высказывания о *всех* мыслимых алгоритмах. Такие высказывания

невозможны без четкого представления о том, что является алгоритмом и что им не является, т. е. они невозможны без строгого формального определения алгоритма.

Формализация понятия алгоритма реализуется с помощью построения **алгоритмических моделей**. Можно выделить три основных типа универсальных алгоритмических моделей: **рекурсивные функции** (понятие алгоритма связывается с вычислениями и числовыми функциями), **машины Тьюринга** (алгоритм представляется как описание процесса работы некоторой машины, способной выполнять лишь небольшое число весьма простых операций), **нормальные алгоритмы Маркова** (алгоритмы описываются как преобразования слов в произвольных алфавитах). Весьма интересной представляется также выбор в качестве основной алгоритмической системы клеточных автоматов [24].

### 2.1.1 Рекурсивные функции

Рекурсивные функции предложил использовать в 30-х годах прошлого века известный американский математик Чёрч (т. н.  $\lambda$ -исчисление) [29, 33]. В 1960 г. другой американский математик Маккарти предложил интерпретируемый алгоритмический язык Lisp, реализующий  $\lambda$ -исчисление. Язык Lisp особенно популярен в США среди специалистов по искусственноому интеллекту. На Lisp'е написана начинка текстового редактора Emacs, а один из диалектов Lisp'a (AutoLisp) используется в качестве внутреннего языка известной чертежно-графической системы AutoCAD. В нашей стране функциональное программирование наиболее последовательно преподается в МИФИ. Нельзя не отметить, что к этому классу алгоритмических моделей принадлежит и самая компактная из них — комбинаторно-логическая (CL, [96, 97, 98]).

### 2.1.2 Машины Тьюринга

В 1936 г. аспирант Алан Тьюринг при исследовании алгоритмических проблем разрешимости (одной из знаменитых проблем Гильберта) предложил для уточнения понятия алгоритма использовать абстрактную вычислительную машину с очень простым набором операций. Построенная на базе этой машины алгоритмическая теория Тьюринга оказалась настолько плодотворной, что предвосхитила все последующие идеи универсальных вычислительных машин с программным управлением, включая идеи автоматизации программирования. В знак признания этих заслуг именем Тьюринга названа высшая награда ACM, ежегодно присуждаемая за наиболее выдающиеся результаты в области информатики. Список лауреатов тьюринговской премии можно найти на сайте ACM <http://www.acm.org/awards/taward.html>. В 1942 г. группа британских математиков и филологов под руководством А. Тьюринга успешно применила теорию алгоритмов к разгадке шифров фашистской Германии [100]. В послевоенное время Тьюринг принимал участие в создании реальных вычислительных машин, но на этот раз фортуна изменила ему и ни один из этих реальных проектов не был доведен до конца. Этому способствовали свойственные многим гениальным людям странности и чудачества в характере самого Тьюринга. Они же и привели к его трагической гибели от своеобразной английской рулетки: смешивая различные химические вещества он случайно получил яд и принял его вовнутрь. Недавно вышедшая книга Шарля Петцольда [108], посвящённая разбору основополагающего труда Тьюринга, ещё раз подтверждает актуальность Тьюринговской теории алгоритмов.

### 2.1.3 Нормальные алгоритмы Маркова

Алгоритмы Маркова предложены в 1950 г. нашим соотечественником академиком А. А. Марковым<sup>1</sup> (мл.) [17, 25]. На этой концепции основан оригинальный алгоритмический язык Рефал, предложенный в 1976 г. сотрудником Института Прикладной Математики В. Ф. Турчином, ныне профессором Нью-Йоркского университета. В. Ф. Турчин известен как составитель культовых сборников «Физики шутят» и «Физики продолжают шутить», в которых остроумие Турчина достигает таких же высот, как и его концептуальное кредо, изложенное в монографии [81]. Концепция НАМ и Рефала настолько ортогональна тьюринговой и другим операционным моделям, что она непременно цитируется в научных статьях и упоминается во всех курсах информатики. В настоящее время язык Рефал продолжает развиваться, в частности, в 2001–2003 гг. аспирант М. А. Левинская, выполняя диссертационную работу на соискание ученой степени кандидата физико-математических наук, предложила и успешно реализовала древовидное расширение Рефала [99].

На лекции демонстрируется пример нормального алгоритма Маркова и эквивалентная ему программа на Рефале прибавления 1 к десятичному числу (алгоритм работает не с самим числом, а с буквами его цифрового изображения):

>1 → 1> >0 → 0> > → < 1< → <0 0< ↠ 1 < ↠ 1 → >	add { e.1 '1' = <add e.1> '0'; e.2 '0' = e.2 '1'; = '1'; }
--	--

### 2.1.4 Тезис Тьюринга-Черча

Использование алгоритмических моделей опирается на тезис Тьюринга-Чёрча о том, что всякий интуитивный алгоритм может быть выражен средствами одной из алгоритмических моделей. Опробовать этот тезис пока никому не удалось. «Доказательством» же тезиса Тьюринга вполне может служить конструктивное построение фон Неймановской модели из Тьюринговской, предлагаемое в этом разделе курса.

Существенно, что все алгоритмические модели описывают один и тот же класс процессов обработки сообщений. Доказано, что одни модели сводятся к другим, т. е. всякий алгоритм, описанный средствами одной модели, может быть описан и средствами другой.

Наиболее близкой к вычислительным машинам является алгоритмическая модель Тьюринга. Поэтому именно эту модель мы выберем для формализации понятия алгоритма.

## 2.2 Машины Тьюринга

### 2.2.1 Неформальное описание

Машина Тьюринга состоит из ограниченной с одного конца бесконечной **ленты**, разделенной на **ячейки**, и комбинированной читающей и пишущей **головки**, которая может

<sup>1</sup>Вы можете больше узнать про личность академика А. А. Маркова у профессора В. К. Титова, который выполнял дипломную работу на мехмате МГУ под его руководством.

перемещаться вдоль ленты от ячейки к ячейке.

В каждой ячейке ленты может быть записан один знак алфавита  $A$ , называемого **рабочим алфавитом МТ**, либо **пробел** (его мы будем обозначать знаком  $\lambda$ ).

Головка МТ в каждый момент времени располагается над одной из ячеек ленты, называемой **рабочей ячейкой**, и воспринимает знак, записанный в этой ячейке (букву алфавита  $A$  или  $\lambda$ ). При этом головка находится в одном из конечного множества  $Q = \{q_0, q_1, \dots, q_s\}$  дискретных **состояний**, среди которых выделено одно **начальное состояние**  $q_0$ . В зависимости от состояния, в котором находится головка, и от буквы, записанной в рабочей ячейке, МТ выполняет одну из **команд**, составляющих ее **программу**.

Выполнение команды состоит в выполнении **элементарного действия**, предписанного этой командой, и переводе головки в новое состояние (которое, в частности, может совпадать со старым). Определено три вида элементарных действий: **сдвиг** головки на одну ячейку **влево** (если считать, что лента МТ ограничена слева, то для крайней левой ячейки сдвиг влево не определен), **сдвиг** головки на одну ячейку **вправо**, **запись** в рабочую ячейку какой-либо буквы рабочего алфавита  $A$  либо пробел (при этом буква, которая была записана в рабочей ячейке до выполнения записи, стирается). Таким образом, каждая команда может сменить рабочую ячейку, сделав новой рабочей ячейкой одну из соседних ячеек старой рабочей ячейки, или, не меняя рабочей ячейки, изменить знак, записанный в рабочей ячейке, и изменить состояние головки.

Перед началом работы МТ на ее ленту записывается исходное сообщение так, что в каждую ячейку ленты записывается одна буква сообщения, либо пробел. Будучи конечным, любое сообщение занимает конечное число ячеек ленты. При этом ячейки, расположенные справа от последней буквы исходного сообщения, заполняются пробелами и считаются *пустыми*. В начале работы МТ ее головка приводится в начальное состояние  $q_0$  и помещается над начальной рабочей ячейкой, которая определенным и фиксированным для *каждой конкретной МТ* образом расположена относительно исходного сообщения. Обычно в качестве рабочей ячейки мы будем брать ячейку, расположенную *вслед* за исходным сообщением, то есть, ячейку, содержащую символ пробела  $\lambda$ , расположенную непосредственно справа от последней буквы сообщения (все ячейки, расположенные справа от начальной рабочей ячейки, содержат в начале работы МТ символы  $\lambda$ ). Пара  $(q_0, a)$ , где  $q_0$  — начальное состояние головки,  $a \in A \cup \{\lambda\}$  — буква, записанная в начальной рабочей ячейке, определяет команду программы МТ, которая должна быть выполнена первой. В результате выполнения этой команды образуется новая пара  $(q, a')$ , где  $q$  — текущее состояние головки,  $a'$  — буква, записанная в текущей рабочей ячейке, которая определяет следующую команду и т. д.

Таким образом, работа МТ полностью определяется ее программой и сообщением, которое было записано на ленте перед началом работы МТ.

Каждая команда МТ описывается упорядоченной четверкой символов  $(q, a, v, q')$ , где

- $q \in Q$  — символ текущего состояния головки;
- $a \in A \cup \{\lambda\}$  — буква, записанная в текущей рабочей ячейке;
- $v \in \{l, r\} \cup A \cup \{\lambda\}$  — символ выполняемого действия ( $v = l$  означает сдвиг головки влево,  $v = r$  — сдвиг головки вправо, а  $v \in A \cup \{\lambda\}$  — запись соответствующего знака в рабочую ячейку).

- $q' \in Q$  — символ состояния, в которое команда переводит головку.

Пара  $(q, a)$  определяет, какая из команд программы МТ должна быть выполнена (предполагается, что программа содержит не более одной команды, начинающейся с пары  $(q, a)$ ).

### 2.2.2 Математически строгое определение

**Определение 2.2.1. Машиной Тьюринга** называется упорядоченная четверка объектов  $T = (A, Q, P, q_0)$ , где  $T$  — символ МТ,  $A$  — конечное множество букв (рабочий алфавит),  $Q$  — конечное множество символов (имен состояний),  $q_0$  — имя начального состояния,  $P$  — множество упорядоченных четверок  $(q, a, v, q')$ ,  $q, q' \in Q$ ,  $a \in A \cup \{\lambda\}$ ,  $v \in \{l, r\} \cup A \cup \{\lambda\}$  (программа), определяющее три функции: **функцию выхода**  $F_l: Q * \bar{A} \rightarrow \bar{A}$  ( $\bar{A} = A \cup \{\lambda\}$ ), **функцию переходов**  $F_t: Q * \bar{A} \rightarrow Q$ , и **функцию движения головки**  $F_v: Q * \bar{A} \rightarrow \{l, r, s\}$  (символ  $s$  означает, что головка неподвижна).

**Замечание.** Первоначально программы машин Тьюринга задавались наборами пятерок, включавших как функцию выхода, так и функцию движения. В частности, Шенон доказывал свои знаменитые теоремы именно в такой форме записи. В пятерках движение головки и запись буквы дополняют друг друга, а в четверках они являются взаимно исключающими действиями. Кроме того, существуют и другие способы задания программ Тьюринга (диаграммы состояний, таблицы переходов).

Рассмотрим реализацию одной и той же программы в четверках и пятерках.

#### Программа 2.1. Пример бесконечной анимации

00, , -, 00	00, , -, u, 00
00, -, \, 00	00, -, \, u, 00
00, \,  , 00	00, \,  , u, 00
00,  , /, 00	00,  , /, u, 00
00, /, , 01	00, /, , r, 00
01, , r, 00	

Эта программа сочетает кувырок палочки с ее продвижением вперед, фактически выполняя примитивную анимацию почти по Диснею: инерция человеческого зрения домысливает эти рывки до полноценных кульбитов. Поскольку начальное состояние совпадает с конечным, эта программа никогда не останавливается. В силу универсальности пятерок соответствующая МТ имеет вообще только одно состояние. Мы еще вернемся к этим программам после изучения теорем Шенона, который показал незавершимость программ машин Тьюринга с одним состоянием. Хотя эти примеры иллюстрируют необходимость дополнительных команд, а, следовательно, и состояний при программировании в четверках, все же можно составить аналогичный пример программы МТ в четверках с одним состоянием:

00, , -, 00	00, , -, u, 00
00, -, \, 00	00, -, \, u, 00
00, \,  , 00	00, \,  , u, 00
00,  , /, 00	00,  , /, u, 00
00, /, , 00	00, /, , u, 00

Но в этом примере пришлось пожертвовать движением и вместо кувыркающейся палочки мы получили вращающуюся на месте крутилку, используемую в ОС UNIX для текстовой индикации выполнения некоторого процесса.

**Определение 2.2.2. Ситуацией** (или **состоянием ленты**) называется упорядоченная пара объектов  $S = (z, k)$ , где  $S$  — имя ситуации,  $z$  — сообщение, записанное на ленте,  $k$  — неотрицательное целое число, равное расстоянию (в ячейках) от края ленты до рабочей ячейки. Иными словами,  $k$  фиксирует положение рабочей ячейки на ленте.

Иными словами, ситуация — это все, что записано на ленте с выделенной рабочей ячейкой. Очень удобно следующее наглядное обозначение для ситуации (линейная запись!):

$$S = [a_{i_1} a_{i_2} \dots a_{i_{k-1}} (a_{i_k}) a_{i_{k+1}} \dots a_{i_n} \lambda],$$

где  $[$  — край ленты,  $a_{i_k} \in A \cup \{\lambda\}$ ,  $\dots$  — пропущенные знаки, скобки  $()$  выделяют рабочую ячейку,  $\rangle$  — бесконечный правый край ленты, образованный пустыми ячейками.

**Определение 2.2.3. Конфигурацией** называется упорядоченная пара объектов  $C = (S, q)$ , где  $S$  — текущая ситуация,  $q$  — текущее состояние головки.

Для конфигурации будем использовать обозначение

$$C = [a_{i_1} a_{i_2} \dots a_{i_{k-1}} (q, a_{i_k}) a_{i_{k+1}} \dots a_{i_n} \lambda].$$

Каждая конфигурация  $C$  однозначно определяет команду МТ, которая должна быть выполнена. После выполнения указанной команды получается новая конфигурация  $C'$ , про которую говорят, что она **непосредственно следует** за конфигурацией  $C$ ; обозначение  $C \Rightarrow C'$ , где  $\Rightarrow$  — символ бинарного отношения непосредственного следования. Отношение непосредственного следования выполняется и для ситуаций  $S$  и  $S'$ , соответствующих конфигурациям  $C$  и  $C'$ ; его мы будем обозначать  $S \Rightarrow S'$ . Будем говорить, что конфигурация  $C'$  **следует** за конфигурацией  $C$ , если найдется  $n$  конфигураций  $C_0, C_1, \dots, C_{n-1}$  ( $n = 1, 2, \dots$ ), таких, что  $C = C_0 \dots C_{n-1} = C'$ , и  $C_i \Rightarrow C_{i+1}$ ,  $i = 0, 1, \dots, n - 2$ . Если конфигурация  $C'$  **следует** за конфигурацией  $C$ , будем писать  $C \xrightarrow{*} C'$ . Отношение следования тоже переносится на ситуации  $S \xrightarrow{*} S'$ . Здесь звездочка традиционно обозначает рефлексивно-транзитивное замыкание отношения следования, включающее случаи непосредственного следования (один такт) и пустого следования, возможного только при отказе МТ на неприменимых исходных конфигурациях, когда ни одного такта не было выполнено.

Ситуацию (конфигурацию), соответствующую начальному состоянию МТ, будем называть **начальной ситуацией** (конфигурацией).

**Пример 2.2.4.** Опишем машину Тьюринга  $K$ , которая копирует на ленте слово над однобуквенным алфавитом  $A = \{|$ }, т. е., исходя из ситуации  $S_0 = [z_0 \lambda w(\lambda) \lambda]$ , где  $z_0$  — некоторое (не интересующее нас) сообщение, которое может быть записано на ленте до копируемого слова  $w$ ,  $w = |...|$  ( $n$  палочек), заканчивает работу в ситуации  $S_f = [z_0 \lambda w \lambda w(\lambda) \lambda]$  (при этом слово  $w$  оказывается скопированным на ленте).

Пусть для определенности  $n = 5$ , т. е.,  $w = |||||$ , тогда работу машины  $K$  можно описать с помощью следующей последовательности ситуаций:

$$S_0 =$$

$$\begin{array}{llll}
 [z_0\lambda|||(\lambda)\lambda > \xrightarrow{*} [z_0(\lambda)|||\lambda\lambda > \Rightarrow [z_0\lambda(|)|||\lambda\lambda > \Rightarrow [z_0\lambda(\lambda)|||\lambda\lambda > \xrightarrow{*} \\
 [z_0\lambda\lambda|||(\lambda)\lambda > \Rightarrow [z_0\lambda\lambda|||\lambda(\lambda)\lambda > \Rightarrow [z_0\lambda\lambda|||(\lambda)|\lambda > \Rightarrow [z_0\lambda(\lambda)|||\lambda|\lambda > \xrightarrow{*} \\
 [z_0\lambda(|)|||\lambda|\lambda > \Rightarrow [z_0\lambda(|)|||\lambda|\lambda > \Rightarrow [z_0\lambda|(\lambda)|||\lambda|\lambda > \Rightarrow [z_0\lambda|\lambda|||(\lambda)|\lambda > \xrightarrow{*} \\
 [z_0\lambda|\lambda|||\lambda|(\lambda)\lambda > \Rightarrow [z_0\lambda|\lambda|||\lambda|(\lambda)\lambda > \xrightarrow{*} [z_0\lambda|(\lambda)|||\lambda||\lambda > \Rightarrow [z_0\lambda|(|)|||\lambda||\lambda > \xrightarrow{*} \\
 [z_0\lambda|||(|)||\lambda||\lambda > \Rightarrow [z_0\lambda||(\lambda)||\lambda||\lambda > \xrightarrow{*} [z_0\lambda|||\lambda|||(\lambda)\lambda >
 \end{array}$$

Программа может быть записана в виде следующей таблицы:

Состояние	Буква		
			$\lambda$
начало работы	—	$l$	поиск начала исходного слова
поиск начала исходного слова	$l$	поиск начала исходного слова	$\lambda$ начало исходного слова найдено
начало исходного слова найдено	—	$r$	копирование очередной буквы
копирование очередной буквы	$\lambda$	пометка копируемой буквы	$\lambda$ слово скопировано
пометка копируемой буквы	—	$\lambda$	перенос буквы в копию исходного слова
перенос буквы в копию исходного слова	—	$r$	поиск конца исходного слова
поиск конца исходного слова	$r$	поиск конца исходного слова	$\lambda$ конец исходного слова найден
конец исходного слова найден	—	$r$	поиск конца копии
поиск конца копии	$r$	поиск конца копии	$l$ конец копии найден
конец копии найден	—		очередная буква скопирована
очередная буква скопирована	$l$	поиск начала копии слова	—
поиск начала копии слова	$l$	поиск начала копии слова	$\lambda$ начало копии слова найдено
начало копии слова найдено	—	$l$	поиск пометки в исходном слове
поиск пометки в исходном слове	$l$	поиск пометки в исходном слове	$\lambda$ пометка найдена
пометка найдена	—		копирование следующей буквы
копирование следующей буквы	$r$	копирование очередной буквы	—
слово скопировано	—	$r$	установка головки
установка головки	$r$	установка головки	$\lambda$ конец работы
конец работы	$s$	конец работы	$s$ конец работы

Конечно, состояния можно было бы перенумеровать, и тогда программа имела бы более

компактный, но менее понятный вид:

	00	01	02	03	04	05	06	07	08
	<i>l</i>	<i>l</i>	<i>r</i>		$\lambda$	<i>r</i>	<i>r</i>	<i>r</i>	<i>r</i>
	1	1	3	4	5	6	6	8	8
$\lambda$	<i>l</i>	$\lambda$	<i>r</i>	$\lambda$	$\lambda$	<i>r</i>	$\lambda$	<i>r</i>	$\lambda$
	1	2	3	16	5	6	7	8	9

	09	10	11	12	13	14	15	16	17	18
		<i>l</i>	<i>l</i>	<i>l</i>	<i>l</i>		<i>r</i>	<i>r</i>	<i>r</i>	<i>s</i>
	10	11	11	13	13	15	03	17	17	18
$\lambda$		<i>l</i>	$\lambda$	<i>l</i>	$\lambda$		<i>r</i>	<i>r</i>	$\lambda$	<i>s</i>
	10	11	12	13	14	15	03	17	18	18

Эта таблица легко преобразуется в "машинный" код интерпретатора МТ, используемого для выполнения лабораторных работ:

00,   , <, 01	04, , , 05	09, ,   , 10	14, ,   , 15
00, , <, 01	05,   , >, 06	10,   , <, 11	15,   , >, 03
01,   , <, 01	05, , >, 06	10, , <, 11	15, , >, 03
01, , , 02	06,   , >, 06	11,   , <, 11	16,   , >, 17
02,   , >, 03	06, , , 07	11, , , 12	16, , >, 17
02, , >, 03	07,   , >, 08	12,   , <, 13	17,   , >, 17
03,   ,   , 04	07, , >, 08	12, , <, 13	17, , , 18
03, , , 16	08,   , >, 08	13,   , <, 13	18, , , 18
03, , , 16	08, , , 09	13, , , 14	
04,   , , 05	09,   ,   , 10	14,   ,   , 15	

Из примера (2.2.4) видно, что даже простые действия выполняются на машинах Тьюринга достаточно сложными программами. Кроме того, запись программ МТ в виде таблиц плохо обозрима и потому неудобна для понимания. Поэтому мы перейдем к другой форме записи.

## Лекция 7

### 2.2.3 Диаграммы Тьюринга

Диаграммы Тьюринга представляют одни МТ через другие, более простые МТ иным, визуально-топологическим способом, причём, как будет показано далее, этот способ не менее строг и полон, нежели "обычные" МТ. Так, машина, копирующая на ленте записанное на ней слово, рассмотренная в примере (2.2.4), может быть представлена через МТ, которые ищут начало слова на ленте, конец слова на ленте, копируют одну из букв слова и т. д. (имена состояний подобраны в этом примере так, чтобы легко было бы выделить более простые МТ). Эти более простые МТ в свою очередь могут быть представлены через еще более простые МТ (это тоже нетрудно проиллюстрировать на примере (2.2.4)) и т. д. Такой нисходящий процесс представления МТ через более простые МТ должен обязательно оборваться, так как рано или поздно мы сведем описание каждой

из рассматриваемых МТ к элементарным действиям, введенным при определении МТ. При этом рассматриваемая МТ будет описана через **элементарные** МТ, т. е. такие, которые уже нельзя описать через более простые МТ, так как каждая из них выполняет всего одно элементарное действие и останавливается.

Элементарные МТ над алфавитом  $A_p = (a_1, a_2, \dots, a_p)$  определяются следующими программами.

1. Элементарная МТ  $l$  (сдвиг головки на одну ячейку влево)

$$\begin{array}{ll} (q_0, \lambda, l, q_1) & (q_1, \lambda, s, q_1) \\ (q_0, a_1, l, q_1) & (q_1, a_1, s, q_1) \\ \dots & \dots \\ (q_0, a_p, l, q_1) & (q_1, a_p, s, q_1) \end{array}$$

Если применить МТ  $l$  к крайней левой ячейке ленты, то результат будет таким, как если бы было применено элементарное действие  $l$ : действие МТ не определено ввиду перехода через край ленты.

2. Элементарная МТ  $r$  (сдвиг головки на одну ячейку вправо)

$$\begin{array}{ll} (q_0, \lambda, r, q_1) & (q_1, \lambda, s, q_1) \\ (q_0, a_1, r, q_1) & (q_1, a_1, s, q_1) \\ \dots & \dots \\ (q_0, a_p, r, q_1) & (q_1, a_p, s, q_1) \end{array}$$

3. Элементарные МТ  $\lambda$ ,  $a_1, \dots, a_p$  (запись соответствующего знака на ленту). Мы приведем программу машины  $a_i$  ( $i = 0, 1, \dots, p$ ;  $a_0 = \lambda$ ):

$$\begin{array}{ll} (q_0, \lambda, a_i, q_1) & (q_1, \lambda, s, q_1) \\ (q_0, a_1, a_i, q_1) & (q_1, a_1, s, q_1) \\ \dots & \dots \\ (q_0, a_p, a_i, q_1) & (q_1, a_p, s, q_1) \end{array}$$

Комбинируя элементарные МТ, можно получать более сложные МТ. Например, применяя последовательно элементарную МТ  $l$  (или  $r$ ) до тех пор, пока в рабочей ячейке не встретится знак  $\lambda$ , получим МТ, сдвигающую головку до левого (правого) конца слова, т. е., выполняющую действие  $[z_0 \lambda w(\lambda) \lambda] \Rightarrow [z_0(\lambda) w \lambda]$  (соответственно  $[z_0(\lambda) w \lambda] \Rightarrow [z_0 \lambda w(\lambda) \lambda]$ ). Построенные МТ мы будем обозначать буквами  $L$  (соответственно  $R$ ). При конструировании новых МТ можно наряду с элементарными МТ использовать и машины  $L$  и  $R$ :

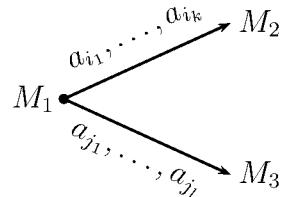
$$L = \textcircled{L}^{\neq \lambda} \quad R = \textcircled{R}^{\neq \lambda}$$

Диаграмма Тьюринга, будучи ориентированным на человека вариантом алгоритмической модели Тьюринга, рисуется на чистом листе бумаги слева-направо и сверху вниз. При использовании систем-диаграммеров, этот принцип сохраняется, потому что

читать диаграмму (в несемитских и неиероглифических письменностях) принято также слева-направо и сверху-вниз. Состояния будем обозначать на диаграммах точками ( $\bullet$ ). Позиционно-топологическая различимость мест для точек (точкомест) заменяет в диаграммах уникальные имена состояний. Каждому символу МТ соответствует два состояния — начальное и заключительное. Поэтому каждый символ МТ должен быть изображен на диаграмме между двумя точками (например,  $\cdot R \cdot$  или  $\cdot L \cdot$  и т. д.). Если после действия, определяемого машиной  $M_1$ , должно быть выполнено действие, определяемое машиной  $M_2$ , то правая точка (заключительное состояние) машины  $M_1$  соединяется стрелкой с левой точкой (начальным состоянием) машины  $M_2$ , причем над стрелкой надписываются те знаки рабочего алфавита  $A_p$ , которые вместе с заключительным состоянием машины  $M_1$  определяют переход к действию, которое задается машиной  $M_2$ .

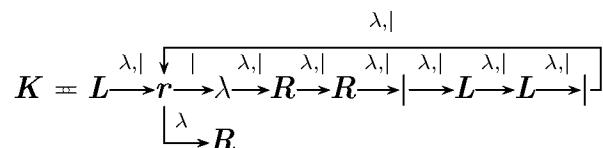
$$M_1 \xrightarrow{a_{i_1}, \dots, a_{i_k}} M_2$$

В случае разветвки над двумя и более стрелками, ведущими из одной машины, надписываются разные непересекающиеся наборы букв рабочего алфавита, по которым осуществляется продолжение работы МТ по соответствующим ветвям. Если объединение этих наборов не совпадает со всем рабочим алфавитом, то на остальных буквах в этом месте машина Тьюринга будет отказывать. Если же наборы букв разных стрелок, исходящих из одной и той же вершины, пересекаются, то такая диаграмма Тьюринга описывает недетерминированный процесс, не являющийся алгоритмом в обычном смысле этого слова.



Для повторения какого-либо участка диаграммы до тех пор, пока в рабочей ячейке находится одна из букв некоторого фиксированного набора, необходимо замкнуть этот участок диаграммы стрелкой с соответствующей надписью. Прекращение повторения осуществляется по букве, не входящей в этот набор.

Таким образом, диаграмма Тьюринга состоит из символов (имен) машин Тьюринга, точек и стрелок, над которыми надписаны знаки алфавита  $A_p$ . В качестве примера рассмотрим диаграмму МТ из примера (2.2.4). Она имеет вид:



Остальные циклы в этой диаграмме скрыты в машинах  $R$  и  $L$ .

Примем следующие упрощения в записи диаграмм: если над стрелкой надписаны все знаки алфавита  $A_p \cup \{\lambda\}$ , то их можно опустить; если стрелка, над которой ничего не написано, соединяет две соседние точки, то ее также можно опустить, слияя точки, которые

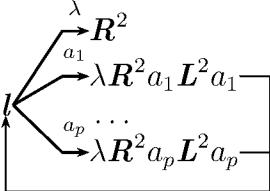
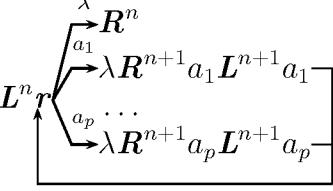
соединяла эта стрелка, в одну (заключительное состояние машины  $M_1$  совпадает с начальным состоянием машины  $M_2$ ); если на диаграмме расположены подряд  $k$  символов одной и той же машины  $M$ , то их можно заменить одним символом  $M^k$ . После перечисленных упрощений диаграмма машины из примера (2.2.4) примет вид:

$$K = Lr \xrightarrow{\lambda} \boxed{R^2 | L^2 |} \xrightarrow{\lambda} R$$

Легко видеть, что эта диаграмма значительно компактнее и нагляднее программ из примера (2.2.4).

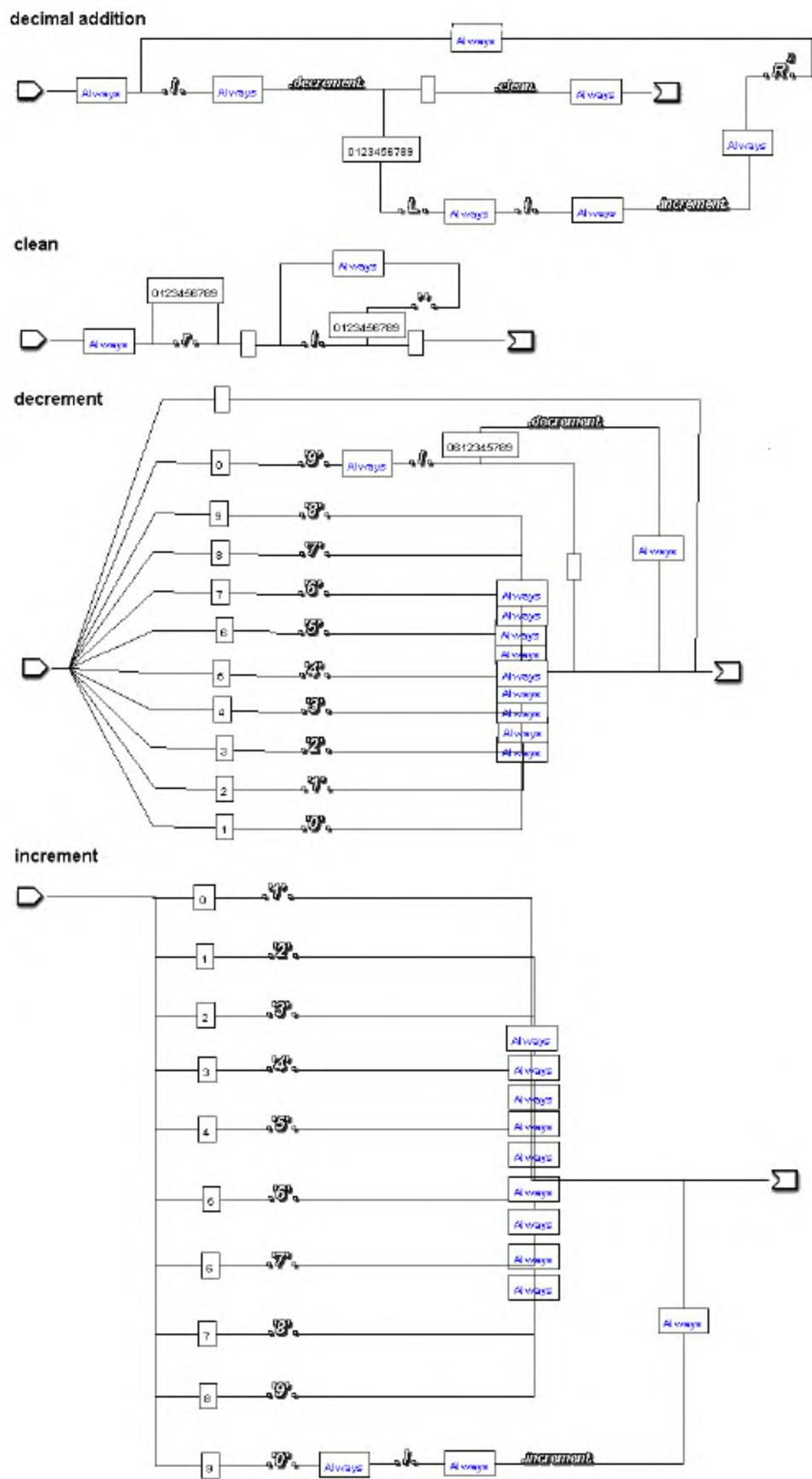
Приведем некоторые примеры диаграмм Тьюринга [7]:

Символ	Действие	Диаграмма
$R$	$[(\lambda)w\lambda] \xrightarrow{*} [\lambda w(\lambda)\lambda]$	$\overset{\neq\lambda}{\textcirclearrowleft}_r$
$L$	$[\lambda w(\lambda)\lambda] \xrightarrow{*} [(\lambda)w\lambda]$	$\overset{\neq\lambda}{\textcirclearrowleft}_l$
$\mathfrak{R}$	$[(\lambda)w_1\lambda w_2\lambda \dots \lambda w_n\lambda] \xrightarrow{*} \xrightarrow{*} [\lambda w_1\lambda w_2\lambda \dots \lambda w_n(\lambda)\lambda]$	$\overset{\neq\lambda}{\textcirclearrowleft}_{Rr} \xrightarrow{\lambda} l$
$\mathfrak{L}$	$[\lambda w_1\lambda w_2\lambda \dots \lambda w_n(\lambda)\lambda] \xrightarrow{*} \xrightarrow{*} [(\lambda)w_1\lambda w_2\lambda \dots \lambda w_n\lambda]$	$\overset{\neq\lambda}{\textcirclearrowleft}_{Ll} \xrightarrow{\lambda} r$
$K$	$[\lambda w(\lambda)\lambda] \xrightarrow{*} [\lambda w\lambda w(\lambda)\lambda]$	$Lr \xrightarrow{\lambda} \boxed{\begin{array}{c} \xrightarrow{a_1} R \\ \xrightarrow{a_p \dots} \xrightarrow{\lambda} \boxed{\lambda R^2 a_1 L^2 a_1} \end{array}}$
$W_r$	$[(\lambda)w\lambda] \xrightarrow{*} [\lambda\lambda \dots (\lambda)]$	$r \xrightarrow{\downarrow \neq\lambda} \boxed{\lambda}$
$W_l$	$[\lambda w(\lambda)] \xrightarrow{*} [(\lambda)\lambda \dots \lambda]$	$\boxed{l \xrightarrow{\downarrow \neq\lambda} \lambda}$
$V$	$[\lambda w_1\lambda w_2(\lambda)\lambda] \xrightarrow{*} [\lambda w_2(\lambda)\lambda]$	$\# L^2 \# W_r \xrightarrow{\lambda} \boxed{\begin{array}{c} \xrightarrow{\lambda} \overset{\neq\#}{\textcirclearrowleft}_l \xrightarrow{\lambda} \lambda R \\ \xrightarrow{a_1} \xrightarrow{\lambda} \overset{\neq\lambda}{\textcirclearrowleft}_l \xrightarrow{\lambda} ra_1 \\ \dots \\ \xrightarrow{a_p} \xrightarrow{\lambda} \overset{\neq\lambda}{\textcirclearrowleft}_l \xrightarrow{\lambda} ra_p \end{array}}$

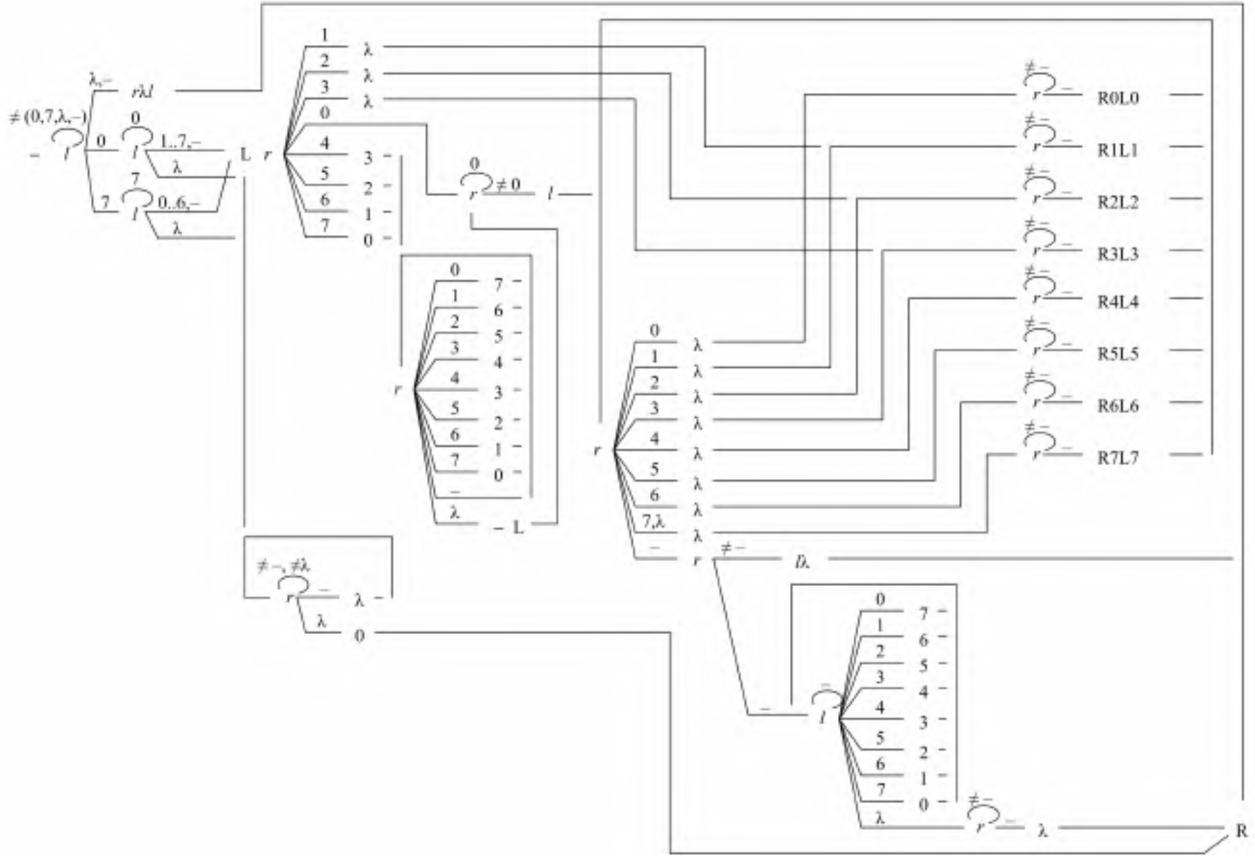
Символ	Действие	Диаграмма
$I$	$[\lambda w(\lambda)\lambda] \xrightarrow{*} [\lambda w\lambda w^{-1}(\lambda)\lambda]$	
$K_n$	$[\lambda w_1\lambda w_2\lambda \dots \lambda w_n(\lambda)\lambda] \xrightarrow{*} \xrightarrow{*} [\lambda w_1\lambda w_2\lambda \dots \lambda w_n\lambda w_1(\lambda)\lambda]$	

**Замечание.** В силу относительной адресации МТ они чрезвычайно чувствительны к начальной ситуации на ленте и при неточном позиционировании будут работать неправильно или не будут работать вообще. Но есть исключения. Машины  $R$  и  $L$  проезжают не только полное слово от границы до границы, но также и остаток слова и пустое слово.

**Замечание.** Наше определение диаграммы путем ее нисходящей декомпозиции на более простые машины не коснулось случая, когда среди таких машин снова появилась бы сама декомпозируемая машина, точнее, ее символ. Хотя рекурсия принципиально не свойственна низкоуровневым операционным алгоритмическим моделям, диаграммы, будучи процедурным программированием, вполне могут быть рекурсивными. Более того, функциональные алгоритмические модели ( $\lambda$ -исчисление, НАМ) существенно рекурсивны. Однако рекурсивное определение диаграммы Тьюринга не может быть статически расписано до элементарных машин и должно быть возложено на среду выполнения диаграмм, которая динамически конструирует необходимый диагностический код в процессе интерпретации. Если она рекурсивна, то такие диаграммы Тьюринга допустимы. Например, в натурализованном алгоритме десятичного сложения синхронно работающие машины декремента второго операнда и инкремента первого, решающие задачу, могут быть описаны рекурсивно: в случае возникновения займа (переноса), необходимо применить ту же самую машину декремента (инкремента) к следующему разряду. Именно так реализовано десятичное сложение в диаграммере Д. В. Дзюбы JDT2:



Ниже приводится весьма каллиграфический пример диаграммы Тьюринга, сконструированный студентом Буниным С.А. в среде MS Visio (стрелки добавить по вкусу).



## 2.2.4 Понятие моделирования

**Определение 2.2.5.** Рассмотрим две МТ  $T = (A, Q, P, q_0)$  и  $T' = (A', Q', P', q'_0)$ . Будем говорить, что машина  $T'$  моделирует машину  $T$ , и обозначать это  $T' \simeq T$ , если выполняются следующие условия:

1. Указан способ кодирования знаков (букв) алфавита  $A$  знаками (буквами или словами) алфавита  $A' C$ :  $A \rightarrow A'$ ;
  2. Каждому состоянию  $q \in Q$  машины  $T$  поставлено в соответствие некоторое состояние  $q' \in Q'$  машины  $T'$ , т. е. определено отображение  $Q \rightarrow Q'$ .
- Условия (1) и (2) позволяют для каждой конфигурации  $C$  машины  $T$  составить конфигурацию  $C'$  машины  $T'$ , являющуюся образом конфигурации  $C$ , заменяя каждую букву на ленте машины  $T$  ее кодом, а состояние  $q$  — состоянием  $q'$ ;*
3. Если  $C_0$  — начальная конфигурация машины  $T$ , то ее образ  $C'_0$  — начальная конфигурация машины  $T'$ ;
  4. Если машина  $T$  из начальной конфигурации  $C_0$  после конечного числа *тактов* (переходов от одной конфигурации к другой) останавливается в конфигурации  $C_k$ , то

- машина  $T'$  из начальной конфигурации  $C'_0$ , являющейся образом  $C_0$ , после конечного числа тактов также останавливается в конфигурации  $C'_k$ , являющейся образом  $C_k$ ;
5. Если из начальной конфигурации  $C_0$  машина  $T$  пробегает (конечную или бесконечную) последовательность конфигураций  $C_0, C_1, \dots, C_k, \dots$ , то каждая последовательность конфигураций, пробегаемая машиной  $T'$  из начальной конфигурации  $C'_0$  (образа конфигурации  $C_0$ ), содержит в качестве подпоследовательности последовательность конфигураций  $C'_0, C'_1, \dots, C'_k, \dots$ , где  $C'_i$  ( $i = 0, 1, \dots, k, \dots$ ) — образы конфигураций  $C_i$  машины  $T$ .

Из определения 2.2.5 следует, что если машина  $T'$  моделирует машину  $T$ , то она описывает тот же самый алгоритм, что и  $T$ , но, возможно, проходит при выполнении алгоритма большее число промежуточных конфигураций. Таким образом, понятие моделирования вводит бинарное отношение алгоритмического равенства между машинами Тьюринга. Другие операции отношения над алгоритмами будут введены позднее, по мере изложения Тьюринговской теории алгоритмов. Следует заметить, что равенство алгоритмов даже в таком простом случае — машин Тьюринга — требует более сложного определения, чем равенство строк или чисел.

## 2.2.5 Эквивалентность диаграмм и программ МТ

**Теорема 2.2.6.** Каждой программе  $P$ , задающей МТ  $T = (A, Q, P, q_0)$ , можно эффективным образом сопоставить диаграмму  $D$ , образованную символами элементарных МТ, так, чтобы МТ, определяемая этой диаграммой, моделировала бы машину  $T$ .

▷Нам нужно указать способ (алгоритм) эффективного построения диаграммы по программе  $P$ , а потом убедиться, что для исходной МТ и МТ, определяемой диаграммой  $D$ , выполняются все пять условий определения 2.2.5.

**2.2.6.1. Построение диаграммы  $D$ .** Каждой строке  $(q, a, v, q') \in P$  поставим в соответствие на диаграмме символы  $\cdot v \cdot$ . Для  $v = s$  ничего ставить не надо, ведь пустые действия на диаграмме не отображаются. В этом случае окончанием работы машины, определяемой диаграммой, будет правая точка предыдущего элемента диаграммы.

**Замечание.** Если бы наши программы были бы в пятерках, непосредственной трансляции команды в элемент диаграммы здесь бы не получилось. Вот ещё одна причина использования четвёрок!

Поставим еще одну точку ( $\bullet$ ), соответствующую начальному состоянию диаграммы, и соединим ее стрелками  $\rightarrow$  с левыми точками всех символов  $v$ , соответствующих строкам вида  $(q_0, a_i, v, q) \in P$ . Для каждой пары строк  $(q, a_k, v, q') \in P$  и  $(q', a_j, v', q'') \in P$  соединим стрелкой  $\rightarrow$  правую точку символа  $v$ , соответствующую строке  $(q, a_k, v, q') \in P$ , с левой точкой символа  $v'$ , соответствующей строке  $(q', a_j, v', q'') \in P$  и надпишем над стрелкой букву  $a_j$ . В полученной таким образом диаграмме произведем необходимые упрощения, описанные в конце п. 2.2.3. Диаграмма  $D$  построена.

**2.2.6.2. Доказательство того, что машина  $T_D$ , определяемая диаграммой  $D$ , моделирует машину  $T$ .** Для этого достаточно показать, что выполнены условия (1)–(5) определения 2.2.5. Условие (1) выполняется потому, что алфавиты машин  $T$  и  $T_D$  совпадают.

Состояниям машины  $T_D$  соответствуют точки на диаграмме  $D$ . По построению диаграммы каждому состоянию  $q$  машины  $T$  соответствует одна или две точки на диаграмме

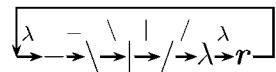
*D*. В последнем случае сопоставим состоянию  $q$  правую точку соответствующего символа на диаграмме. Таким образом, условие (2) выполнено.

Условия (3)–(5) выполнены, так как диаграмма  $D$  определяет ту же последовательность элементарных действий, что и программа  $P$ , за исключением того, что у машины  $T_D$  могут оказаться лишние «пустые» действия, соответствующие переходам от левых точек символов  $v$  к правым.  $\square$

**Замечание.** Процедура построения диаграммы МТ по ее программе (последовательности четверок) является алгоритмом и, следовательно, может быть автоматизирована. В соответствии с этим алгоритмом может быть построен транслятор программ Тьюринга в диаграммы, причем основная сложность такой трансляции заключается в генерации диаграммы, представление в ЭВМ которой одновременно должно быть удобным и для визуализации, и для обработки, и, конечно же, для выполнения. В современных системах автоматизации программирования (CASE) диаграммы являются одним из самых распространенных способов представления алгоритмов, автоматически преобразуемых в программы (иногда наоборот!). Как было сказано выше, именно диаграммный способ построения является наиболее удобным для нисходящей разработки алгоритма.

**Пример 2.2.7.** Рассмотрим диаграмму, полученную в результате применения теоремы к программе 2.1:

После преобразований эта диаграмма примет вид:



**Теорема 2.2.8.** Каждой диаграмме Тьюринга  $D$  может быть *эффективным образом* сопоставлена программа  $P$  так, что МТ, описываемая этой программой, моделирует МТ, описываемую диаграммой  $D$ .

▷Доказательство снова состоит из двух этапов.

**2.2.8.1. Построение программы  $P$ .** Заменим на диаграмме  $D$  все символы неэлементарных МТ их диаграммами, продолжая такую замену до тех пор, пока на диаграмме не останутся только обозначения элементарных МТ. В полученной диаграмме перенумеруем одинаковые символы элементарных МТ, снабдив их числовыми индексами. Перенумеруем также все точки, соответствующие *выходным* состояниям. Каждому символу элементарной МТ сопоставим программу этой машины из п. 2.2.3, пометив каждое состояние этой программы дополнительными индексами  $j$  и  $v$ , где  $v$  совпадает с названием соответствующей элементарной МТ, а  $j$  — номер, присвоенный этой элементарной МТ. Например, элементарной машине  $r_j$  ( $j$ -е вхождение элементарной МТ  $r$  в диаграмме) будет сопоставлена программа

$$\begin{array}{ll}
 (q_0^{v,j}, \lambda, r, q_1^{v,j}) & (q_1^{v,j}, \lambda, s, q_1^{v,j}) \\
 (q_0^{v,j}, a_1, r, q_1^{v,j}) & (q_1^{v,j}, a_1, s, q_1^{v,j}) \\
 \dots & \dots \\
 (q_0^{v,j}, a_p, r, q_1^{v,j}) & (q_1^{v,j}, a_p, s, q_1^{v,j})
 \end{array}$$

Точке с индексом  $j$  сопоставим программу вида

$$\begin{array}{l}
 (q_j, \lambda, s, q_j) \\
 (q_j, a_1, s, q_j) \\
 \dots
 \end{array}$$

$$(q_j, a_p, s, q_j)$$

Соединим все программы, сопоставленные символам элементарных МТ и точкам, в одну и произведем в полученной программе следующие изменения:

- если на диаграмме символы элементарных МТ  $v_j$  и  $v_k$  соединены стрелкой  $\xrightarrow{a}$ , то строку  $(q_1^{v,j}, a, s, q_1^{v,j})$  заменим на  $(q_1^{v,j}, a, a, q_0^{v,k})$ , т. е. останов машины  $v_j$  заменим на перезапись буквы рабочей ячейки с последующим переходом прямо в начальное состояние машины  $v_k$ , минуя конечное состояние машины  $v_j$ ;
- если на диаграмме между символами элементарных МТ  $v_j$  и  $v_k$  стоит точка, то проведем вышеуказанную замену для всех строк вида  $(q_i, a, s, q_i)$ , тем самым предотвращается остановка машины на промежуточных этапах, изображаемых точками — «пустыми» машинами;
- введем новое начальное состояние  $q_0$  для генерируемой программы (оно соответствует крайней левой точке диаграммы) и добавим к программе пустые командные строки  $(q_0, \lambda, \lambda, q_0), (q_0, a_i, a_i, q_0)$  ( $i = 1, 2, \dots, p$ ), где  $a_i$  — символы элементарных МТ, непосредственно следующих за крайней левой точкой диаграммы. Эти строки являются модифицированной программой элементарной машины «●», в которой остановка заменена перезаписью буквы в рабочей ячейке, т. е. реализуют холостой ход конструируемой программы;
- перенумеруем все состояния программы, за исключением нового  $q_0$  произвольным образом, введя сплошную нумерацию состояний (например, в лексикографическом возрастании индексов:  $q_i^{v,j} \rightarrow q_k$ . Постройте сами такое отображение индексов по схеме  $\{l, r, s, \lambda, a_i\} \times \mathbb{N}^2 \rightarrow \mathbb{N}$ ).

На этом построение программы  $P$  завершается.

#### 2.2.8.2. Проверка условий (1)–(5) определения 2.2.5

Условие (1) выполняется, так как рабочие алфавиты рассматриваемых МТ совпадают. Условие (2) выполняется по построению программы  $P$ . Условия (3)–(5) выполняются потому, что обе МТ выполняют над сообщением, записанным на ленте (по построению программы  $P$ ) одни и те же элементарные действия в одной и той же последовательности (дополнительные элементарные действия имеют вид  $(q, a, a, q')$ , т. е. не меняют ситуации на ленте).  $\square$

В заключение теоремы можно сказать, что трансляция любой высокогоуровневой визуальной диаграммы в низкоуровневый машинный код описана нами достаточно конструктивным алгоритмическим образом. Здесь, как и раньше, основные трудности реализации заключаются в эффективном представлении диаграмм в памяти ЭВМ. Такого рода трансляторы диаграмм в программный код имеются в CASE-системах автоматизации программирования, т. н. *диаграммерах*, выходными языками которых обычно являются C++, HTML, JavaScript или SQL. Кроме того, необходимо заметить, что доказательство взаимной эквивалентности программ и диаграмм Тьюринга существенно проще, чем, например, строгое сопоставление Паскаля и Си.

## Лекция 8

## 2.3 Нормальные алгоритмы Маркова

В 1947–1954 гг. академик А. А. Марков (мл.) предложил алгоритмическую систему, в которой, как и в машине Тьюринга, преобразуются текстовые сообщения, но на основе других принципов [3, 25]. Элементарными тактами обработки алгоритмов Маркова являются замены подслов исходного сообщения на некоторые другие слова.

Алгоритмы Маркова представляют собой мощное средство описания, во многих случаях обладающее большей наглядностью и компактностью, например, по сравнению с машинами Тьюринга.

Нормальные алгоритмы Маркова по существу являются *детерминистическими* текстовыми заменами, которые для каждого входного слова однозначно задают вычисления и тем самым в случае их завершения порождают определенный результат. Это может быть обеспечено, например, установлением приоритета применения правил. Такие приоритеты могут быть заданы линейным порядком их записи [20]. В алгоритмической системе Маркова нет понятия ленты и подразумевается непосредственный доступ к различным частям преобразуемого слова.

Марковская стратегия применения правил заключается в следующем:

1. если применимо несколько правил, то берется правило, которое встречается в описании алгоритма первым;
2. если правило применимо в нескольких местах обрабатываемого слова, то выбирается самое левое из этих мест.

Поскольку нет никакой свободы в выборе очередного шага обработки, алгоритмы Маркова являются детерминистическими. Поскольку результат алгоритма Маркова этой стратегией однозначно, то он также является *детерминированным*.

Итак, нормальный алгоритм Маркова (НАМ) представляет собой упорядоченный набор правил-продукций — пар слов (цепочек знаков, в том числе пустых цепочек длины 0), соединенных между собой символами  $\rightarrow$  или  $\mapsto$ . Каждая продукция представляет собой формулу замены части входного слова, совпадающей с левой частью формулы, на ее правую часть. И левая, и правая части продукции могут быть пустыми: либо выполняется безусловная подстановка правой части, либо удаляется часть исходного слова. Однако поскольку пустое слово присутствует и слева, и справа от каждой буквы преобразуемого слова, то подстановка с пустой левой частью зацекливается, и соответствующий алгоритм неприменим ни к какому входному слову. Если удается применить какую-то формулу подстановки, заменив вхождение ее левой части в исходном слове на правую часть, происходит возврат в начало алгоритма, и снова ищутся вхождения левой части первой продукции в измененное слово. Если же какую-то продукцию не удалось применить, проверяется следующая за ней, и так далее. Процесс выполнения нормального алгоритма заканчивается в одном из двух случаев: либо все формулы оказались неприменимыми, т. е. в обрабатываемом слове нет вхождений левой части ни одной формулы подстановки; либо только что применилась так называемая терминальная (завершающая) продукция, в которой правую и левую часть разделяет символ  $\mapsto$ . Терминальных продукции в одном НАМ может быть несколько.

В любом из этих случаев нормальный алгоритм применим к данному входному слову. Если же в процессе выполнения нормального алгоритма бесконечно долго применяются нетерминальные правила, то алгоритм неприменим к данному входному слову.

Существует два простых достаточных признака применимости НАМ ко всем входным словам [3]:

1. левые части всех продукции непустые, а в правых частях нет букв, входящих в левые части;
2. в каждом правиле правая часть короче левой.

Рассмотрим примеры НАМ. Алгоритм кодирования по Цезарю:

$$C: A_{26} \rightarrow A_{26},$$

т. е. нам необходимо сопоставлять каждой очередной букве  $a_i$  исходного слова «следующую» кодированную букву  $a_{i+3} \pmod{26}$ . В отличие от машины Тьюринга, передвигающей головку вдоль ленты, нормальный алгоритм Маркова не имеет такого же непосредственного доступа к очередной букве обрабатываемого сообщения. Однако текущая позиция в обрабатываемом по Маркову слове может быть смоделирована с помощью маркерного или курсорного символа, исследованного учениками Маркова Нагорным и В. С. Черняевским [101]. Маркер является дополнительной несобственной буквой рабочего алфавита, помечающей интересующее нас место в обрабатываемом слове. Заметим, что метасимволы в НАМ играют ту же роль, что и промежуточные состояния в МТ.

$$\begin{array}{ll} *A \rightarrow D* & (1) \\ *B \rightarrow E* & (2) \\ \dots & \dots \\ *W \rightarrow Z* & (23) \\ *X \rightarrow A* & (24) \\ *Y \rightarrow B* & (25) \\ *Z \rightarrow C* & (26) \\ * \mapsto & (27) \\ \rightarrow * & (28) \end{array}$$

Первые 27 правил нашего алгоритма не применимы ни к каким правильным исходным данным, поскольку они не могут содержать несобственного знака \*. Следовательно, сработает 28-ое правило, которое поставит \* вместо первого пустого слова, т. е. в начало перекодируемой последовательности. После чего, согласно правилам исполнения НАМ, для поиска следующей применимой продукции они будут перебираться сначала. Теперь одно из первых 27 правил может быть применено. Причем 27-ое правило сработает для пустого слова, к которому алгоритм тоже применим, а одно из правил (1)–(26) применится к первой букве слова, перекодируя ее и перебрасывая звездочку к следующей букве. Далее эта операция перекодирования с продвижением курсора повторяется до тех пор, пока не исчерпаются буквы входного слова. Это произойдет, когда маркер окажется за последней буквой и ни одно из 26 первых правил не будет применимо.

Проследим за работой марковского алгоритма, кодирующего имя императора, давшее название коду:

Слово	Правило
CAESAR	
*CAESAR	(3)

F*AESAR	(1)
FD*ESAR	(5)
FDH*SAR	(13)
FDHV*AR	(1)
FDHVD*R	(12)
FDHVDU*	(27)
FDHVDU	

В результате работы алгоритма маркер движется вправо до конца слова, пока не будет удален терминирующим правилом (27).

Для некоторых задач требуется два и более маркеров. Например, копирование слова требует пометки исходной и целевой позиций обрабатываемого слова.

$$\begin{array}{ll}
 \alpha\beta b & \rightarrow \beta ba \quad (1) \\
 a\beta & \rightarrow \beta b\beta a \quad (2) \\
 b & \rightarrow c \quad (3) \\
 c & \rightarrow \quad \quad \quad (4) \\
 a & \mapsto \quad \quad \quad (5) \\
 & \rightarrow a \quad \quad \quad (6)
 \end{array}$$

При копировании слова возникает проблема постановки копии очередной буквы на правильную позицию (как мы знаем, НАМ не в состоянии обратиться к конкретной позиции). Для этого в описанном выше изощренном алгоритме используется буква  $a$ , чтобы пометить текущую букву оригинала. Согласно правилу (2), буква копируется, маркер  $a$  перемещается за нее, а букву и ее копию разделяет маркер  $b$ . Если копируемое слово содержит более одной буквы, то маркер  $b$  используется для перемещения копии буквы на подходящее место. Если перед маркером  $b$  расположены две буквы слова подряд, т. е. без маркеров, то эти буквы меняются местами, а курсор занимает позицию между ними. Таким образом, всякая копия буквы снабжается префиксом в виде маркера  $b$ . При этом сначала вводимая, а потом удаляемая буква  $c$  совсем не лишняя. Без неё алгоритм будет работать неверно. Нижеприведенный пример поможет понять работу этого алгоритма:

Слово	Правило
101	(6)
<u>a</u> 101	(2)
<u>1</u> b <u>1</u> <u>a</u> 01	(2)
<u>1</u> b <u>1</u> 0 <u>b</u> 0a1	(1)
<u>1</u> b0 <u>b</u> 10 <u>a</u> 1	(2)
<u>1</u> b0b1 <u>01</u> <u>b</u> 1a	(1)
<u>1</u> b0b1 <u>11</u> <u>b</u> 01a	(1)
<u>1</u> b0b1b101a	(3)
101101	

**Замечание.** Буквы  $\alpha$  и  $\beta$  в алгоритме копирования слова используются в качестве метасимволов. Они не входят в алфавит алгоритма, а лишь обозначают буквы субалфавита  $\{0, 1\}$ , сокращая запись алгоритма.

Рассмотрим еще один класс НАМ — присоединяющие алгоритмы [25].

Дописывание 0 к двоичному слову	$*0 \rightarrow 0*$ $*1 \rightarrow 1*$ $* \mapsto 0$ $\mapsto *$
Выделение первого слова в последовательности типа 100*0110	$*0 \rightarrow *$ $*1 \rightarrow *$ $* \mapsto$
Выделение последнего слова в последовательности типа 100*0110	$0* \rightarrow *$ $1* \rightarrow *$ $* \mapsto$
Выделение среднего слова в последовательности типа 1001*101#010	$0* \rightarrow *$ $1* \rightarrow *$ $#0 \rightarrow #$ $#1 \rightarrow #$ $* \mapsto$ $# \mapsto$

Высокий уровень НАМ в сравнении с МТ особенно ярко проявляется в рекурсивных правилах вычисления выражений. Приведем соответствующий алгоритм для вычисления частного класса булевых константных выражений, заданных на множестве символов  $\{\vee, \neg, \text{true}, \text{false}, (\text{, })\}$ :

$\neg\neg$	$\rightarrow$	(1)
$\neg\text{true}$	$\rightarrow \text{false}$	(2)
$\neg\text{false}$	$\rightarrow \text{true}$	(3)
$(\text{true})$	$\rightarrow \text{true}$	(4)
$(\text{false})$	$\rightarrow \text{false}$	(5)
$\text{false} \vee$	$\rightarrow$	(6)
$\vee\text{false}$	$\rightarrow$	(7)
$\text{true} \vee \text{true}$	$\rightarrow \text{true}$	(8)

Нормальные алгоритмы Маркова весьма оригинально применяются для арифметических вычислений: вместо числовых расчетов обрабатываются текстовые изображения чисел.

Во-первых, для марковской алгоритмизации работы с целыми числами полезны натуральная и кардинальная системы счисления, в которых, напомним, число 3 изображается как «|||» и «||||» соответственно. Тогда сложение операндов, разделенных \*, реализуется в одно действие (для кардинальной системы нужно удалить лишнюю палочку из суммы):

	Натуральная	Кардинальная
Алгоритм	$* \mapsto$	$  * \mapsto$
Эффект	$    *    \Rightarrow     $	$     *     \Rightarrow      $

Некоторая сложность вычитания обусловлена его алгебраичностью:

	Натуральная	Кардинальная
Алгоритм	$ *  \rightarrow *$ $* \mapsto$	$ *  \rightarrow *$ $* \mapsto  $
Эффект	$    *    \Rightarrow  $ $   *     \Rightarrow  $ $  *   \Rightarrow$	$    *    \Rightarrow   $ $   *     \Rightarrow   $ $  *   \Rightarrow  $

Этот алгоритм вычисляет модуль разности.

Если же использование в качестве разности меньшего и большего чисел модуля их разности неприемлемо, то можно выдать нулевой результат. Этот распространенный случай, видимо, происходит от машинных команд сравнения двух чисел. Если левый операнд оказывается больше правого, то результат операции является положительной разностью этих чисел, иначе — нуль. Таким образом, эта операция является булевской, поскольку нуль означает «ложь», а не нуль — «истина». Надо отметить, что именно так трактуются логические выражения в ассемблере и Си. Приведем этот алгоритм:

	Натуральная	Кардинальная
Алгоритм	$ *  \rightarrow *$ $*  \rightarrow *$ $* \mapsto$	$ *  \rightarrow *$ $*  \rightarrow *$ $* \mapsto  $
Эффект	$    *    \Rightarrow  $ $   *     \Rightarrow$ $  *   \Rightarrow$	$    *    \Rightarrow   $ $   *     \Rightarrow  $ $  *   \Rightarrow  $

После успешного обычного вычитания стирается остаток, если он оказался справа от разделителя операндов в случае, когда вычитаемое больше уменьшаемого и результат полагается равным 0. Терминирующее правило стирает сам разделитель.

Мультипликативные операции чуть сложнее, если не считать операций типа вычисления остатка от деления на 5 в натуральной системе счисления:

	Натуральная	Кардинальная
Алгоритм	$      \rightarrow$	$      \rightarrow$
Эффект	$      \Rightarrow   $	$      \Rightarrow    $

В ЭВМ операции деления и взятия остатка, как правило, неразделимы. С помощью НАМ можно реализовать такую комбинированную операцию (подразумевается, что второй операнд — снова пять палочек «|||||»):

	Натуральная	Кардинальная
Алгоритм	$*      \rightarrow  *$ $* \mapsto *$ $\mapsto *$	$*      \rightarrow  *$ $* \mapsto *$ $\mapsto  *$
Эффект	$      \Rightarrow   *   $ $    \Rightarrow *    $	$      \Rightarrow    *    $ $    \Rightarrow   *    $

Частное от деления на 5:

	Натуральная	Кардинальная
Алгоритм	$*  \cdot   \cdot   \rightarrow   *$ $*  \rightarrow *$ $* \mapsto$ $\rightarrow *$	$*  \cdot   \cdot   \rightarrow   *$ $*  \rightarrow *$ $* \mapsto$ $\rightarrow   *$
Эффект	$    \Rightarrow  $ $    \Rightarrow$	$    \Rightarrow //$ $    \Rightarrow  $

Инкремент двоичного числа (уже в позиционной системе счисления!):

$$\begin{aligned}
 >1 &\rightarrow 1> \\
 >0 &\rightarrow 0> \\
 > &\rightarrow < \\
 1< &\rightarrow <0 \\
 0< &\mapsto 1 \\
 < &\mapsto 1 \\
 &\rightarrow >
 \end{aligned}$$

Поразрядная импликация двоичного числа:

$$\begin{aligned}
 c0 &\rightarrow 0c \\
 c1 &\rightarrow 1c \\
 c &\rightarrow a \\
 0a &\rightarrow (0)b \\
 1a &\rightarrow (1)b \\
 0(0) &\rightarrow (0)0 \\
 0(1) &\rightarrow (1)0 \\
 1(0) &\rightarrow (0)1 \\
 1(1) &\rightarrow (1)1 \\
 0*(0) &\rightarrow *[1] \\
 0*(1) &\rightarrow *[1] \\
 1*(0) &\rightarrow *[0] \\
 1*(1) &\rightarrow *[1] \\
 [0]0 &\rightarrow 0[0] \\
 [0]1 &\rightarrow 1[0] \\
 [1]0 &\rightarrow 0[1] \\
 [1]1 &\rightarrow 1[1] \\
 [0]b &\rightarrow a0 \\
 [1]b &\rightarrow a1 \\
 *a &\mapsto \\
 * &\rightarrow *c
 \end{aligned}$$

Проверка числа на нечетность:

$$\begin{aligned}
 e11 &\rightarrow e1 \\
 e10 &\rightarrow e0 \\
 e01 &\rightarrow e1 \\
 e00 &\rightarrow e0
 \end{aligned}$$

$$\begin{array}{ccc} e & \mapsto & \\ & \rightarrow & e \end{array}$$

Более сложный алгоритм [20] — умножение двух чисел, заключенных в угловые скобки и разделенных звездочкой (заметьте, что здесь нет терминальных продукции!):

$$\begin{array}{ll} |>*< & \rightarrow >*<d \\ d| & \rightarrow |md \\ dm & \rightarrow md \\ d> & \rightarrow > \\ <>*< & \rightarrow <e \\ e| & \rightarrow e \\ em & \rightarrow |e \\ e> & \rightarrow > \end{array}$$

Как видно из этого примера, даже для относительно простой задачи составление НАМ оказывается весьма трудным делом, к тому же корректность такой системы вовсе неочевидна [20].

Алгоритмическая модель Маркова лежит в основе весьма интересного отечественного языка программирования рекурсивных функций РЕФАЛ, предложенного еще в 1966 году профессором Турчиным В. Ф. [81]. (Некоторая информация о РЕФАЛе находится на хрестоматийном диске.) Кроме того, развитые средства текстового поиска и замен — ядро НАМ — хорошо реализованы в современных интерпретируемых командных языках Perl, Python, PHP и в библиотеке STL.

## 2.4 Исследование алгоритмической модели Тьюринга

### 2.4.1 Теоремы Шеннона

Теоремы Шеннона позволяют выяснить, какой смысл имеют состояния головки машины Тьюринга. Из этих теорем следует, что состояния головки — это вид памяти, причем если у машины Тьюринга много состояний, то их число можно уменьшить, увеличив соответственно число букв рабочего алфавита.

**Теорема 2.4.1.** Для любой машины Тьюринга  $T = (A_p, Q, P, q_0)$  с множеством состояний  $Q = \{q_0, q_1, \dots, q_s\}$  можно эффективным образом построить машину Тьюринга  $T' = (A_r, \{\alpha, \beta\}, P', q'_0)$ , моделирующую машину  $T$  и имеющую всего два состояния:  $\alpha$  и  $\beta$ . Рабочий алфавит  $A_r$  машины  $T'$  содержит  $r = 3(p+1)(s+1) + p$  букв.

**Теорема 2.4.2.** Для любой машины Тьюринга  $T = (A_p, Q, P, q_0)$  можно эффективным образом построить машину Тьюринга  $T'' = (A_1, Q'', P'', q''_0)$ , моделирующую машину  $T$  и имеющую однобуквенный рабочий алфавит  $A_1 = \{a_1\} = \{|$ .

Произведение  $(p+1)(s+1)$  является одной из характеристик эффективности алгоритма, описываемого МТ: алгоритм тем эффективнее, чем меньше значение произведения  $(p+1)(s+1)$ . Следовательно, машины  $T'$  и  $T''$  из теорем 2.4.1 и 2.4.2 определяют менее эффективные алгоритмы, чем моделируемая машина  $T$ . Отметим также, что алгоритмы, определяемые указанными машинами  $T'$  и  $T''$ , выполняются за большее число тактов, чем алгоритм, определяемый исходной машиной  $T$  (это следует из определения 2.2.5).

Доказательство теорем Шеннона следует брошюре [4]. В частности, первая теорема сначала доказывается в авторской редакции — в терминах пятерок.

Связь между четверками и пятерками довольно проста:

$$(q, a, a', v, q') \leftrightarrow (q, a, a', q''), (q'', a', v, q')$$

$$(q, a, v, q') \leftrightarrow (q, a, v, s, q') \text{ или } (q, a, a, v, q')$$

Вообще, для любой МТ  $T_5$  в пятерках с  $k$  состояниями  $q_0, q_1, \dots, q_{k-1}$  можно эффективным образом построить моделирующую ее машину  $T_4$  в четверках с числом состояний  $3k$  и с тем же рабочим алфавитом: для каждого состояния  $q_i$  машины  $T_5$  введём два дополнительных состояния машины  $T_4$  —  $q_{i,l}, q_{i,r}$ . Для каждой команды  $(q_i, a, a', v, q_j)$  машины  $T_5$  с подвижной головкой внесём в программу машины  $T_4$  команду  $(q_i, a, a', q_{j,v})$ . Кроме того, дополним программу машины  $T_4$  командами вида  $(q_{j,v}, x, v, q_j)$  для всех  $j \in \{0, \dots, k-1\}$ ,  $v \in \{l, r\}$ ,  $x \in A_p$  ( $A_p$  — алфавит машины  $T_5$ ). Выполнение команд машины  $T_5$  разбивается на два такта: на первом такте в рабочую ячейку записывается нужная буква, а на втором происходит сдвиг и переход в конечное состояние; при этом информация о направлении сдвига и конечном состоянии хранится в промежуточном состоянии  $q_{i,v}$ .

**Замечание.** Напишите программу на Паскале или на Си, транслирующую четверки в пятерки и наоборот!

Дополнительную информацию о машинах Тьюринга с малым числом состояний можно почерпнуть из работы Шеннона [18] и в сборнике [7]. В частности, там говорится о невозможности построить машину только с одним состоянием. В связи с этим вспомните ранее приведённые примеры неостанавливающихся машин.

**Замечание.** Доказательства этих теорем были предложены величайшим учёным в области информатики и кибернетики Клодом Шеноном в 1956 году. В 2005 году студентом Рисенбергом Д. В. результат первой теоремы был усилен и количество вспомогательных букв было сокращено с  $4(p+1)(s+1) + p$  до  $3(p+1)(s+1) + p$  благодаря оригинальному способу моделирования возврата головки МТ в исходную ячейку команды. Кроме того, им же было получено доказательство первой теоремы Шеннона для машин Тьюринга со специализированными командами, приводимое далее, и его диаграммная иллюстрация, полученная методом нисходящего проектирования.

**Замечание.** (*Теоремы Шеннона в переложении для фортепиано*). Теоремы Шеннона обосновывают трансформацию (перекодирование) любых алгоритмов либо к меньшему числу состояний (этапов, стадий, в некотором смысле меток) ценой существенного увеличения рабочего алфавита, либо к очень бедному (но зато технически реализуемому!) алфавиту. Существуют некоторые простые соображения в пользу этих теорем, аналогичные кодированию любого алфавита словами из палочек. Так, известной всем морзянкой из двух собственных букв (точка, тире) и одной несобственной (пауза) можно передавать не только текстовые сообщения, но и изображения (фототелеграф), и даже закодированные аудио- и видеоданные. Голосовая связь в то время из-за несовершенства аппаратуры и линий связи была невозможна. Более современный пример: импульсная и тоновая коммутация абонентов телефонной сети. При импульсной коммутации номер 123-45-67 набирается как последовательность слов

| || ||| |||| | |||| | |||||

длины 34, а при тоновой — как последовательность аудиобукв, представляемых разночастотными звуками

$$f_{900} \oplus f_{700} \quad f_{1100} \oplus f_{700} \quad f_{1100} \oplus f_{900} \quad f_{1300} \oplus f_{700} \quad f_{1300} \oplus f_{900} \quad f_{1300} \oplus f_{1100} \quad f_{1500} \oplus f_{700}.$$

существенно меньшей длины (13).

Налицо удобство тоновой системы. Но почему же она не применялась в нашей стране? Ответ следует из второй теоремы Шеннона: обогащение алфавита недопустимо по тем временам усложнило бы абонентскую и коммутационную аппаратуру. Кстати, настоящий импульсный телефон вместо наборного диска должен иметь лишь одну кнопку. Кнопка для паузы может отсутствовать как на телеграфном ключе. Теоремы Шеннона также дают обоснование правомерности запрета использования дополнительных букв при программировании машин Тьюринга, поскольку всякая лишняя буква может быть дезавуирована несколькими дополнительными состояниями.

Вторая теорема Шеннона была применена на практике задолго до своего доказательства: Дж. фон Нейман предложил использовать двоичную систему счисления в конструкции компьютеров не только для представления данных, но и для записи программ.

## 2.4.2 Доказательство I теоремы Шеннона (К. Шенон, 1956)

▷

**2.4.2.1.** Рассмотрим какую-нибудь конфигурацию машины  $T$ :

$$C = [\lambda a_{j_1} a_{j_2} \dots a_{j_{k-1}} \quad a_{j_k} \quad a_{j_{k+1}} \dots a_{j_n} \lambda] \\ q_i$$

Команда машины  $T$ , которая должна быть применена в конфигурации  $C$ , определяется состоянием головки  $q_i$  и буквой  $a_{j_k}$ , воспринимаемой головкой. Из определения 2.1.1 следует, что если машина  $T'$  моделирует машину  $T$ , то среди конфигураций машины  $T'$  должна содержаться конфигурация  $C'$ , являющаяся образом конфигурации  $C$  и содержащая в закодированном виде информацию о паре  $(q_i, a_{j_k})$ , определяющую работу моделируемой машины. Но у машины  $T'$  всего два состояния; следовательно, в конфигурации  $C'$  головка машины  $T'$  может находиться в одном из состояний  $\alpha$  или  $\beta$ , воспринимая букву  $a' \in A_r$ . При этом информация о паре  $(q_i, a_{j_k})$  может содержаться только в букве  $a'$ . Для того чтобы это было возможно, добавим к алфавиту  $A_p$  машины  $T$   $(p+1)(s+1)$  букв  $a'$ , каждая из которых представляет собой одну из пар  $(q_i, a_{j_k})$ . Для удобства примем для буквы  $a'$ , представляющей пару  $(q_i, a_{j_k})$ , обозначение  $a' = b_{i,j_k}$ . В качестве образа конфигурации возьмем конфигурацию

$$C' = [\lambda a_{j_1} a_{j_2} \dots a_{j_{k-1}} \quad b_{i,j_k} \quad a_{j_{k+1}} \dots a_{j_n} \lambda] \\ \alpha$$

Конфигурация  $C'$  отличается от конфигурации  $C$  лишь состоянием и содержимым ячейки, воспринимаемой головкой: буква, записанная в этой ячейке, содержит информацию о паре  $(q_i, a_{j_k})$ .

**2.4.2.2.** Рассмотрим какой-нибудь торт машины  $T$ . Имеем

$$[\lambda a_{j_1} a_{j_2} \dots a_{j_{k-1}} \quad a_{j_k} \quad a_{j_{k+1}} \dots a_{j_n} \lambda] \xrightarrow{q_i} [\lambda a_{j_1} a_{j_2} \dots a_{j_{k-1}} a_{j_k'} \quad a_{j_{k+1}} \quad \dots a_{j_n} \lambda] \\ q_m \quad (2.1)$$

Согласно определению 2.1.1, такту 2.1 машины  $T$  должна соответствовать такая последовательность тактов моделирующей машины  $T'$ :

$$[\lambda a_{j_1} a_{j_2} \dots a_{j_{k-1}} b_{i,j_k} a_{j_{k+1}} \dots a_{j_n} \lambda] \xrightarrow{T'} [\lambda a_{j_1} \dots a_{j_{k-1}} a_{j_k'} b_{m,j_{k+1}} \dots a_{j_n} \lambda] \quad (2.2)$$

Но замена буквы в рабочей ячейке производится до перемещения головки, откуда следует, что если в конфигурации  $C'$  мы заменим  $b_{i,j_k}$  на  $a_{j_k}$ , то будет утеряна информация о номере следующего состояния  $q_m$ . Следовательно, информация о номере  $q_m$  должна быть сначала помещена в рабочую ячейку конфигурации  $C'$ , а потом перенесена в новую рабочую ячейку, т. е. последовательность конфигураций 2.2 должна иметь вид

$$\begin{aligned} & [\lambda a_{j_1} \dots a_{j_{k-1}} \underset{\alpha}{b_{i,j_k}} a_{j_{k+1}} \dots a_{j_n} \lambda] \xrightarrow{T'} [a_{j_1} \dots a_{j_{k-1}} b_{m,j_{k'}}^+ \underset{*}{a_{j_{k+1}}} \dots a_{j_n} \lambda] \xrightarrow{T'} \\ & \xrightarrow{\alpha} [\lambda a_{j_1} a_{j_2} \dots a_{j_{k-1}} a_{j_k'} \underset{\alpha}{b_{m,j_{k+1}}} \dots a_{j_n} \lambda] \end{aligned}$$

В последнем соотношении звездочкой (\*) обозначено одно из двух состояний машины  $T'$  (далее будет уточнено, какое именно). Символом  $b_{m,j_{k'}}^+$  обозначена еще одна дополнительная буква рабочего алфавита машины  $T'$  (она отличается от буквы  $b_{m,j_{k'}}$ ). Буква  $b_{m,j_{k'}}^+$  находится в ячейке, соседней с новой рабочей, и показывает, что информация о номере  $m$  состояния  $q_m$  должна быть перенесена из ячейки, содержащей эту букву, в рабочую ячейку. После того как перенос информации о номере состояния  $q_m$  будет завершен, буква  $b_{m,j_{k'}}^+$  должна быть заменена буквой  $a_{j_k'}$ . Следовательно, рабочий алфавит машины  $T'$  должен помимо букв  $b_{i,j_k}$ , представляющих пары  $(q_i, a_{j_k})$  в текущей рабочей ячейке, содержать еще  $(p+1)(s+1)$  букву  $b_{i,j_k}^+$ , т. е. к рабочему алфавиту  $A_p$  моделируемой машины  $T$  уже добавлено  $2(p+1)(s+1)$  букв вида  $b_{i,j_k}$  и  $b_{i,j_k}^+$ . В дальнейшем буквы  $b_{i,j_k}$  будем для симметрии обозначать  $b_{i,j_k}^-$ .

Для того чтобы воспринять букву  $b_{m,j_k}^+$ , головка должна быть помещена против ячейки, содержащей эту букву. При этом необходимо иметь информацию о том, в какую из соседних ячеек (левую или правую) необходимо перенести номер состояния  $q_m$ , заменив букву  $a_j$ , записанную в этой ячейке, буквой  $b_{m,j}^-$ . Эта информация может определять состояние головки машины  $T'$ . Состояние  $\beta$  означает, что информация о состоянии головки будет передаваться налево, а состояние  $\alpha$  — направо.

Передача информации о номере состояния  $q_m$  из одной ячейки ленты в соседнюю осуществляется машиной  $S$ , построенной Шенномоном в ходе доказательства рассматриваемой теоремы. Машина  $S$  переносит эту информацию не за один, а за несколько тактов, причем головка «качается» над ячейками, между которыми передается информация, воспринимая попеременно то одну, то другую ячейку.

Таким образом, рабочий алфавит машины  $T'$  должен содержать все буквы алфавита  $A_p$  и еще  $(p+1)(s+1)$  букв вида  $b_{i,j_k}^+$ , и  $2(p+1)(s+1)$  букв вида  $b_{i,j,x}^-$  где  $x \in \{l, r\}$  показывает местоположение источника информации.

**2.4.2.3.** Построим машину Шеннона  $S$ . Эта машина должна работать следующим образом. Пусть моделируется такт (2.1) машины  $T$ , т. е. новой рабочей ячейкой становится правая соседняя ячейка. Машина  $S$  помещает в старую рабочую ячейку букву  $b_{m,j_{k'}}^+$ , обозначающую, что номер  $m$  состояния  $q_m$  должен быть перенесен в соседнюю ячейку,

и перемещает головку направо, переводя ее в состояние  $\beta$ , которое показывает, что информация о номере состояния будет приниматься слева. Если бы информация о номере  $m$  передавалась налево, то головка была бы перемещена налево и оставлена в состоянии  $\alpha$ . Следующий такт машины  $S$  состоит в записи в новую рабочую ячейку буквы  $b_{0,j_{k+1},l}^-$ , обозначающей что информация о номере состояния будет приниматься слева при перемещении головки влево. Далее головка машины читает букву  $b_{0,j_{k+1},l}^-$ , сдвигается налево и остается в состоянии  $\alpha$ , означающем что информация будет передаваться направо. Если бы информация передавалась налево, то после записи буквы  $b_{m,j_k}^+$  в старую рабочую ячейку головка была бы перемещена налево и оставлена в состоянии  $\beta$ . В новую рабочую ячейку была бы записана буква  $b_{0,j_{k-1},r}^-$ , после чего головка была бы сдвинута направо, в старую рабочую ячейку, и оставлена в состоянии  $\beta$ . Во время «качаний» буквы  $b_{m,j_k}^+$  заменяются последовательно буквами такого же вида, но с меньшими значениями первого индекса (номер состояния уменьшается с шагом 1 до 0). Буквы  $b_{0,j_{k+1},l}^-$ , наоборот, заменяются соответствующими буквами с большими значениями первого индекса (номер состояния увеличивается с шагом 1). Через  $m$  таких «качаний» головки машины  $S$  в старой рабочей ячейке будет содержаться буква  $b_{0,j_k}^+$ , а в новой рабочей ячейке — буква  $b_{m,j_{k+1},l}^-$ , т. е. информация о номере состояния будет перенесена в новую рабочую ячейку и останется лишь заменить букву  $b_{0,j_k}^+$  буквой  $a_{j_k}$ .

Таким образом, работа машины  $S$  определяется следующей таблицей:

Буква в рабочей ячейке	Состояние $\alpha$			Состояние $\beta$		
	Записываемая буква	Действие	Новое состояние	Записываемая буква	Действие	Новое состояние
$a_i$	$b_{0,i,r}^-$	$r$	$\beta$	$b_{0,i,l}^-$	$l$	$\alpha$
$b_{j,i,l}^-$	Зависит от команды моделируемой машины			$b_{j+1,i,l}^-$	$l$	$\alpha$
$b_{j,i,r}^-$	$b_{j-1,i}^+$	$r$	$\beta$	$b_{j+1,i,r}^-$	$r$	$\beta$
$b_{j,i}^+ (j > 0)$		$r$	$\alpha$	$b_{j-1,i}^+$	$l$	$\beta$
$b_{0,i}^+$	$a_i$	$r$	$\alpha$	$a_i$	$l$	$\alpha$

Работа машины  $T'$  теперь может быть описана следующим образом.

Начальной конфигурации

$$C_0 = [\lambda a_{j_1} a_{j_2} \dots a_{j_{k-1}} \quad a_{j_k} \quad a_{j_{k+1}} \dots a_{j_m}] \quad q_0$$

машины  $T$  сопоставим конфигурацию

$$C'_0 = [\lambda a_{j_1} a_{j_2} \dots a_{j_{k-1}} \quad b_{0,j_k,x}^- \quad a_{j_{k+1}} \dots a_{j_m} \dots \lambda] \quad \alpha \quad (2.3)$$

машины  $T'$  (значение  $x$  может быть любым, так как все равно буква  $b_{0,j_k,x}^-$  будет сразу же заменена одной из букв  $b_{i,j}^+$ ).

Каждой команде машины  $T$  вида

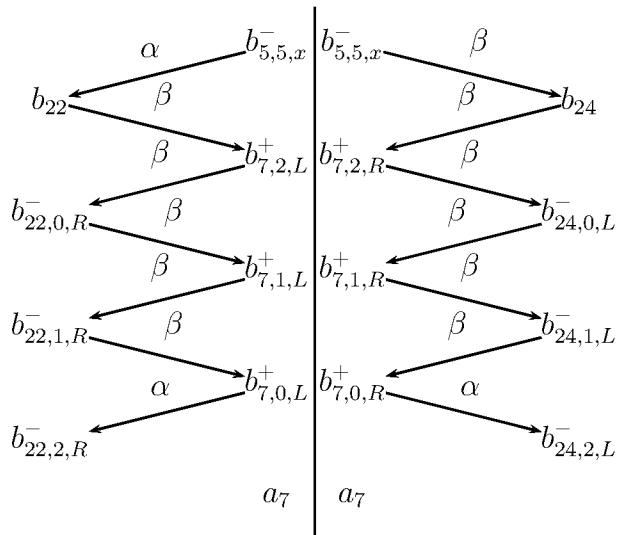
$$(q_i, a_j, a_l, r, q_m) \text{ или } (q_i, a_j, a_l, l, q_m)$$

поставим в соответствие команды машины  $T'$

$$(\alpha, b_{i,j,x}^-, b_{m,l}^+, r, \beta) \text{ и } (\alpha, b_{i,j,x}^-, b_{m,l}^+, l, \alpha) \quad (2.4)$$

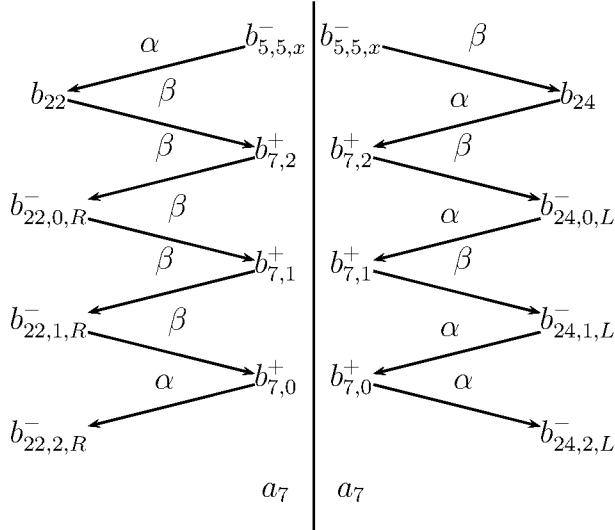
соответственно, где  $x$  пробегает множество  $\{l, r\}$ . Не рассмотренные выше команды с неподвижной головкой моделируются непосредственно без качаний: команде  $(q_i, a_j, a_l, s, q_m)$  машины  $T$  поставим в соответствие команду  $(q, b_{i,j,x}^-, b_{m,l,x}^-, s, q)$  машины  $T'$ . Каждая из команд (2.4) «запускает» машину Шеннона  $S$ , которая за  $t$  качаний реализует последовательность тактов (2.2), моделирующую торт (2.1). Легко видеть, что заключительная конфигурация машины  $T'$  будет отличаться от заключительной конфигурации машины  $T$  только тем, что в рабочей ячейке будет записана буква вида  $b_{j,i,x}^-$ , соответствующая заключительной паре  $(q_j, a_i)$  машины  $T$ . Таким образом, машина  $T'$  действительно имеет всего два состояния и моделирует машину  $T$ . Теорема доказана.  $\square$

**Замечание.** Рассмотрим пример моделирования такта классической машиной Шеннона  $S$ . Пусть в некоторый момент времени моделируемая машина должна выполнить команду  $(q_5, a_5, a_7, v, q_2)$ , где  $v = \{l, r\}$ . В терминах машины  $S$  это означает, что в текущей ячейке содержится буква  $b_{5,5,x}^-$  (индекс  $x$  для нас не важен, поскольку он принимает одно из двух значений  $\{l, r\}$  и означает, с какой стороны головка подошла к этой ячейке). Машина  $S$  находится в состоянии  $\alpha$ . Работу этой машины нам поможет проследить следующая диаграмма:



Из этой диаграммы видно, что при переносе информации о состоянии используется лишь одно состояние машины  $S$ , а именно  $\beta$ . Только вначале используется состояние  $\alpha$  для того, чтобы перенести в соседнюю ячейку информацию о том, откуда приехала головка, т. е. машина  $S$  содержит резерв, который может быть использован в целях оптимизации. Так, можно отказаться от букв вида  $b_{i,j,x}^+$ , заменив их буквами вида  $b_{i,j}^+$ , а информацию о положении целевой ячейки сохранять при помощи состояний машины  $S$ : например,  $\beta$  указывает, что головка моделируемой машины должна переместиться налево, а  $\alpha$  — направо.

Теперь перепишем пример для новой машины  $S'$ . Диаграмма из примера примет вид:



### 2.4.3 Доказательство I теоремы Шеннона (Д. Рисенберг, 2005)

▷

**2.4.2.1.** Рассмотрим какую-нибудь конфигурацию машины  $T$ :

$$C = [\lambda a_{j_1} a_{j_2} \dots a_{j_{k-1}} \underset{q_i}{a_{j_k}} a_{j_{k+1}} \dots a_{j_n} \lambda]$$

Команда машины  $T$ , которая должна быть применена в конфигурации  $C$ , определяется состоянием головки  $q_i$  и буквой  $a_{j_k}$ , воспринимаемой головкой. Из определения 2.2.5 следует, что если машина  $T'$  моделирует машину  $T$ , то среди конфигураций машины  $T'$  должна содержаться конфигурация  $C'$ , являющаяся образом конфигурации  $C$  и содержащая в закодированном виде информацию о паре  $(q_i, a_{j_k})$ , так как эта информация определяет работу моделируемой машины. Но у машины  $T'$  всего три состояния; следовательно, в конфигурации  $C'$  головка машины  $T'$  может находиться в одном из состояний  $p$ ,  $q$  или  $t$ , воспринимая букву  $a' \in A_r$ . При этом информация о паре  $(q_i, a_{j_k})$  может содержаться только в букве  $a'$ . Для того чтобы это было возможно, добавим к алфавиту  $A_p$  машины  $T$   $(p+1)(s+1)$  букв  $a'$ , каждая из которых представляет собой одну из пар  $(q_i, a_{j_k})$ . Для удобства примем для буквы  $a'$ , представляющей пару  $(q_i, a_{j_k})$ , обозначение  $a' = b_{i,j_k}$ . В качестве образа конфигурации возьмем конфигурацию

$$C' = [\lambda a_{j_1} a_{j_2} \dots a_{j_{k-1}} \underset{q}{b_{i,j_k}} a_{j_{k+1}} \dots a_{j_n} \lambda]$$

Конфигурация  $C'$  отличается от конфигурации  $C$  лишь состоянием и содержимым ячейки, воспринимаемой головкой: буква, записанная в этой ячейке, содержит информацию о паре  $(q_i, a_{j_k})$ .

**2.4.2.2.** Рассмотрим какой-нибудь торт машины  $T$ . Имеем

$$[\lambda a_{j_1} a_{j_2} \dots a_{j_{k-1}} \underset{q_i}{a_{j_k}} a_{j_{k+1}} \dots a_n \lambda] \xrightarrow{T} [\lambda a_{j_1} a_{j_2} \dots a_{j_{k-1}} a_{j_k} \underset{q_m}{a_{j_{k+1}}} \dots a_n \lambda] \quad (2.1)$$

Согласно определению 2.2.5, такту 2.1 машины  $T$  должна соответствовать следующая последовательность тактов моделирующей машины  $T'$ :

$$[\lambda a_{j_1} a_{j_2} \dots a_{j_{k-1}} b_{i,j_k} a_{j_{k+1}} \dots a_n \lambda] \xrightarrow{T'} [\lambda a_{j_1} \dots a_{j_k} b_{m,j_{k+1}} \dots a_n \lambda] \quad (2.2)$$

Но если сразу переместить головку в новую ячейку, то будет утеряна информация о номере следующего состояния  $q_m$ . Следовательно, информация о номере  $q_m$  должна быть сначала помещена в рабочую ячейку конфигурации  $C'$ , а потом перенесена в новую рабочую ячейку, т. е. последовательность конфигураций 2.2 должна иметь вид

$$\begin{aligned} & [\lambda a_{j_1} \dots a_{j_{k-1}} \ b_{i,j_k} \ a_{j_{k+1}} \dots a_{j_n} \lambda] \xrightarrow{*} [a_{j_1} \dots a_{j_{k-1}} b_{m,j_k}^+ \ a_{j_{k+1}} \dots a_{j_n} \lambda] \xrightarrow{t} \\ & \Rightarrow^* [\lambda a_{j_1} a_{j_2} \dots a_{j_{k-1}} a_{j_k} \ b_{m,j_{k+1}} \dots a_{j_n} \lambda] \end{aligned}$$

В последнем соотношении звездочкой (\*) обозначено состояние  $p$  или  $q$  машины  $T'$  (далее будет уточнено, какое именно). Символом  $b_{m,j_k}^+$  обозначена еще одна дополнительная буква рабочего алфавита машины  $T'$  (она отличается от буквы  $b_{m,j_k}$ ). Буква  $b_{m,j_k}^+$  находится в ячейке, соседней с новой рабочей, и показывает, что информация о номере  $t$  состояния  $q_m$  должна быть перенесена из ячейки, содержащей эту букву, в рабочую ячейку. После того как перенос информации о номере состояния  $q_m$  будет завершен, буква  $b_{m,j_k}^+$  должна быть заменена буквой  $a_{j_k}$ . Следовательно, рабочий алфавит машины  $T'$  должен, помимо букв  $b_{i,j_k}$ , представляющих пары  $(q_i, a_{j_k})$  в текущей рабочей ячейке, содержать еще  $(p+1)(s+1)$  букву  $b_{i,j_k}^+$ , т.е. к рабочему алфавиту  $A_p$  моделируемой машины  $T$  уже добавлено  $2(p+1)(s+1)$  букв вида  $b_{i,j_k}$  и  $b_{i,j_k}^+$ . В дальнейшем буквы  $b_{i,j_k}$  будем для симметрии обозначать  $b_{i,j_k}^-$ .

Кроме того, введем еще  $(p+1)$  букву  $b_i^-$ , которые будут указывать, что в данную ячейку будет передаваться информация, но не показывают, с какой стороны.

Для того чтобы воспринять букву  $b_{m,j_k}^+$ , головка должна быть помещена против ячейки, содержащей эту букву. При этом необходимо иметь информацию о том, в какую из соседних ячеек (левую или правую) необходимо перенести номер состояния  $q_m$ , заменив букву  $a_j$ , записанную в этой ячейке, буквой  $b_j^-$ . Эта информация может определяться либо состоянием головки машины  $T'$ , либо содержимым рабочей ячейки. Состояние  $t$  означает, что информация будет передаваться налево, а состояние  $q$  — направо. Поскольку нашей целью является минимизация числа состояний машины  $T'$ , информацию о направлении переноса номера состояния  $q_m$  будем хранить на ленте, удвоив число букв  $b_{i,j_k}^+$  и снабдив их еще одним значком, который может принимать значения  $l$  или  $r$ , указывая направление передачи информации о состоянии  $q_m$ .

Передача информации о номере состояния  $q_m$  из одной ячейки ленты в соседнюю осуществляется машиной  $S$ . Машина  $S$  переносит эту информацию не за один, а за несколько тактов, причем головка «качается» над ячейками, между которыми передается информация, воспринимая попеременно то одну, то другую ячейку. Отсюда следует, что необходимо также удвоить количество букв  $b_{i,j_k}^-$ , так как, воспринимая одну из букв, головка должна получать информацию о том, откуда переносится номер состояния  $q_m$  (слева или справа).

Таким образом, рабочий алфавит машины  $T'$  должен содержать все буквы алфавита  $A_p$ ,  $(p+1)$  букву вида  $b_j^-$  и еще  $4(p+1)(s+1)$  букв вида  $b_{i,j_k,z}^x$ , где  $x \in \{+, -\}$  показывает, является ли ячейка источником или приёмником информации, а  $z \in \{l, r\}$  показывает направление передачи или приема информации о номере состояния.

Так как машины составляются в четверках, они не могут одновременно и сдвигать головку, и заменять букву в рабочей ячейке. Поэтому для каждого типа букв надо будет

выделить состояние, отвечающее за движение, и состояние, отвечающее за запись новой буквы.

Кроме того, записывая букву  $b_j^-$  в некоторую ячейку, мы потеряем информацию о том, с какой стороны мы в нее пришли. Чтобы восстановить эту информацию, машина будет пытаться найти передающую ячейку справа от текущей. Если ей это не удастся, то передающую ячейку необходимо искать слева.

**2.4.2.3.** Построим машину  $S$ . Эта машина должна работать следующим образом. Пусть моделируется такт (2.1) машины  $T$ , т. е. новой рабочей ячейкой становится правая (левая) соседняя ячейка. Тогда машина  $T'$  выполняет следующую последовательность действий:

1. Записывает в текущей ячейке букву  $b_{m,j_k,r}^+$  ( $b_{m,j_k,l}^+$ ), переводит головку в состояние  $q$ , сдвигается направо (налево) и записывает в новой рабочей ячейке букву  $b_{j_{k+1}}^-$  ( $b_{j_{k-1}}^-$ ):

$$\begin{aligned} [\lambda a_{j_1} a_{j_2} \dots a_{j_{k-1}} & b_{i,j_k,r}^- a_{j_{k+1}} \dots a_{j_m}] \xrightarrow[p]{T'} [\lambda a_{j_1} a_{j_2} \dots a_{j_{k-1}} & b_{m,j_k,r}^+ a_{j_{k+1}} \dots a_{j_m}] \xrightarrow[q]{T'} \\ \xrightarrow[T']{} [\lambda a_{j_1} a_{j_2} \dots a_{j_{k-1}} & b_{m,j_k,r}^+ a_{j_{k+1}} \dots a_{j_m}] \xrightarrow[t]{T'} [\lambda a_{j_1} a_{j_2} \dots a_{j_{k-1}} & b_{j_{k+1}}^- \dots a_{j_m}] \end{aligned}$$

При движении налево последовательность состояний имеет следующий вид:

$$[\lambda a_{j_1} a_{j_2} \dots a_{j_{k-1}} & b_{i,j_k,r}^- a_{j_{k+1}} \dots a_{j_m}] \xrightarrow[p]{T'}^* [\lambda a_{j_1} a_{j_2} \dots & b_{j_{k-1}}^- b_{m,j_k,l}^+ a_{j_{k+1}} \dots a_{j_m}]$$

2. Информация о том, с какой стороны находится передающая ячейка, оказывается утерянной. Чтобы ее восстановить, машина сдвигает головку вправо и оставляет ее в состоянии  $p$ . Команды машины подобраны таким образом, что если справа будет найдена передающая ячейка, то головка вернется назад в состоянии  $q$ . Если же там оказывается обычная ячейка (с буквой  $a_{j_{k+1}}$ ), то головка вернется назад в состоянии  $p$ . В результате буква в принимающей ячейке будет заменена на новую, показывающую, с какой стороны передается информация:

$$\begin{aligned} [\lambda a_{j_1} a_{j_2} \dots a_{j_{k-1}} & b_{m,j_k,r}^+ b_{j_{k+1}}^- \dots a_{j_m}] \xrightarrow[t]{T'} [\lambda a_{j_1} a_{j_2} \dots b_{m,j_k,r}^+ b_{j_{k+1}}^- & a_{j_{k+2}} \dots a_{j_m}] \xrightarrow[p]{T'} \\ \xrightarrow[T']{} [\lambda a_{j_1} a_{j_2} \dots b_{m,j_k,r}^+ & b_{j_{k+1}}^- a_{j_{k+2}} \dots a_{j_m}] \xrightarrow[t]{T'} [\lambda a_{j_1} a_{j_2} \dots b_{m,j_k,r}^+ & b_{0,j_{k+1},l}^- a_{j_{k+2}} \dots a_{j_m}] \\ \left( \begin{aligned} [\lambda a_{j_1} a_{j_2} \dots & b_{j_{k-1}}^- b_{m,j_k,l}^+ a_{j_{k+1}} \dots a_{j_m}] \xrightarrow[T']{} [\lambda a_{j_1} a_{j_2} \dots b_{j_{k-1}}^- & b_{m,j_k,l}^+ a_{j_{k+1}} \dots a_{j_m}] \xrightarrow[p]{T'} \\ \xrightarrow[q]{T'} [\lambda a_{j_1} a_{j_2} \dots & b_{j_{k-1}}^- b_{m,j_k,l}^+ a_{j_{k+1}} \dots a_{j_m}] \xrightarrow[t]{T'} [\lambda a_{j_1} a_{j_2} \dots & b_{0,j_{k-1},r}^- b_{m,j_k,l}^+ a_{j_{k+1}} \dots a_{j_m}] \end{aligned} \right) \end{aligned}$$

3. Теперь можно начать перенос информации о номере состояния машины путем качаний головки.

Во время качаний буквы  $b_{m,j_k,r}^+$  ( $b_{m,j_k,l}^+$ ) заменяются последовательно буквами такого же вида, но с меньшими значениями первого индекса (номер состояния уменьшается с шагом 1 до 0). Буквы  $b_{0,j_{k+1},l}^-$  ( $b_{0,j_{k-1},r}^-$ ), наоборот, заменяются соответствующими буквами с большими значениями первого индекса (номер состояния увеличивается с шагом 1).

$$\begin{aligned}
 & [\lambda a_{j_1} a_{j_2} \dots a_{j_{k-1}} \ b_{m,j_k,r}^+ \ b_{0,j_{k+1},l}^- \dots a_{j_m}] \xrightarrow[t]{T'} [\lambda a_{j_1} a_{j_2} \dots a_{j_{k-1}} \ b_{m-1,j_k,r}^+ \ b_{0,j_{k+1},l}^- \dots a_{j_m}] \xrightarrow[p]{T'} \\
 & \Rightarrow [\lambda a_{j_1} a_{j_2} \dots a_{j_{k-1}} b_{m-1,j_k,r}^+ \ b_{0,j_{k+1},l}^- \dots a_{j_m}] \xrightarrow[p]{T'} [\lambda a_{j_1} a_{j_2} \dots a_{j_{k-1}} b_{m-1,j_k,r}^+ \ b_{1,j_{k+1},l}^- \dots a_{j_m}] \xrightarrow[t]{T'} \\
 & \Rightarrow [\lambda a_{j_1} a_{j_2} \dots a_{j_{k-1}} b_{m-1,j_k,r}^+ \ b_{1,j_{k+1},l}^- \dots a_{j_m}] \\
 & \left( [\lambda a_{j_1} a_{j_2} \dots b_{0,j_{k-1},r}^- \ b_{m,j_k,l}^+ \ a_{j_{k+1}} \dots a_{j_m}] \xrightarrow[t]{T'} [\lambda a_{j_1} a_{j_2} \dots b_{1,j_{k-1},r}^- \ b_{m-1,j_k,l}^+ \ a_{j_{k+1}} \dots a_{j_m}] \right)
 \end{aligned}$$

Через  $t$  таких качаний головки машины  $S$  в старой рабочей ячейке будет содержаться буква  $b_{0,j_k,r}^+$  ( $b_{0,j_k,l}^+$ ), а в новой рабочей ячейке — буква  $b_{m,j_{k+1},l}^-$  ( $b_{m,j_{k-1},r}^-$ ), т. е. информация о номере состояния будет перенесена в новую рабочую ячейку. Останется лишь заменить букву  $b_{0,j_k,r}^+$  ( $b_{0,j_k,l}^+$ ) буквой  $a_{j_k}$  и перейти в новую рабочую ячейку в состоянии, означающем что качания завершились.

$$\begin{aligned}
 & [\lambda a_{j_1} a_{j_2} \dots a_{j_{k-1}} \ b_{0,j_k,r}^+ \ b_{m,j_{k+1},l}^- \dots a_{j_m}] \xrightarrow[t]{T'} [\lambda a_{j_1} a_{j_2} \dots a_{j_{k-1}} a_{j_k} \ b_{m,j_{k+1},l}^+ \dots a_{j_m}] \\
 & \left( [\lambda a_{j_1} a_{j_2} \dots b_{m,j_{k-1},r}^- \ b_{0,j_k,l}^+ \ a_{j_{k+1}} \dots a_{j_m}] \xrightarrow[t]{T'} [\lambda a_{j_1} a_{j_2} \dots b_{m,j_{k-1},r}^- \ a_{j_k} a_{j_{k+1}} \dots a_{j_m}] \right)
 \end{aligned}$$

Таким образом, работа машины  $S$  определяется следующей таблицей:

Буква в рабо- чей ячейке	состояние $p$		состояние $q$		состояние $t$	
	записы- ваемая буква или действие	новое состоя- ние	записы- ваемая буква или действие	новое состоя- ние	записы- ваемая буква или действие	новое состоя- ние
$a_i$	$l$	$p$	$r$	$q$	$b_i^-$	$t$
$b_i^-$	$b_{i,0,l}^-$	$t$	$b_{i,0,r}^-$	$t$	$r$	$p$
$b_{j,i,r}^+$	$r$	$p$	$r$	$t$	$b_{j-1,i,r}^+ \ (j > 1)$	$p$
					$a_i \ (j = 1)$	$q$
$b_{j,i,l}^+$	$l$	$q$	$l$	$t$	$b_{j-1,i,l}^+ \ (j > 1)$	$p$
					$a_i \ (j = 1)$	$p$
$b_{j,i,l}^-$	$b_{j+1,i,l}^-$	$t$	зависит от команды моделируемой машины		$l$	$t$
$b_{j,i,r}^-$	зависит от команды моделируемой машины		$b_{j+1,i,r}^-$	$t$	$r$	$t$

Работа машины  $T'$  теперь может быть описана следующим образом.  
Начальной конфигурации

$$C_0 = [\lambda a_{j_1} a_{j_2} \dots \quad a_{j_k} \quad \dots a_{j_m}] \\ q_0$$

машины  $T$  сопоставим конфигурацию

$$C'_0 = [\lambda a_{j_1} a_{j_2} \dots \quad b_{0,j_k,l}^- \quad \dots a_{j_m} \dots \lambda] \\ q \quad (2.3)$$

машины  $T'$ .

Каждой команде машины  $T$  вида

$$(q_i, a_j, r, q_m) \text{ или } (q_i, a_j, l, q_m)$$

поставим в соответствие пару команд машины  $T'$

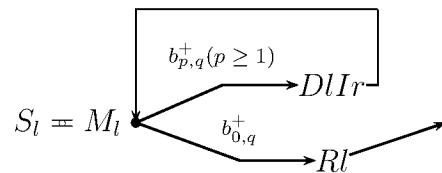
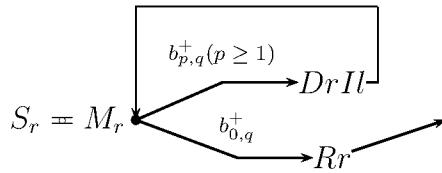
$$(q, b_{i,j,l}^-, b_{m,j,r}^+, q), (p, b_{i,j,r}^-, b_{m,j,r}^+, q) \\ \text{и } (p, b_{i,j,r}^-, b_{m,j,r}^+, q), (q, b_{i,j,l}^-, b_{m,j,l}^+, q) \quad (2.4)$$

соответственно. Какая именно из двух команд будет выполнена, определяется предыдущим тактом машины. Каждая из команд (2.4) запускает машину  $S$ , которая за  $t$  качаний реализует последовательность тактов (2.2), моделирующую такт (2.1).

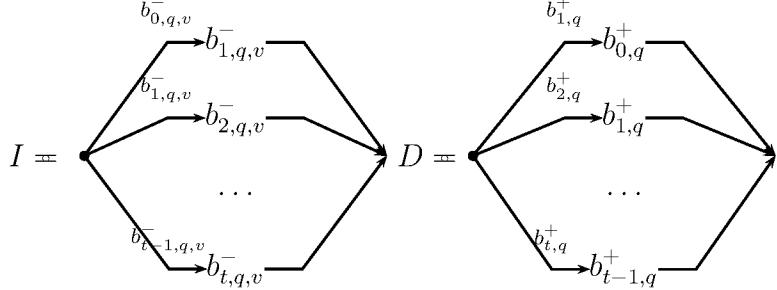
Командам машины  $T$  с неподвижной головкой соответствуют пары команд машины  $T'$ : команде  $(q_i, a_j, a_l, q_m)$  соответствуют команды  $(p, b_{j,i,r}^-, b_{m,l,r}^-, p)$  и  $(q, b_{j,i,l}^-, b_{m,l,l}^-, q)$ . Легко видеть, что заключительная конфигурация машины  $T'$  будет отличаться от заключительной конфигурации машины  $T$  только тем, что в рабочей ячейке будет записана буква вида  $b_{j,i,x}^-$ , соответствующая заключительной паре  $(q_j, a_i)$  машины  $T$ . Таким образом, машина  $T'$  действительно имеет всего три состояния и моделирует машину  $T$ . Теорема 2.4.1 доказана.  $\square$

Машину Шеннона  $S$  можно описать через диаграмму. Для этого разобьем команды исходной машины на три вида:

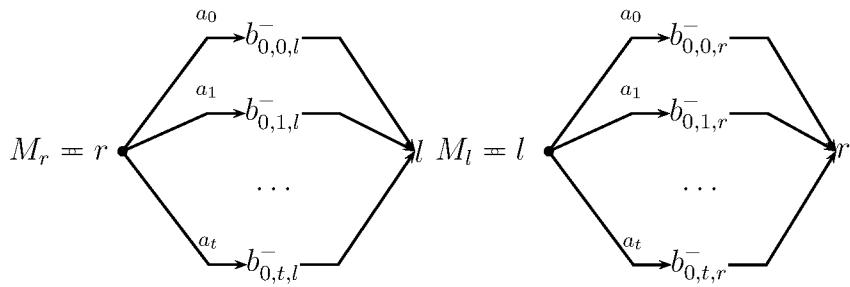
1. Команды с подвижной головкой, т. е. команды вида  $(q_i, a_j, a_l, r, q_m)$  и  $(q_i, a_j, a_l, l, q_m)$ . Им будут соответствовать машины  $B_{i,j} = b_{m,l}^+ S_r$  и  $B_{i,j} = b_{m,l}^+ S_l$  описываемые следующими диаграммами:



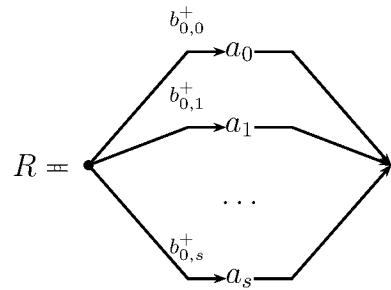
В этих диаграммах были использованы вспомогательные машины, которые также следует описать. Машины  $I$  и  $D$  выполняют инкремент и декремент — увеличивают или уменьшают значение правого индекса буквы в рабочей ячейке.



Машины  $M_r$  и  $M_l$  помечают ту ячейку, в которую на данном шаге будет передаваться информация, и возвращают головку назад.



Наконец, машины  $R$  восстанавливают в рабочей ячейке знак из алфавита машины  $T$ .



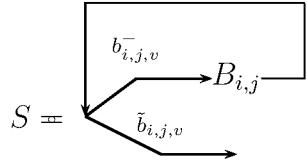
Всюду здесь стрелки, помеченные буквами вида  $b_{p,q}$ , являются на самом деле семействами стрелок по одной для каждого  $p = 0, 1, \dots, t$  и  $q = 0, 1, \dots, s$ , но т. к. переход по любой из таких стрелок равнозначен с точки зрения выполняемых дальнейших действий, он были объединены в одну для наглядности.

2. Командам с неподвижной головкой  $(q_i, a_j, a_l, s, q_m)$  соответствуют машины

$$B_{i,j} = \cdot b_{m,l,r}^- \cdot .$$

3. Терминалные команды машины  $T$  (команды вида  $(q_i, a_j, a_j, s, q_i)$ ) представляют собой особый случай, и им не ставится в соответствие никакие машины.

Теперь можно составить диаграмму машины  $S$ :



Под  $\tilde{b}_{i,j,v}$  здесь подразумеваются все такие буквы алфавита машины  $S$ , что пара  $(q_i, a_j)$  определяет терминальную команду машины  $T$  (именно поэтому им не была поставлена в соответствие никакая машина). Остальные буквы отмечены как  $b_{i,j,v}^-$ .

Машина  $S$ , определяемая полученной диаграммой, моделирует машину Шеннона (достаточно сравнить способы их построения), а следовательно, и исходную машину  $T$ .

**Замечание.** Сама по себе диаграмма не может являться доказательством теоремы, т. к. из нее не следует, что описываемая ей машина может иметь всего два состояния, но она служит хорошей иллюстрацией работы машины Шеннона, построенной в доказательстве.

**Замечание.** Диаграмма машины  $S$  является схемой. Это легко видеть, т. к. структурными являются машины  $I, D, M_l, M_r, R, S_l, S_r$  и, соответственно, машины  $B_{i,j}$ . Структурность машины  $S$  отражает структурность алгоритма, с помощью которого мы строили эту машину. Его можно описать так: пока не встреченна буква, соответствующая терминальной команде, моделировать следующий такт машины  $T$  (соответствует внешнему циклу на диаграмме  $S$ ). Описание моделирования отдельного такта (за исключением подготовительных и завершающих действий) тоже «циклическо»: пока первый индекс буквы в передающей ячейке не равен нулю, увеличить первый индекс буквы в принимающей ячейке и уменьшить первый индекс буквы в передающей ячейке на 1.

## Лекция 9

### 2.4.4 Доказательство теоремы 2.4.2

Применим диаграммы Тьюринга к доказательству второй теоремы Шеннона.

Для доказательства теоремы 2.4.2 необходимо построить машину  $T_1$  с рабочим алфавитом  $A_1 = \{\lambda\}$ , моделирующую работу машины  $T$  с рабочим алфавитом  $A_p$  ( $p > 1$ ).

Сначала установим соответствие между конфигурацией исходной машины  $T$  и моделирующей машины  $T_1$ . Кодирование букв алфавита  $\bar{A}_p$  над алфавитом  $\bar{A}_1$  будем производить способом, указанным в утверждении 1.4.1: в качестве кода буквы  $a_j \in \bar{A}_p$  ( $j = 0, 1, \dots, p$ ) будем брать слово  $a'_j = a_0 \underbrace{a_1 a_1 \dots a_1}_{j+1} \in \bar{A}_1^*$ . Для наглядности будем обозначать  $a_0$  через  $\lambda$ , а букву  $a_1$  через  $|$ , так что  $a'_j = \lambda \underbrace{| \dots |}_{j+1}$ . Ситуации машины

$T [\lambda a_{i_1} a_{i_2} \dots a_{i_{k-1}} (a_{i_k}) a_{i_{k+1}} \dots a_{i_m} \lambda] >$ ,  $a_j \in \bar{A}_p$ ,  $j = i_1, i_2, \dots, i_m$  поставим в соответствие

следующую ситуацию машины  $T_1$ :

$$[\lambda\lambda|\lambda \underbrace{\parallel \dots \parallel}_{i_1+1} \lambda \underbrace{\parallel \dots \parallel}_{i_2+1} \dots \lambda \underbrace{\parallel \dots \parallel}_{i_{k-1}+1} (\lambda) \underbrace{\parallel \dots \parallel}_{i_k+1} \lambda \underbrace{\parallel \dots \parallel}_{i_{k+1}+1} \dots \lambda \underbrace{\parallel \dots \parallel}_{i_m+1} \lambda|\lambda >$$

Соответствие между ситуациями позволяет установить и соответствие между конфигурациями.

Диаграмму машины  $T_1$  получим из диаграммы исходной машины  $T$  следующим образом. Сначала приведем диаграмму машины  $T$  к виду, в котором она содержит только символы элементарных машин (такое преобразование уже рассматривалось при доказательстве теоремы 2.4.2). В полученной диаграмме заменим символы элементарных машин  $r, l, a_i, \lambda (i = 1, 2, \dots, p)$  и стрелки диаграммами машин  $M_r, M_l, M_{a_i}, M_\lambda, M_\rightarrow$  с таким расчетом, чтобы полученная диаграмма описывала машину, моделирующую исходную машину  $T$ . Перейдем к построению машин  $M_r, M_l, M_{a_i}, M_\rightarrow$  и  $M_\lambda$ .

Машина  $M_r$  должна моделировать работу элементарной машины  $r$ , которая, как известно, переводит ситуацию

$$[\lambda a_{i_1} a_{i_2} \dots a_{i_{k-1}} (a_{i_k}) a_{i_{k+1}} \dots a_{i_m} \lambda] \xrightarrow{r} >$$

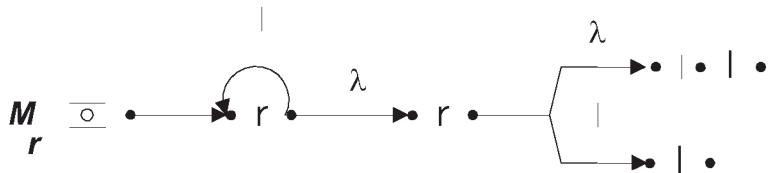
в ситуацию

$$[\lambda a_{i_1} a_{i_2} \dots a_{i_{k-1}} a_{i_k} (a_{i_{k+1}}) a_{i_{k+2}} \dots a_{i_m} \lambda] >$$

Следовательно, машина  $M_r$  должна осуществлять такую последовательность тактов:

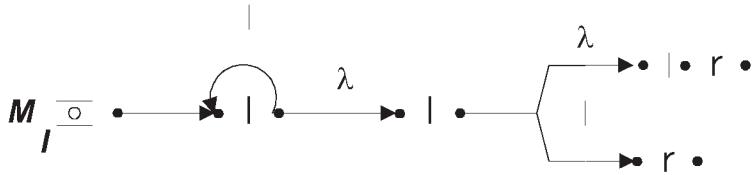
$$\begin{aligned} & [\lambda a'_{i_1} a'_{i_2} \dots a'_{i_{k-1}} (\lambda) \underbrace{\parallel \dots \parallel}_{i_k+1} \lambda \underbrace{\parallel \dots \parallel}_{i_{k+1}+1} \dots a'_{i_m} \lambda] \xrightarrow{M_r} & \\ & \xrightarrow{M_r} [\lambda a'_{i_1} a'_{i_2} \dots a'_{i_{k-1}} \lambda \underbrace{\parallel \dots \parallel}_{i_k+1} (\lambda) \underbrace{\parallel \dots \parallel}_{i_{k+1}+1} \dots a'_{i_m} \lambda] & \end{aligned}$$

Таким образом, диаграмма машины  $M_r$ , являясь вариантом машины  $\mathbf{R}$ , должна иметь следующий вид:



т.е. машина  $M_r$  осуществляет сдвиг головки на слово над алфавитом  $A_1$  (оно кодирует букву на ленте исходной машины), а затем просматривает следующую букву на ленте: если она отличается от  $\lambda$ , т.е. следующая буква является кодом какой-то буквы из  $\bar{A}_p$ , то головка возвращается назад, чтобы начало кода буквы (символ  $\lambda$ ) попало в рабочую ячейку; если же она равна  $\lambda$ , то эта буква заменяется  $\lambda'$  (здесь мы учитываем, что лента бесконечна и мы не можем заранее заменить все буквы  $\lambda$  их кодами  $\lambda'$ ).

Машина  $M_l$  строится аналогично машине  $M_r$ . Она имеет диаграмму, аналогичную машине  $\mathbf{L}$ :



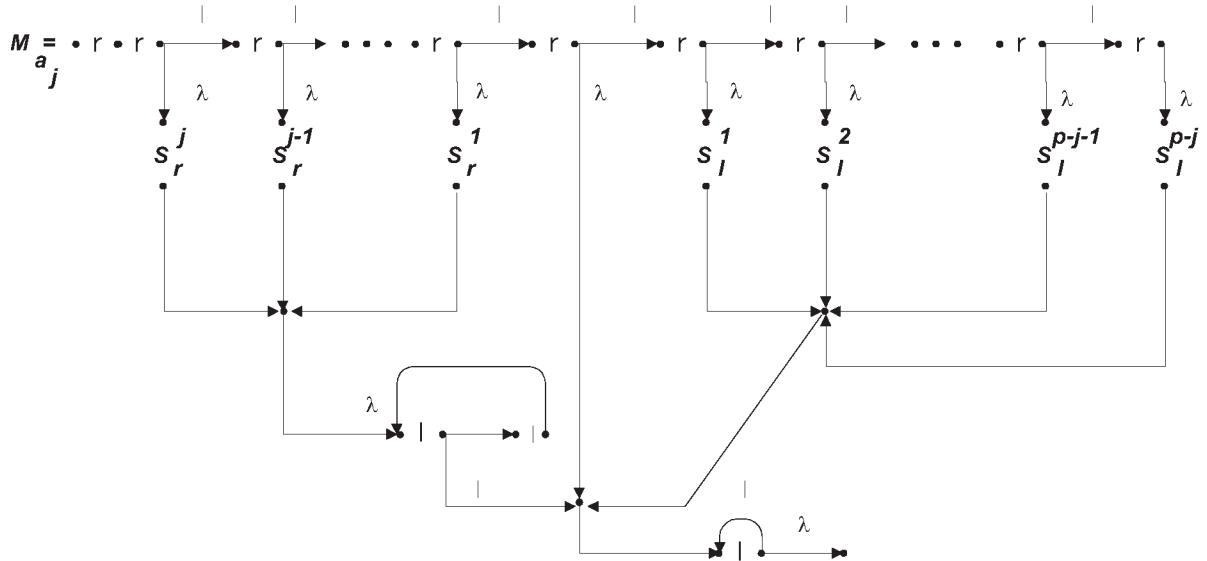
Машине  $M_\lambda$  должна заменять  $a_{i_k}$  на  $a_i$  или на  $\lambda$ .

Машине  $M_{a_j}$  должна моделировать работу элементарной машины  $a_j$ , т. е. осуществлять такую последовательность тактов:

$$\begin{aligned}
 & [\lambda a'_{i_1} a'_{i_2} \dots a'_{i_{k-1}} (\lambda) \underbrace{\dots}_{i_k+1} \lambda \underbrace{\dots}_{i_{k+1}+1} \dots a'_{i_m} \lambda] \xrightarrow{M_{a_j}}^* \\
 & \xrightarrow{M_{a_j}}^* [\lambda a'_{i_1} a'_{i_2} \dots a'_{i_{k-1}} (\lambda) \underbrace{\dots}_{j+1} \lambda \underbrace{\dots}_{i_{k+1}+1} \dots a'_{i_m} \lambda]
 \end{aligned}$$

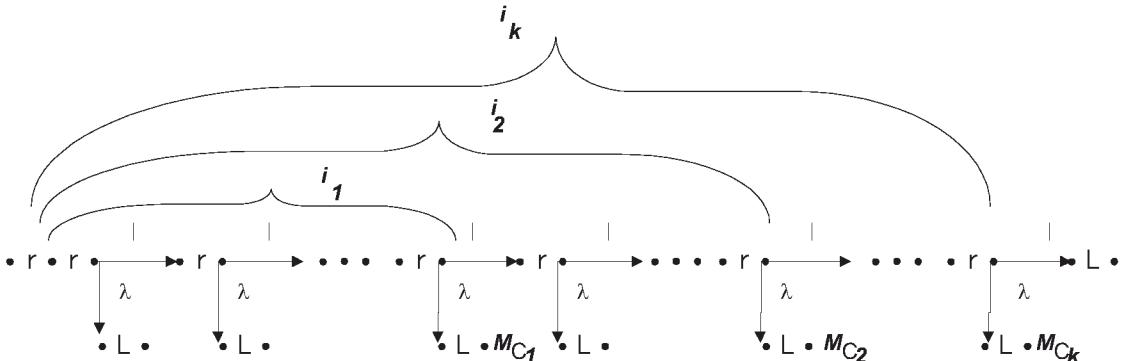
Таким образом, машина  $M_{a_i}$  должна изменять количество палочек в одном слове, не меняя остальной последовательности слов на ленте. Частные случаи машины  $M_{a_j}$ , когда  $j = i_{k\pm 1}$ , были рассмотрены в примерах 3.4.2 и 3.4.3. При построении машины  $M_{a_j}$  будут использованы машины  $S_r$  и  $S_l$ , построенные в этих примерах.

Машине  $M_{a_j}$  должна заменить на ленте кодовое слово  $a'_{i_k}$  словом  $a'_j$ , не меняя остальных слов. Для этого машина  $M_{a_j}$  должна подсчитать количество палочек в слове  $a_{i_k}$  и, если  $i_k < j$ , добавить к слову  $a'_{i_k}$   $j - i_k$  палочек, если же  $i_k > j$ , стереть в слове  $a'_{i_k}$   $i_k - j$  палочек. Эти действия может осуществить машина с диаграммой:



Последовательность машин  $r$  в верхней части диаграммы представляет собой счетчик палочек в слове  $a'_{i_k}$  (оно может содержать от одной до  $p + 1$  палочек, так как является кодом буквы  $a_{i_k} \in A_p$ ). Если  $i_k < j$ , то  $(j - i_k)$  раз включается машина  $S_r$ , в результате чего на ленте высвобождается место для  $j - i_k$  палочек; если  $i_k > j$ , то  $(i_k - j)$  раз включается машина  $S_l$ , в результате чего стирается  $i_k - j$  палочек. После того как слово  $a'_{i_k}$  превращено в слово  $a'_j$ , выполняется машина  $L$ , устанавливающая головку на начало слова  $a'_j$ . Таким образом, машина  $M_{a_j}$  действительно заменяет код буквы  $a_{i_k}$  кодом буквы  $a_j$ .

Машину  $M_{\rightarrow}$ . Пусть на диаграмме исходной машины  $T$  символ  $C$  был соединен с символами  $C_1, C_2, \dots, C_k$  стрелками вида  $\xrightarrow{a_{i_1}}, \xrightarrow{a_{i_2}}, \dots, \xrightarrow{a_{i_k}}$  соответственно. Символам  $C_1, C_2, \dots, C_k$  на диаграмме машины  $T_1$  соответствуют символы  $M_C, M_{C_1}, \dots, M_{C_k}$ . Эти символы соединяются между собой следующим образом (для определенности мы предполагаем, что  $i_1 < i_2 < \dots < i_k$ ):



Здесь последовательность букв  $r$  снова представляет собой счетчик палочек в коде буквы, которая у исходной машины расположена в рабочей ячейке. Узнав, какая буква в рабочей ячейке, мы возвращаемся к ее началу (машина  $L$ ) и переходим, если это нужно, к одной из машин  $M_{C_1}, M_{C_2}, \dots, M_{C_k}$ .

Анализ работы машин  $M_r, M_l, M_{a_j}$  ( $j = 0, 1, \dots, p$ ),  $M_{\rightarrow}$  показывает, что машина  $T_1$ , определяемая диаграммой, полученной из диаграммы машины  $T$  заменой символов  $r, l, a_j$  и стрелок символами  $M_r, M_l, M_{a_j}$  и  $M$  соответственно, действительно моделирует машину  $T$  над алфавитом  $A_1$ . Теорема доказана.

## Лекция 10

### 2.5 Вычислимые функции

#### 2.5.1 Понятие функции на множестве слов

Пусть  $A_s$  и  $A_t$  — два алфавита и пусть  $L \subseteq A_s^*$ . **Функцией** из множества  $L$  в множество  $A_t^*$  называется правило, которое каждому слову  $u \in L$  ставит в соответствие единственное слово  $w \in A_t^*$ , называемое значением функции  $f$  на слове  $u$  (это значение мы будем обозначать  $w = f(u)$ ). Множество  $L$  называется **областью определения** функции  $f$  и обозначается  $Def(f)$ . Множество всех  $w \in A_t^*$ , являющихся значениями функции  $f$  на всевозможных словах  $u \in Def(f)$ , называется **множеством значений** функции  $f$  и обозначается  $Im(f)$ . Функция  $f$  определяет **отображение**  $f : L \rightarrow A_t^*$  множества  $L$  в множество  $A_t^*$ . При этом  $w = f(u)$  является **образом слова**  $u$ , а множество  $Im(f)$  — **образом множества**  $L$  при отображении  $f$ .

Мы определили функцию одной переменной, или **одноместную** функцию. Для определения  $n$ -местной функции напомним понятие декартова произведения. **Декартово произведение**  $M \times N$  двух множеств  $M$  и  $N$  определяется как множество всех упорядоченных пар  $(m, n)$ , где  $m \in M, n \in N$ . В частности, если  $M = N$ , то декартово произведение  $M \times M = M^2$  представляет собой множество всех упорядоченных пар элементов множества  $M$ . Аналогично множество  $M^3 = M^2 \times M = M \times M^2$  можно определить как множество всех

упорядоченных троек элементов множества  $M$  и т.д. Таким образом,  $M^n$  определяется по индукции как множество всевозможных упорядоченных  $n$ -ок элементов множества  $M$ .

$n$ -местная функция  $f$  определяется как *отображение*  $f: L \rightarrow A_t^*$ , где  $L \subseteq (A_s^*)^n$ . Таким образом,  $n$ -местная функция  $f$  задается правилом, которое каждому элементу  $X \in L \subseteq (A_s^*)^n$  (здесь  $X$  — последовательность из  $n$  слов  $u_1, u_2, \dots, u_n$  над алфавитом  $A_s$ ) ставит в соответствие слово  $w \in A_t^*$ . Значение  $w$   $n$ -местной функции  $f$  на последовательности слов  $X = (u_1, u_2, \dots, u_n)$  обозначается  $w = f(u_1, u_2, \dots, u_n)$ . Область определения  $Def(f)$  и множество значений  $Im(f)$   $n$ -местной функции  $f$  определяются так же, как и для одноместной функции.

## 2.5.2 Понятие вычислимости

### 2.5.3 Функции, вычислимые по Тьюрингу

$n$ -местная функция  $f: L \rightarrow A_t^*$ ,  $L \subseteq (A_s^*)^n$  может быть определена с помощью машины Тьюринга. Пусть  $T$  — машина Тьюринга с рабочим алфавитом  $A_p$  таким, что  $A_s \subseteq A_p$ ,  $A_t \subseteq A_p$ .  $n$ -местная функция  $f_T$  может быть определена следующим образом: последовательность  $X = (u_1, u_2, \dots, u_n) \in (A_s^*)^n$  принадлежит  $Def(f_T)$  тогда и только тогда, когда машина  $T$ , примененная в ситуации  $[z\lambda u_1\lambda u_2\dots\lambda u_n(\lambda)\lambda]$ , останавливается в ситуации  $[\lambda z\lambda w(\lambda)\lambda]$ , где  $z \in \bar{A}_p^*$ ,  $w \in A_t^*$ ; в этом случае слово  $w \in A_t^*$  по определению является значением функции  $f_T$  на рассматриваемой последовательности  $X$ :  $w = f_T(u_1, u_2, \dots, u_n)$ . Таким образом, МТ  $T$  соответствует функция  $f_T$ , определяемая этой машиной. Это позволяет описать действие машины  $T$ , указав функцию  $f_T$ . С другой стороны, применяя МТ к различным сообщениям, можно узнать, какие сообщения задают функцию, а какие — нет.

**Определение 2.5.1.** Функция  $f: (A_s^*)^n \rightarrow A_t^*$  называется **вычислимой по Тьюрингу** (ВТ-функцией), если существует МТ с рабочим алфавитом  $A_p$ , содержащим алфавиты  $A_s$  и  $A_t$  (в силу принятого соглашения об алфавитах (см. п.2.6.1) для этого достаточно, чтобы  $p = \max(s, t)$ ) такая, что функция  $f_T : (A_s^*)^n \rightarrow A_t^*$ , определяемая этой МТ, совпадает с  $f$  ( $f_T \equiv f$ ). О любой такой МТ говорят, что она вычисляет  $f$ .

Из определения 2.4.3 в частности, следует, что если МТ  $T'$  моделирует машину  $T$ , то эти машины вычисляют одну и ту же функцию  $f$ .

Класс ВТ-функций определяет границы возможностей программирования: для невычислимых функций составление алгоритма невозможно!

### 2.5.4 Нормированные вычисления

**Определение 2.5.2.** Функция  $f: (A_s^*)^n \rightarrow A_t^*$  называется **нормированно вычислимой по Тьюрингу** (НВТ-функцией), если существует МТ  $T_f$ , которая вычисляет  $f$  и при этом удовлетворяет следующим требованиям:

1. если  $X = (u_1, u_2, \dots, u_n) \notin Def(f)$ , то машина  $T_f$  после применения к  $X$  никогда не останавливается;
2. если  $X = (u_1, u_2, \dots, u_n) \in Def(f)$ , то  $T_f$  вычисляет значение функции  $[\lambda z\lambda u_1\lambda u_2\dots\lambda u_n(\lambda)\lambda] \xrightarrow{T_f}^* [\lambda z\lambda u_1\lambda u_2\dots\lambda u_n\lambda f(u_1, u_2, \dots, u_n)(\lambda)\lambda]$ ,  $z \in \bar{A}_p^*$  и останавливается.

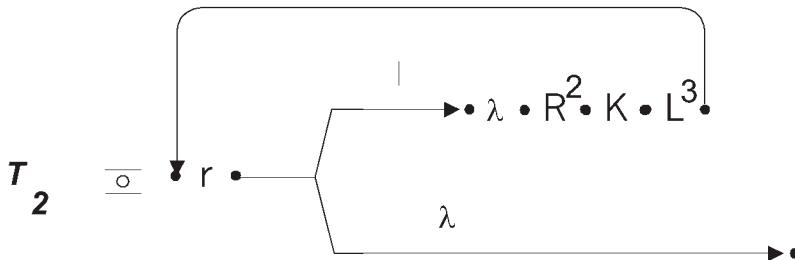
Смысл нормированного вычисления состоит в том, что при таком вычислении часть ленты, расположенная левее символа  $\lambda$ , непосредственно предшествующего  $u_1$ , не меняется и не влияет на работу машины  $T_f$ . При нормированном вычислении головка не должна заходить левее указанного символа  $\lambda$ .

**Пример 2.5.3.** Пусть  $u$  и  $w$  — два непустых слова над алфавитом  $A = \{|\} : u \in \{|\}^*, w \in \{|\}^*$ . Эти слова можно считать изображениями натуральных чисел, сопоставив натуральному числу  $n$  слово над  $\{|\}^*$ , содержащее  $n$  палочек:  $\varphi(\underbrace{|| \dots |}_n) = n$ .

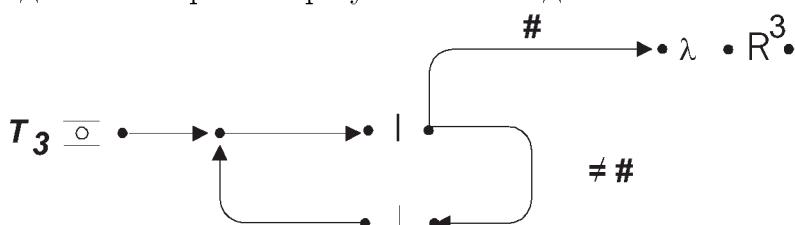
Построим МТ  $P$ , реализующую нормированное вычисление функции  $f(u, w) = uw$  (через  $uw$  обозначено произведение натуральных чисел, изображаемых словами  $u$  и  $w$ ). Действие машины  $P$  можно описать следующим образом:

$$[\lambda z \lambda u \lambda w (\lambda) \lambda] \xrightarrow{P} [\lambda z \lambda u \lambda w \lambda u w (\lambda) \lambda]$$

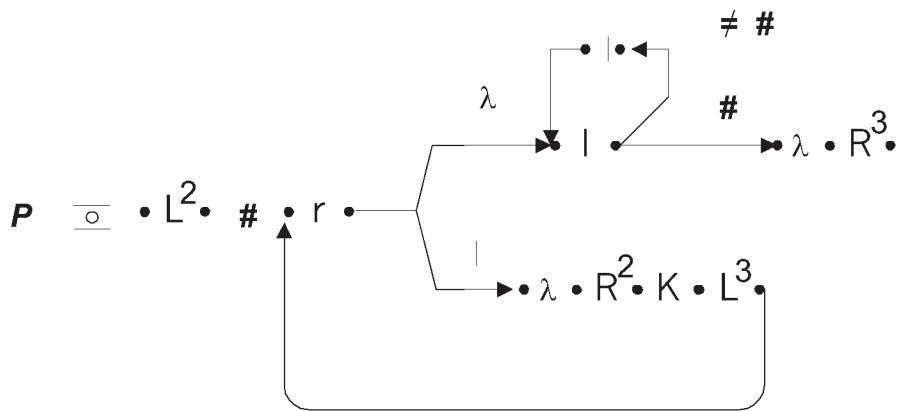
При вычислении  $uw$  машина  $P$  не должна заходить в ячейки части ленты, обозначенной  $[\lambda z]$ , и в частности не должна менять записи в этой части. Для этого мы поместим непосредственно слева от слова  $u$  специальный символ  $\#$  (он будет обозначать границу доступной части ленты). Это можно сделать, применив машину  $T_1 = L^2\#$ . Далее будем просматривать слово  $u$  и копировать слово  $w$  столько раз подряд, сколько палочек содержится в слове  $u$ . Это действие может быть реализовано машиной



Машина  $T_2$  стирает слово  $u$ , поэтому после ее работы необходимо восстановить это слово, а также убрать символ  $\#$ , заменив его несобственной буквой  $\lambda$ , и поместить головку непосредственно справа от результата. Эти действия помогут выполнить машина



Объединяя диаграммы машин  $T_1, T_2, T_3$ , получил диаграмму машины  $P$ , реализующей нормированное вычисление функции  $f(u, w) = uw$ :



Всякая НВТ-функция является в то же время и ВТ-функцией. Это следует из определения 2.4.4. Оказывается, что верно и обратное утверждение, т. е. имеет место следующая важная

**Теорема 2.5.4.** Всякая ВТ-функция является НВТ-функцией, причем соответствующая МТ не использует никаких вспомогательных букв.

Доказательство этой теоремы будет приведено в п. 2.7.4, так как машину  $T_N$ , которая нормированно вычисляет ту же функцию  $f_T$ , что и машина  $T$ , удобно строить, используя методику, которая будет описана в разделе 2.6.

**Теорема 2.5.5.** Для любой машины Тьюринга можно эффективным образом построить машину  $T_N$ , которая имеет ленту, не ограниченную только с одного конца, и которая моделирует машину  $T$ .

▷ Теорема 2.4.4 является простым следствием теоремы 2.4.3. В самом деле, если машина  $T$  вычисляет функцию  $f_T$ , то в силу теоремы 2.4.3 можно построить машину  $T_N$ , которая нормированно вычисляет эту функцию и, следовательно, моделирует машину  $T$ . Но поскольку при нормированном вычислении часть ленты, расположенная левее несобственной буквы  $\lambda$ , непосредственно предшествующей первому аргументу функции  $f_T$ , не участвует в вычислениях и не влияет на работу машины  $T_N$ , то эта часть ленты может быть отрезана. При этом работа машины  $T_N$  не изменится. Теорема доказана.

Теорема 2.4.4 позволяет ограничиться рассмотрением машин Тьюринга с лентой, бесконечной только с правого конца, а с левого — ограниченной. Если вычисления нормированы, то проблемы перехода за край ленты не возникает. □

## 2.5.5 Предикаты, вычислимые по Тьюрингу

В математической логике важную роль играет понятие **предиката**. Примером предиката может служить конструкция « $x$  умеет программировать», где  $x$  — переменная со значениями «Иванов», «Петров», «Сидорова» и т. д. Подставляя значения  $x$  в предикат, мы будем получать высказывания «Иванов умеет программировать», «Петров умеет программировать» и т. д., которые представляют одно из двух логических значений: И (истина) и Л (ложь) — в зависимости от квалификации Иванова, Петрова, Сидоровой и др. Если обозначить рассмотренный предикат через  $p(x)$  и предположить, что Петров умеет программировать, а Иванов не умеет, то  $p(\text{Иванов}) = \text{Л}$ ,  $p(\text{Петров}) = \text{И}$ . Аналогично конструкция « $x$  умеет  $y$ », где  $x$  — переменная со значениями «Иванов», «Петров» и т. д., а  $y$  — переменная со значениями «программировать», «отлаживаться», «петь», «свистеть» и т. д., задает предикат  $p(x, y)$  от двух переменных  $x$  и  $y$ . Если Иванов программист, а Пет-

ров — эстрадный певец, то  $p(\text{Иванов, программировать}) = \text{И}$ ,  $p(\text{Петров, отлаживаться}) = \text{Л}$ ,  $p(\text{Петров, свистеть}) = \text{И}$  и т.д.

Таким образом, предикат  $p(x_1, x_2, \dots, x_n)$  определяется как функция  $p : (A_s^*)^n \rightarrow \{\text{И}, \text{Л}\}$ . В случае  $n = 1$  предикат называют **свойством**, в случае  $n = 2, 3, \dots$  предикат задает  **$n$ -арное отношение**.

Предикаты широко применяются при описании алгоритмов (в частности, машин Тьюринга). Естественно, что в этом случае требуется, чтобы предикат сам описывался посредством алгоритма, т.е. был **вычислимым**. Понятия предиката, определяемого машиной Тьюринга, предиката, вычислимого по Тьюрингу, и предиката, нормированно вычислимого по Тьюрингу, определяются аналогично соответствующим понятиям для функций (см. п.2.4.3 и п.2.4.4). Приведем определение предиката, нормированно вычислимого по Тьюрингу.

**Определение 2.5.6.** Предикат  $p : (A_s^*)^n \rightarrow \{\text{И}, \text{Л}\}$  называется **нормированно вычислимым по Тьюрингу** (НВТ-предикатом), если существует МТ  $T_p$  с рабочим алфавитом  $A_t = A_s \cup \{\text{И}, \text{Л}\}$ , которая вычисляет  $p$  (см. определение 2.4.3) и при этом удовлетворяет следующим условиям:

1. если  $X = (u_1, u_2, \dots, u_n) \notin \text{Def}(p)$ , то машина  $T_p$  после применения к  $X$  никогда не останавливается;

2. если  $X = (u_1, u_2, \dots, u_n) \in \text{Def}(p)$ , и  $p(u_1, u_2, \dots, u_n) = \text{И}$ , то

$$[\lambda z \lambda u_1 \lambda u_2 \dots \lambda u_n (\lambda) \lambda] \xrightarrow{T_p}^*$$

$$[\lambda z \lambda u_1 \lambda u_2 \dots \lambda u_n \lambda \text{И} (\lambda) \lambda], z \in \bar{A}_p^*$$

3. если  $X = (u_1, u_2, \dots, u_n) \in \text{Def}(p)$ , и  $p(u_1, u_2, \dots, u_n) = \text{Л}$ , то

$$[\lambda z \lambda u_1 \lambda u_2 \dots \lambda u_n (\lambda) \lambda] \xrightarrow{T_p}^*$$

$$[\lambda z \lambda u_1 \lambda u_2 \dots \lambda u_n \lambda \text{Л} (\lambda) \lambda], z \in \bar{A}_p^*$$

Согласно теореме 2.4.3 любой ВТ - предикат является НВТ - предикатом.

## Лекция 11

### 2.6 Теоремы о сочетаниях алгоритмов

Теоремы о сочетаниях алгоритмов обосновывают использование уже сконструированных МТ при конструировании новых МТ. Вместе с тем они определяют правила, которые необходимо соблюдать при объединении нескольких МТ в одну, более сложную МТ.

#### 2.6.1 Теорема о композиции

**Теорема 2.6.1. (О композиции)** Пусть  $n$ -местная функция  $f : (A_s^*)^n \rightarrow A_t^*$  определяется равенством:

$$f(u_1, u_2, \dots, u_n) = h(g_1(u_1, u_2, \dots, u_n), g_2(u_1, u_2, \dots, u_n), \dots, g_m(u_1, u_2, \dots, u_n)),$$

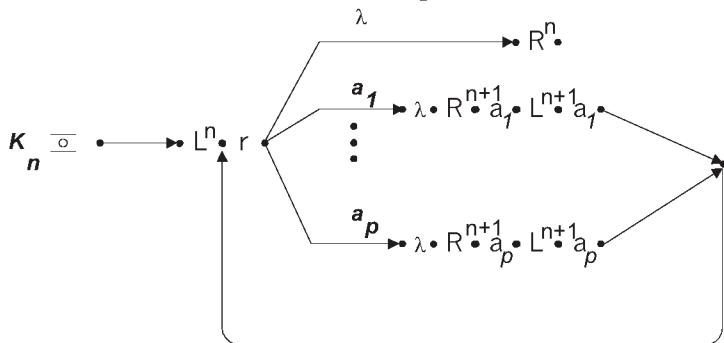
где  $g_i : (A_s^*)^n \rightarrow A_r^*$  ( $i = 1, 2, \dots, m$ ) и  $h : (A_r^*)^m \rightarrow A_t^*$  — ВТ-функции. Тогда  $f$  является ВТ-функцией, причем МТ, вычисляющая функцию  $f$ , может быть эффективно построена из МТ, вычисляющих функции  $g_i$  ( $i = 1, 2, \dots, m$ ) и  $h$ .

**Замечание.** В частном случае, когда  $m = 1, n = 1$ , функция  $f = h(g(k))$  называется **суперпозицией** функций  $g$  и  $h$ .

▷ Из теоремы 4.3.3 следует, что ВТ-функции  $g_i$  ( $i = 1, 2, \dots, m$ ) и  $h$  являются НВТ-функциями. Пусть  $T_{g_i}$  ( $i = 1, 2, \dots, m$ ) и  $T_h$  — МТ, реализующие нормированное вычисление функций  $g_i$  ( $i = 1, 2, \dots, m$ ) и  $h$  соответственно. Построим МТ  $K_n$ , реализующую действие

$$[\lambda w_1 \lambda w_2 \dots \lambda w_n(\lambda) \lambda] \xrightarrow{K_n} [ \lambda w_1 \lambda w_2 \dots \lambda w_n \lambda w_1(\lambda) \lambda ]$$

Машина  $K_n$  описывается диаграммой



Докажем, что машина  $T$ , определяемая диаграммой

$$T = . T_{g_1} K_{n+1}^n T_{g_2} \dots K_{n+1}^n T_{g_m} K_{(m-1)n+m} K_{(m-2)n+m} \dots K_{(m-m)n+m} T_h$$

вычисляет функцию  $f$ , т.е. реализует действие

$$[\lambda u_1 \lambda u_2 \dots \lambda u_n(\lambda) \lambda] \xrightarrow{T} [\lambda z \lambda f(u_1, u_2, \dots, u_n)(\lambda) \lambda],$$

где  $z \in \bar{A}_p^*$  ( $A_p$  — рабочий алфавит машин  $T_{g_i}$  ( $i = 1, 2, \dots, m$ ),  $T_h$ ,  $K_{n+1}$ ,  $K_{(m-j)n+m}$  ( $j = 1, 2, \dots, m$ )),  $T; A_s \subseteq A_p, A_t \subseteq A_p, A_r \subseteq A_p$ ).

Имеем:

$$\begin{aligned} & [\lambda u_1 \lambda u_2 \dots \lambda u_n(\lambda) \lambda] \xrightarrow{T_{g_i}}^* [\lambda u_1 \lambda u_2 \dots \lambda u_n \lambda g_1(\lambda) \lambda] \xrightarrow{K_{n+1}^n}^* \\ & [\lambda u_1 \lambda u_2 \dots \lambda u_n \lambda g_1 \lambda u_1 \lambda u_2 \dots \lambda u_n(\lambda) \lambda] \xrightarrow{T_{g_2}}^* \\ & [\lambda u_1 \lambda u_2 \dots \lambda u_n \lambda g_1 \lambda u_1 \lambda u_2 \dots \lambda u_n \lambda g_2(\lambda) \lambda] \xrightarrow{T_{g_m}}^* \dots \xrightarrow{K_{(m-1)n+m}}^* \\ & [\lambda u_1 \lambda u_2 \dots \lambda u_n \lambda g_1 \lambda u_1 \lambda u_2 \dots \lambda u_n \lambda g_m(\lambda) \lambda] \xrightarrow{K_{(m-m)n+m}}^* \\ & [\lambda u_1 \lambda u_2 \dots \lambda u_n \lambda g_m \lambda g_1 \lambda u_1 \lambda u_2 \dots \lambda u_n \lambda g_m \lambda g_1(\lambda) \lambda] \xrightarrow{T_h}^* \\ & \dots \xrightarrow{K_{(m-m)n+m}}^* [\lambda u_1 \lambda u_2 \dots \lambda u_n \lambda g_1 \lambda u_1 \lambda u_2 \dots \lambda u_n \lambda g_2 \dots \lambda u_1 \lambda u_2 \dots \\ & \lambda u_n \lambda g_m \lambda g_1 \lambda g_2 \dots \lambda g_m(\lambda) \lambda] \xrightarrow{h}^* [\lambda u_1 \lambda u_2 \dots \lambda u_n \lambda g_1 \lambda u_1 \lambda u_2 \dots \lambda u_n \lambda g_2 \dots \\ & \lambda u_1 \lambda u_2 \dots \lambda u_n \lambda g_m \lambda g_1 \lambda g_2 \dots \lambda g_m \lambda h(g_1, g_2, \dots, g_m)(\lambda) \lambda] = \\ & [\lambda z \lambda f(u_1, u_2, \dots, u_n)(\lambda) \lambda] \end{aligned}$$

так как по условию  $h(g_1(u_1, u_2, \dots, u_n), g_2(u_1, u_2, \dots, u_n), \dots, g_m(u_1, u_2, \dots, u_n)) = f(u_1, u_2, \dots, u_n)$ .

Теорема доказана.  $\square$

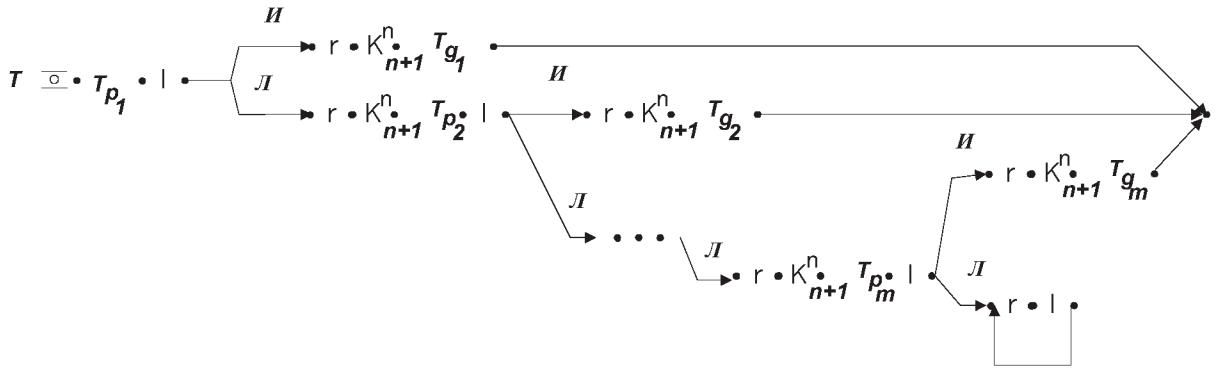
## 2.6.2 Теорема о ветвлении

**Теорема 2.6.2. (О ветвлении)** Пусть  $n$ -местная функция  $f : (A_s^*)^n \rightarrow A_t^*$  определяется равенством:

$$f(u_1, u_2, \dots, u_n) = \begin{cases} g_1(u_1, u_2, \dots, u_n), & \text{если } p_1(u_1, u_2, \dots, u_n) = \text{И} \\ g_2(u_1, u_2, \dots, u_n), & \text{если } p_2(u_1, u_2, \dots, u_n) = \text{И} \\ \vdots \\ g_m(u_1, u_2, \dots, u_n), & \text{если } p_m(u_1, u_2, \dots, u_n) = \text{И} \end{cases}$$

где  $g_i : (A_s^*)^n \rightarrow A_t^* (i = 1, 2, \dots, m)$  — ВТ-функции, а  $p_i : (A_s^*)^n \rightarrow \{\text{И}, \text{Л}\} (i = 1, 2, \dots, m)$  — ВТ-предикаты. Тогда  $f$  является ВТ-функцией, причем МТ, вычисляющая функцию  $f$  может быть эффективно построена из МТ, вычисляющих  $g_i$  и предикаты  $p_i (i = 1, 2, \dots, m)$ .

▷ Пусть  $T_{g_i}$  и  $T_{p_i} (i = 1, 2, \dots, m)$  — МТ, реализующие нормированное вычисление функций  $g_i$  и предикатов  $p_i (i = 1, 2, \dots, m)$  соответственно (возможность эффективного построения таких МТ для любых ВТ-функций следует из теоремы 2.4.3). Машина  $T$ , вычисляющая функцию  $f$ , определяется диаграммой:



понимании этого слова) характер. В этом случае ветвление идет по одной из открывшихся ветвей, выбираемых недетерминированным и неалгоритмическим образом. Подчеркнем, что даже случайный выбор ветки выполняется по некоторому вероятностному закону и, следовательно, детерминирован и алгоритмичен.

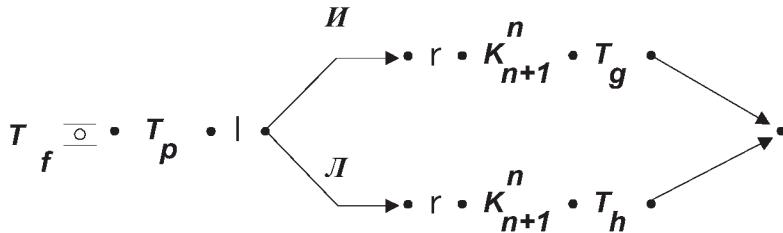
### 2.6.3 Следствие из теоремы о ветвлении

*Следствие.* Пусть  $n$ -местная функция  $f : (A_s^*)^n \rightarrow A_t^*$  определяется равенством:

$$f(u_1, u_2, \dots, u_n) = \begin{cases} g(u_1, u_2, \dots, u_n), & \text{если } p(u_1, u_2, \dots, u_n) = \text{И} \\ h(u_1, u_2, \dots, u_n), & \text{если } p(u_1, u_2, \dots, u_n) = \text{Л} \end{cases}$$

где  $g : (A_s^*)^n \rightarrow A_t^*$  и  $h : (A_s^*)^n \rightarrow A_t^*$  — ВТ-функции, а  $p : (A_s^*)^n \rightarrow \{\text{И}, \text{Л}\}$  — ВТ-предикат. Тогда  $f$  является ВТ-функцией, причем МТ, вычисляющая функцию  $f$ , может быть эффективно построена из МТ, вычисляющих функции  $g$  и  $h$  и предикат  $p$ .

▷ Пусть  $T_g, T_h, T_p$  — МТ, реализующие нормированное вычисление функций  $g$  и  $h$  и предиката  $p$  соответственно. Из теоремы 2.5.2 следует, что машина  $T_f$ , определяемая диаграммой



вычисляет функцию  $f$ .

Отметим, что это следствие более чем частный случай теоремы. Во-первых, для управления двумя ветвями используется только один предикат. Во-вторых, поскольку один предикат автоматически является полным и непротиворечивым набором, такое ветвление никогда не отказывает. В-третьих, психологи говорят, что ветвление более чем в два места является дополнительным источником ошибок. Это было замечено еще в конце 60-х годов, когда Эдсгер Дейкстра предложил идею структурированного программирования с использованием ограниченного набора конструкций, среди которых было только двузвенное ветвление. Это следствие охватывает наиболее распространенные конструкции языков программирования Pascal и C:

<pre> if p then   g else   h;       </pre>	<pre> if(p())   g(); else   h();       </pre>
--	---

□

### 2.6.4 Пример итеративного алгоритма

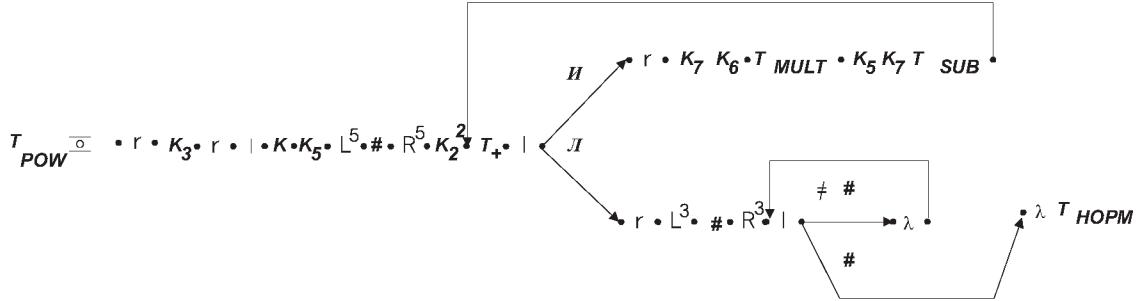
Многие алгоритмы сводятся к повторному выполнению более простых алгоритмов до тех пор, пока не будет выполняться некоторое условие. Один из таких алгоритмов рассмотрен

в примере 2.5.2. Рассмотрим еще несколько примеров таких алгоритмов.

**Пример 2.6.3.** Алгоритм возведения целого положительного числа в степень. Пусть  $T_{\text{MULT}}$  — МТ, реализующая нормированное вычисление функции  $\text{MULT}(u_1, u_2) = u_1 u_2$ , а  $T_+$  — МТ, реализующая нормированное вычисление предиката

$$+(u) = \begin{cases} I, & u \geq 0 \\ \Lambda, & u < 0 \end{cases}$$

Тогда машина  $T_{\text{POW}}$ , нормированно вычисляющая функцию  $w = \text{POW}(u_1, u_2) = u_1^{u_2}$ , может быть построена из МТ  $T_{\text{MULT}}$  и  $T_+$  следующим образом: полагаем  $w = 1$  и с помощью МТ  $T_{\text{MULT}}$  выполняем умножение  $u_1$  на  $w_2$ , одновременно уменьшая  $u_2$  на 1 (для этого нам понадобится МТ  $T_{\text{SUB}}$ , нормированно вычисляющая функцию  $\text{SUB}(u_1, u_2) = u_1 - u_2$ ), до тех пор, пока  $u_2 \neq 0$ . Диаграмма машины  $T_{\text{POW}}$  имеет вид:



Поясним работу машины  $T_{\text{POW}}$ . Последовательность МТ

$$T_1 = K_7 K_6 T_{\text{MULT}} K_5 K_7 T_{\text{SUB}}$$

выполняет следующие действия:

$$\begin{aligned} & [\lambda z \lambda u_1 \lambda w^{(1)} \lambda w^{(2)} \lambda u_2^{(1)} \lambda | u_2^{(2)}(\lambda) \lambda] \xrightarrow{K_7 K_6} \\ & [\lambda z \lambda u_1 \lambda w^{(1)} \lambda w^{(2)} \lambda u_2^{(1)} \lambda | \lambda u_2^{(2)} \lambda u_1 \lambda w^{(2)}(\lambda) \lambda] > \\ & [\lambda z \lambda u_1 \lambda w^{(1)} \lambda w^{(2)} \lambda u_2^{(1)} \lambda | \lambda u_2^{(2)} \lambda u_1 \lambda w^{(2)}(\lambda) \lambda] \xrightarrow{T_{\text{MULT}}} \\ & [\lambda z \lambda u_1 \lambda w^{(1)} \lambda w^{(2)} \lambda u_2^{(1)} \lambda | \lambda u_2^{(2)} \lambda u_1 \lambda w^{(2)} u_1.w^{(2)}(\lambda) \lambda] > \\ & [\lambda z \lambda u_1 \lambda w^{(1)} \lambda w^{(2)} \lambda u_2^{(1)} \lambda | \lambda u_2^{(2)} \lambda u_1 \lambda w^{(2)} u_1.w^{(2)}(\lambda) \lambda] \xrightarrow{K_5 K_7} \\ & [\lambda z \lambda u_1 \lambda w^{(1)} \lambda w^{(2)} \lambda u_2^{(1)} \lambda | \lambda u_2^{(2)} \lambda u_1 \lambda w^{(2)} u_1.w^{(2)} \lambda u_2^{(2)} \lambda | (\lambda) \lambda] \xrightarrow{T_{\text{SUB}}} \\ & [\lambda z \lambda u_1 \lambda w^{(1)} \lambda w^{(2)} \lambda u_2^{(1)} \lambda | \lambda u_2^{(2)} \lambda u_1 \lambda w^{(2)} u_1.w^{(2)} \lambda u_2^{(2)} \lambda | \lambda u_2^{(2)} - 1(\lambda) \lambda] > \end{aligned}$$

Таким образом, действие машины  $T_1$  состоит в

$$[\lambda z \lambda u_1 \lambda w^{(1)} \lambda w^{(2)} \lambda u_2^{(1)} \lambda | \lambda u_2^{(2)}(\lambda) \lambda] >$$

$$\xrightarrow{T_1} [\lambda \bar{z} \lambda \bar{u}_1 \lambda \bar{w}^{(1)} \lambda \bar{w}^{(2)} \lambda \bar{u}_2^{(1)} \lambda | \bar{u}_2^{(2)}(\lambda) \lambda] >$$

где  $\bar{z} = z \lambda u_1 \lambda w^{(1)} \lambda w^{(2)} \lambda u_2^{(1)} \lambda | \lambda u_2^{(2)}$ ,  $\bar{w}^{(1)} = w^{(2)}$ ,  $\bar{w}^{(2)} = u_1.w^{(2)}$ ,  $\bar{u}_2^{(1)} = u_2^{(2)}$ ,  $\bar{u}_2^{(2)} = u_2^{(2)} - 1$

Последовательность МТ

$$T_0 = r K_3 r | K K_5 L^5 \# R^5 K_2^2$$

выполняет действие

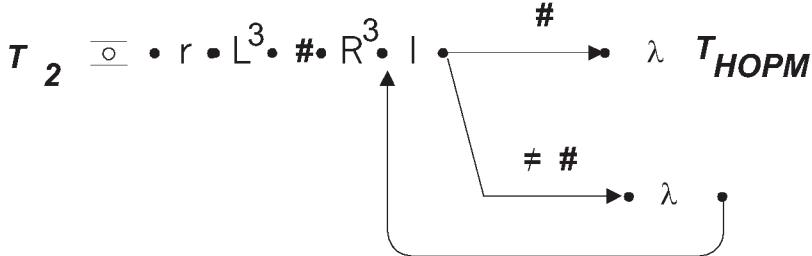
$$[\lambda u_1 \lambda u_2(\lambda) \lambda] \xrightarrow{T_0} [\lambda u_1 \lambda u_2 \# u_1 \lambda | \lambda | \lambda u_2 \lambda | \lambda u_2(\lambda) \lambda] >$$

преобразуя исходную ситуацию машины  $T_{\text{POW}} [\lambda u_1 \lambda u_2(\lambda) \lambda >$  в исходную ситуацию машины  $T_1$

$$[\lambda z \lambda u_1 \lambda w^{(1)} \lambda w^{(2)} \lambda u_2^{(1)} \lambda | \lambda u_2^{(2)}(\lambda) \lambda >$$

где  $z = u_1 \lambda u_2 \#$ ,  $w^{(1)} = w^{(2)} = 1$ ,  $u_2^{(1)} = u_2^{(2)} = u_2$ .

Наконец, последовательность МТ

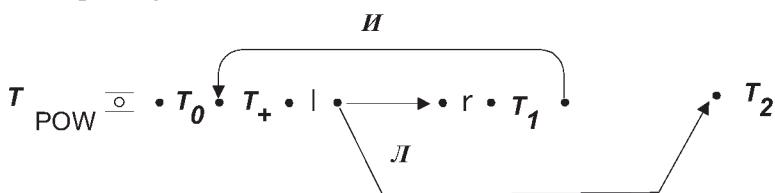


выполняет следующие действия:

$$[\lambda z \lambda u_1 \lambda w^{(1)} \lambda u_2^{u_2} \lambda u_2^{(1)} \lambda 1 \lambda 0(\lambda) \lambda > \xrightarrow{*} [\lambda z \lambda u_1 w^{(1)} \lambda u_2^{u_2} \# \lambda u_2^{(2)} \lambda 1 \lambda 0 \lambda(\lambda) \lambda > \xrightarrow{*}$$

$$[\lambda z \lambda u_1 \lambda w^{(1)} \lambda u_1^{u_2}(\lambda) \lambda >$$

$\xrightarrow{T_{\text{NORM}}} [\lambda u_1 \lambda u_2 \# z \lambda u_1 \lambda w^{(1)} \lambda u_1^{u_2}(\lambda) \lambda > \xrightarrow{*} [\lambda u_1 \lambda u_2 \lambda u_1^{u_2}(\lambda) \lambda >$   
обеспечивая нормированное вычисление функции POW. Обозначения  $T_0, T_1, T_2$  позволяют записать диаграмму машины  $T_{\text{POW}}$  в виде



Из последней диаграммы видно, что если исключить подготовительные и заключительные действия, выполняемые машинами  $T_0$  и  $T_2$  соответственно, то работа машины  $T_{\text{POW}}$  состоит в повторном применении машины  $T_1$  до тех пор, пока предикат, вычисляемый машиной  $T_+$ , имеет значение И.

## 2.6.5 Теорема о цикле

В примере 2.6.4 был рассмотрен алгоритм, состоящий в повторном вычислении двуместных функций MULT и SUB, причем в качестве второго аргумента функции MULT бралось значение этой функции, полученное при предыдущем ее вычислении, а в качестве первого аргумента функции SUB бралось ее предыдущее значение. Количество повторений определялось предикатом  $+$ . Такой алгоритм мы будем называть **циклом**.

Применение циклов при конструировании машин Тьюринга обосновывается следующей теоремой.

**Теорема 2.6.4.** Пусть  $g : (A_s^*)^n \rightarrow A_s^*$  — ВТ-функция, вычисляемая машиной  $T_g$ , а  $p : (A_s^*)^n \rightarrow \{\text{И}, \text{Л}\}$  — ВТ-предикат, вычисляемый машиной  $T_p$ . Тогда функция  $f : (A_s^*)^n \rightarrow A_s^*$ , определяемая вычислительным процессом

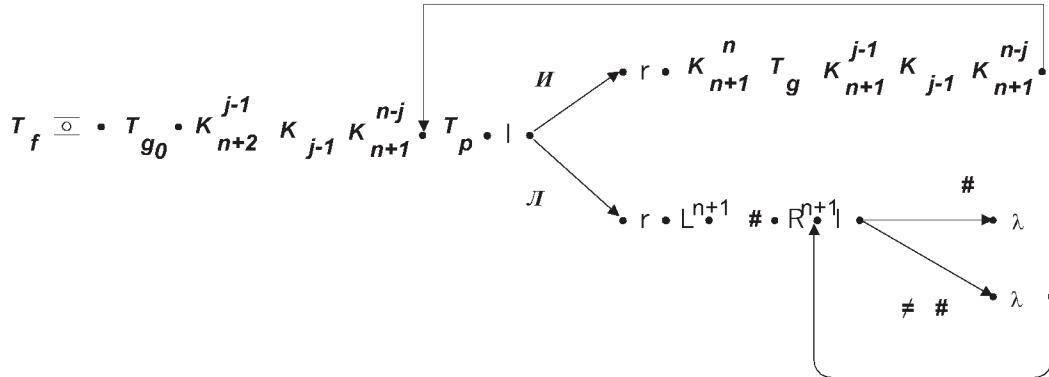
$$g = g_0, g_0 \in A_s^*$$

$$f(u_1, u_2, \dots, u_n) = g(u_1, u_2, \dots, u_{j-1}, g, u_{j+1}, \dots, u_n),$$

пока  $p(u_1, u_2, \dots, g, \dots, u_n) = \text{И}$ ,

тоже является ВТ-функцией, причем МТ, вычисляющая функцию  $f$ , может быть эффективно построена из машин  $T_g$  и  $T_p$ .

▷Машина  $T_f$ , вычисляющая функцию  $f$ , определяется диаграммой



где  $T_{g_0}$  — МТ, записывающая значение  $g_0$  в ячейку, следующую за рабочей:  $T_{g_0} = rg_0r$ .  
Теорема доказана.

Таким образом мы считаем, что в простейшем случае цикл состоит из повторяемого тела — вычислимой функции — и вычислимого предиката, управляющего повторением. В данном случае предикат предшествует повторению и является таким образом предусловием цикла. В частности, цикл, предусловие которого ложно, может не выполниться ни разу. И наоборот: если предикат находится в конце цикла, то он выполнится по крайней мере один раз. При этом в случае отказа в повторении цикла следует продолжение выполнения программы в естественном направлении — вперед, «далее по тексту».

В языках Pascal и С циклы с пред- и постусловием выглядят так:

<pre>while p do   g;</pre>	<pre>while(p())   g();</pre>
<pre>repeat   g; until not p;</pre>	<pre>do   g(); while(p());</pre>

Заметим, что **until** означает «пока не», поэтому вместо предиката обычно записывается его отрицание, поскольку по-русски «отрицательность» слова **until** неочевидна. В отличие от цикла **while**, где истинность предиката означает продолжение выполнения, в цикле **until** истинность предиката завершает цикл. В С оба цикла выполнены в одной логике, что более удобно как для понимания (т. к. не подразумевается неявное отрицание), так и для перестановки места проверки условия цикла. Цикл с тождественно истинным предусловием **while true do g**, не зависящим от текущего состояния повторяемого тела, никогда не останавливается также, как и тождественно ложное постусловие цикла **repeat g until false** никогда не обратится в **true** и не позволит завершить выполнение цикла. □

## 2.6.6 Обобщенная теорема о цикле

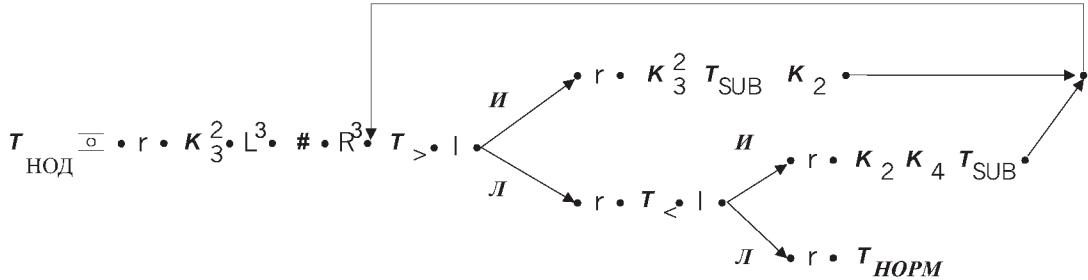
Теорема 2.6.5 может быть обобщена в нескольких направлениях. Наиболее интересно обобщение, предложенное Эдсгером Дейкстрой [11], вытекающее из следующего примера:

**Пример 2.6.5.** Алгоритм Евклида. Построим машину  $T_{\text{нод}}$ , реализующую алгоритм

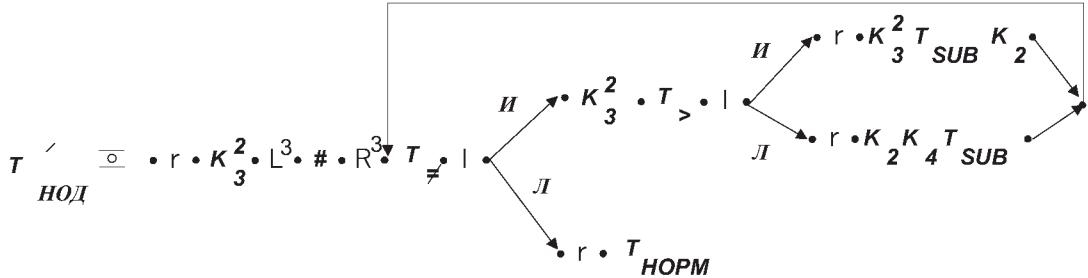
Евклида для вычисления наибольшего общего делителя двух чисел, используя машину  $T_{\text{SUB}}$ , рассматривающуюся в примере 2.4.4, и машины  $T_<$  и  $T_>$ , вычисляющие двуместные предикаты

$$> (u_1, u_2) = \begin{cases} И, \varphi(u_1) > \varphi(u_2) \\ Л, \varphi(u_1) \leq \varphi(u_2) \end{cases} \quad \text{и} \quad < (u_1, u_2) = \begin{cases} И, \varphi(u_1) < \varphi(u_2) \\ Л, \varphi(u_1) \geq \varphi(u_2) \end{cases}$$

соответственно. Алгоритм Евклида состоит в вычитании меньшего из большего до тех пор, пока оба числа не станут равны. Можно доказать, что тогда каждое из этих чисел будет равно искомому наибольшему делителю. Машину  $T_{\text{НОД}}$ , нормированно вычисляющая  $w = \varphi^{-1}(\text{НОД}(\varphi(u_1), \varphi(u_2)))$ , определяется диаграммой



В состав машины  $T_{\text{НОД}}$  входит цикл, определяемый двумя предикатами  $>$  и  $<$ . Смысл использования такого цикла становится ясным при сравнении диаграммы машины  $T_{\text{НОД}}$  с диаграммой машины  $T'_{\text{НОД}}$ , которая тоже реализует алгоритм Евклида.



Машина  $T_{\neq}$  вычисляет предикат

$$\neq (u_1, u_2) = \begin{cases} И, \text{ если } \varphi(u_1) \neq \varphi(u_2) \\ Л, \text{ если } \varphi(u_1) = \varphi(u_2) \end{cases}$$

Машина  $T_{\text{НОРМ}}$  описана в примере 2.5.4.

Следующая теорема является обоснованием применения циклов с несколькими телами и предикатами при конструировании МТ.

**Теорема 2.6.6.** (об обобщённом цикле, Э. Дейкстра) Пусть  $g_i : (A_s^*)^n \rightarrow A_s^*$  ( $i = 1, 2, \dots, m$ ) — ВТ-функции, вычисляемые МТ  $T_{g_i}$ , а  $p_i : (A_s^*)^n \rightarrow \{\text{И}, \text{Л}\}$  ( $i = 1, 2, \dots, m$ ) — ВТ-предикаты, вычисляемые МТ  $T_{p_i}$  ( $i = 1, 2, \dots, m$ ) соответственно. Тогда функция, определяемая соотношениями:

$$g_i = g_{0_i}, g_{0_i} \in A_s^*, i = 1, 2, \dots, m$$

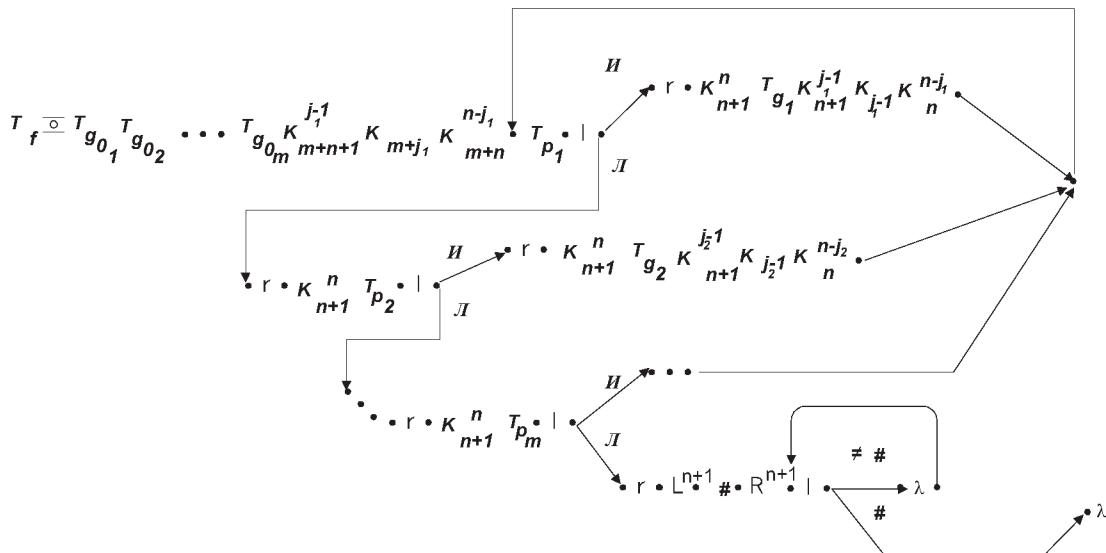
$$f(u_1, \dots, u_n) =$$

$$\begin{cases} g_1(u_1, \dots, u_{j_1-1}, g_1, u_{j_1+1}, \dots, u_n), \text{ если } p_1(u_1, \dots, g_1, \dots, u_n) = И, \\ g_2(u_1, \dots, u_{j_2-1}, g_2, u_{j_2+1}, \dots, u_n), \text{ если } p_2(u_1, \dots, g_2, \dots, u_n) = И, \\ \dots \\ g_m(u_1, \dots, u_{j_m-1}, g_m, u_{j_m+1}, \dots, u_n), \text{ если } p_m(u_1, \dots, g_m, \dots, u_n) = И \end{cases}$$

пока  $\bigvee_{j=1}^m p_j = И$ ,

тоже является ВТ-функцией, причем МТ, вычисляющая функцию, может быть эффективно построена из машин  $T_{g_i}$  и  $T_{p_i}$  ( $i = 1, 2, \dots, m$ ).

▷ Машина  $T_f$ , вычисляющая функцию  $f$ , определяется диаграммой:



Машины  $T_{g_0_i}$  ( $i = 1, 2, \dots, m$ ) аналогичны машине  $T_{g_0}$ , рассмотренной при доказательстве теоремы 4.5.5. Теорема доказана.  $\square$

**Замечание.** Данная теорема описывает цикл с  $m$  альтернативными телами. Также как и в случае ветвления, каждое тело охраняется своим предикатом. Но в отличие от ветвления, цикл ориентирован на многократное повторение, причем в данном случае повторения могут идти по разным телам. Поэтому требования к набору предикатов снижаются. Отказ всех предикатов цикла в отличие от ветвления не приводит к аварийному завершению, а всего лишь прекращает выполнение цикла. («Что русскому хорошо, то немцу смерть».) Если набор предикатов цикла противоречив и при каком-либо повторении цикла соответствующими предикатами открыто для выполнения более одного тела, то выполняется любое из этих тел. То есть в данном случае программируется недетерминированное выполнение, выводящее нас за пределы алгоритма в традиционном понимании. В простейшем случае прямого исполнения такого цикла на последовательном компьютере может быть выполнено первое в порядке написания тело. Таким образом цикл никогда не отказывает. При реализации цикла необходимо учитывать его двойственную природу: стремление продолжать выполнение цикла с запрограммированным завершением, т. е. предикаты  $p_i$  должны зависеть от результатов вычисления функций  $g_k$  на предыдущих итерациях так, чтобы они постепенно закрывали тела цикла, приводя к его завершению. Ведь цикл работает «до последнего предиката»!

## Лекция 12

## 2.7 Схемы машин Тьюринга

### 2.7.1 Конструирование машин Тьюринга «сверху вниз»

В предыдущих разделах в качестве примеров и при доказательстве теорем было рассмотрено достаточно большое количество конкретных МТ. Было показано, что если применять для описания программ МТ технику диаграмм Тьюринга, то имеется принципиальная возможность использования ранее сконструированных МТ при разработке новых диаграмм Тьюринга. Это обстоятельство упрощает процесс конструирования новых МТ, но, к сожалению, лишь несущественно, так как для использования МТ  $T$  в диаграмме, описывающей МТ  $M$ , необходимо, чтобы функция  $f_M$ , вычисляемая машиной  $M$ , выражалась через функцию  $f_T$ , которую вычисляет машина  $T$ , что справедливо далеко не для всех пар машин  $M$  и  $T$ .

Нормирование Тьюринговых вычислений и теоремы о сочетаниях алгоритмов, доказанные в п. 2.6, позволяют разработать и обосновать процедуру, которая обеспечивает систематическое и целенаправленное сведение задачи конструированная некоторой МТ к нескольким задачам конструирования более простых МТ. Эта процедура называется конструированием «сверху вниз» и состоит в следующем. Пусть поставлена задача конструирования МТ  $T$ , вычисляющей функцию  $f$ . Выражаем функцию  $f$  через функции  $f_1, f_2, \dots, f_k$  и составляем диаграмму машины  $T$ , включающую символы машин  $T_j$ , вычисляющих функции  $f_j$  ( $j = 1, 2, \dots, k$ ). Каждую из функций  $f_j$  выражаем через новые функции  $f_{j_1}, \dots, f_{j_{l_j}}$  и составляем диаграммы машин  $T_j$ , включающие символы машин  $T_{j_s}$ , вычисляющих функции  $f_{j_s}$  ( $s = 1, 2, \dots, l_j$ ;  $j = 1, 2, \dots, k$ ). Продолжая процесс до тех пор, пока на каком-то уровне не получается диаграммы, включающие только символы элементарных МТ, получим диаграмму машины  $T$ , которую требовалось сконструировать. Следует отметить, что, выражая одни функции через другие, можно использовать лишь те операции (сочетания функций), которые были указаны в теоремах о сочетаниях алгоритмов, т. е. последовательную композицию функций, ветвление (альтернативное вычисление функций) и цикл (итерацию функций). То есть правила сочетания алгоритмов применяются в обратном направлении как правила декомпозиции.

Процесс конструирования МТ «сверху вниз» поясним на следующем примере.

**Пример 2.7.1.** Сконструируем машину Тьюринга  $X$ , реализующую алгоритм Евклида для вычисления наибольшего общего делителя (НОД) двух натуральных чисел в десятичной позиционной системе счисления. Числа будем представлять на ленте МТ в виде непустых слов  $u$  и  $v$  над алфавитом  $D = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ , интерпретируя слово  $u = \overline{a_n a_{n-1} \dots a_0}$  длины  $n + 1$  как число  $\varphi(u) = \sum_{i=0}^n \varphi(a_i) \cdot 10^i$ .

Действие машины  $X$  описывается следующей парой конфигураций:

$$[\lambda u \lambda v (\lambda) \lambda] \xrightarrow{X} [\lambda u \lambda v \lambda w (\lambda) \lambda]$$

где  $w$  — слово над алфавитом  $D$ , представляющее число НОД( $\varphi(u)$ ,  $\varphi(v)$ ).

Алгоритм Евклида состоит в сравнении чисел и замене большего из них разностью между большим и меньшим до тех пор, пока оба числа не станут равны между собой. Как только числа сравняются, алгоритм Евклида заканчивается, причем каждое из полученных равных чисел равно НОД. Алгоритм Евклида сводит сложную мультипликативную операцию к последовательности простых аддитивных, выполняя вместо деления

необходимое число вычитаний. Следовательно, для составления диаграммы машины  $X$  нам, во-первых, потребуется машина  $C$ , сравнивающая два натуральных числа:

$$[\lambda u \lambda v (\lambda) \lambda] \xrightarrow{C} * [\lambda u \lambda v \lambda \delta(\lambda) \lambda],$$

причем

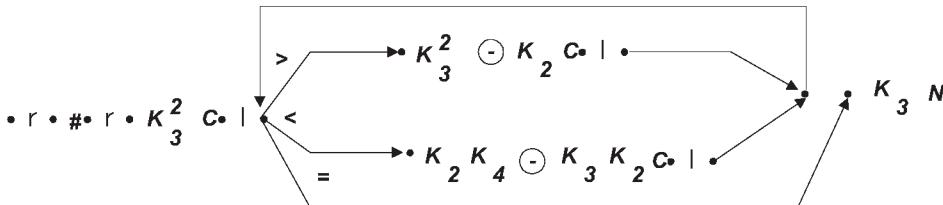
$$\delta = \begin{cases} =, & \text{если } \varphi(u) = \varphi(v), \\ <, & \text{если } \varphi(u) < \varphi(v), \\ >, & \text{если } \varphi(u) > \varphi(v), \end{cases}$$

и, во-вторых, машина  $-$ , вычисляющая разность чисел  $\varphi(u)$  и  $\varphi(v)$  при условии  $\varphi(u) > \varphi(v)$ :

$$[\lambda u \lambda v (\lambda) \lambda] \xrightarrow{-} * [\lambda u \lambda v \lambda w (\lambda) \lambda],$$

где  $w \in D^*$  — слово, представляющее десятичную позиционную запись числа  $\varphi(w) = \varphi(u) - \varphi(v)$ .

Используя символы машин  $C$  и  $-$ , мы можем составить следующую диаграмму машины  $X$ :

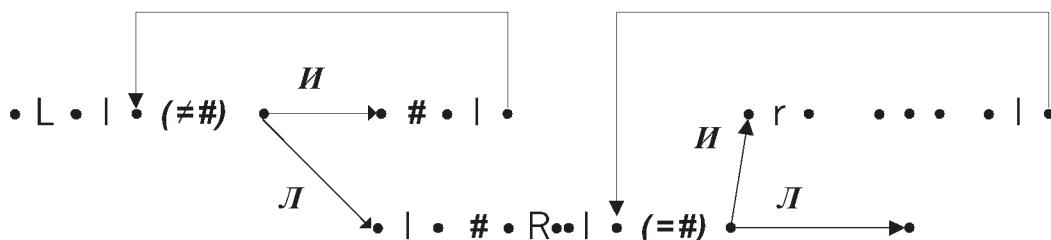


Построенная диаграмма машины  $X$  содержит символы элементарных МТ  $r$ ,  $l$  и  $\#$  (помещает в рабочую ячейку знак  $\#$ ), символы машин  $K_n$  (см. п. 2.2.3, а также 2.6.1) и символы машин  $-$ ,  $C$ ,  $N$ , которые еще не сконструированы. Следовательно, задача конструирования машины  $X$  сведена к задаче конструированная более простых машин  $N$ ,  $-$  и  $C$ .

Машина  $N$  нормирует вычисление, т. е. убирает все промежуточные результаты и помещает окончательный результат вслед за аргументами:

$$[\lambda u \lambda v \lambda \# \lambda W \lambda w (\lambda) \lambda] \xrightarrow{N} * [\lambda u \lambda v \lambda w (\lambda) \lambda].$$

(через  $W$  обозначено слово над расширенным алфавитом машины  $X$  (точнее, цепочка слов), представляющее промежуточные результаты вычисления НОД). Диаграмма машины  $N$  может иметь вид



В последней диаграмме использована машина  $T_{\neq}$ , построенная в п. 2.6.6.

Перейдем к конструированию машин  $C$  и  $-$ . Прежде всего необходимо отметить, что эти машины могут конструироваться независимо одна от другой и в любой последовательности. Иными словами, исходная задача конструирования машины  $X$  сведена к двум

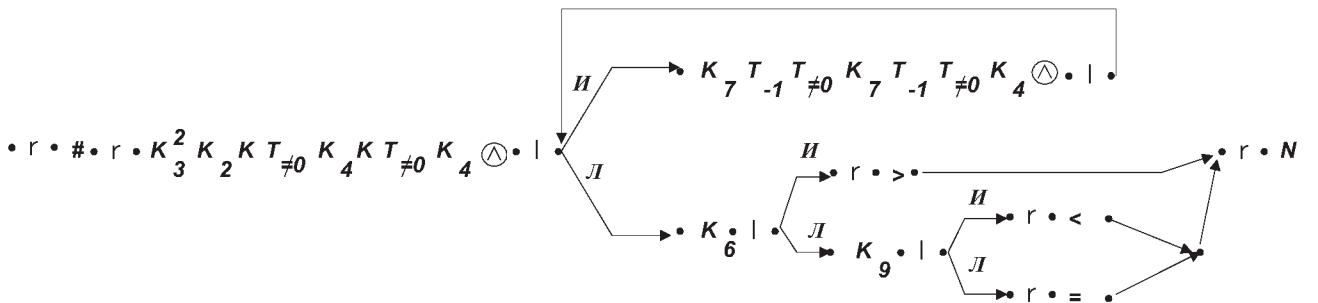
независимым более простым задачам конструирования машин  $C$  и  $-$ . Сравнение двух натуральных чисел будем проводить, вычитая из каждого числа по единице и следя за тем, какое из них раньше обратится в нуль. Этот способ в позиционных системах счисления не применяется, но придает нашей задаче простоту, свойственную действиям в натуральной системе счисления. Для этого нам потребуется машина  $T_{-1}$ , уменьшающая натуральное число в десятичной позиционной записи на единицу:

$$[\lambda W \lambda u(\lambda) \lambda] \xrightarrow{T_{-1}}^* [\lambda W \lambda u \lambda v(\lambda) \lambda],$$

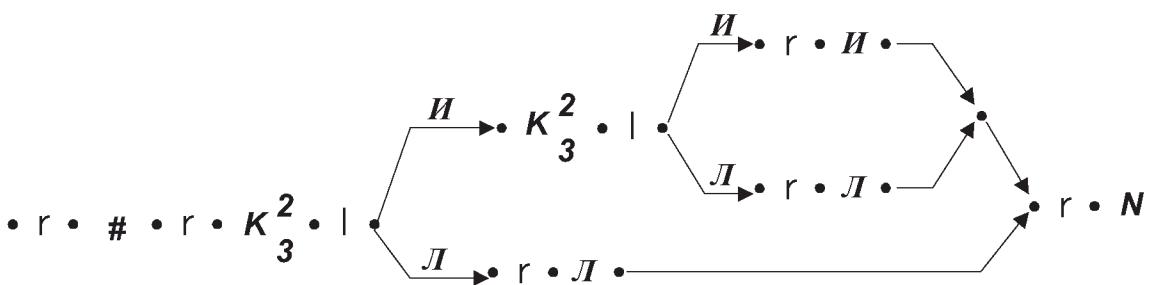
где  $\varphi(v) = \varphi(u) - 1$ , и машина  $T_{\neq 0}$ , вычисляющая значение предиката «натуральное число, записанное на ленте, отлично от нуля»:

$$[\lambda W \lambda u(\lambda) \lambda] \xrightarrow{T_{\neq 0}}^* [\lambda W \lambda u \lambda \delta(\lambda) \lambda],$$

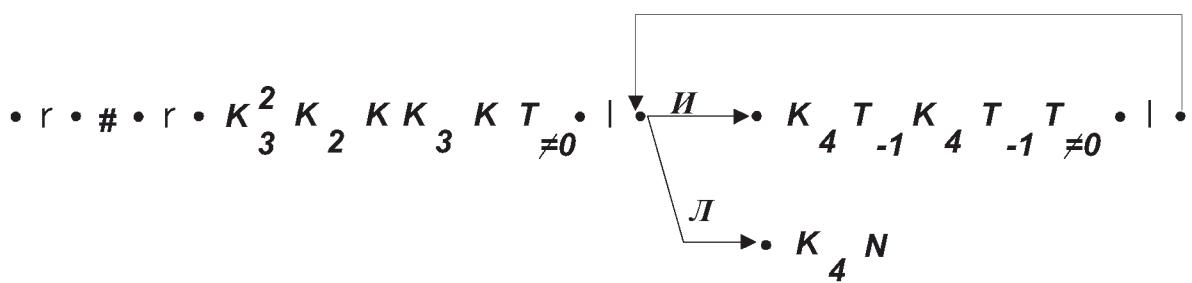
где  $\delta = И$  (истина), если  $\varphi(u) \neq 0$ , и  $\delta = Л$  (ложь), если  $\varphi(u) = 0$ . Диаграмма машины  $C$ :



В этой диаграмме использован символ машины  $\wedge$ , вычисляющей конъюнкцию ( $\&$ ). Диаграмма машины  $\wedge$  настолько проста, что мы ее составим сразу:



Вычитание натуральных чисел при условии, что уменьшаемое больше вычитаемого, будем также производить с помощью машин  $T_{-1}$  и  $T_{\neq 0}$ , вычитая из уменьшаемого и вычитаемого по единице до тех пор, пока вычитаемое не обратится в нуль. Здесь разность двух натуральных чисел получается по определению натурального числа как превышение количества единиц в одном числе над натуральным значением другого числа. Этот способ также выбран для простоты реализации. Таким образом, диаграмма машины  $-$  может иметь вид



Чтобы закончить составление диаграммы машины  $X$ , осталось сконструировать МТ  $T_{-1}$  и  $T_{\neq 0}$ . Диаграммы этих машин настолько просты, что их можно выписать сразу. Если считать, что слово, к которому применяется машина  $T_{-1}$ , изображает число, отличное от нуля, то диаграмма машины  $T_{-1}$  может иметь вид

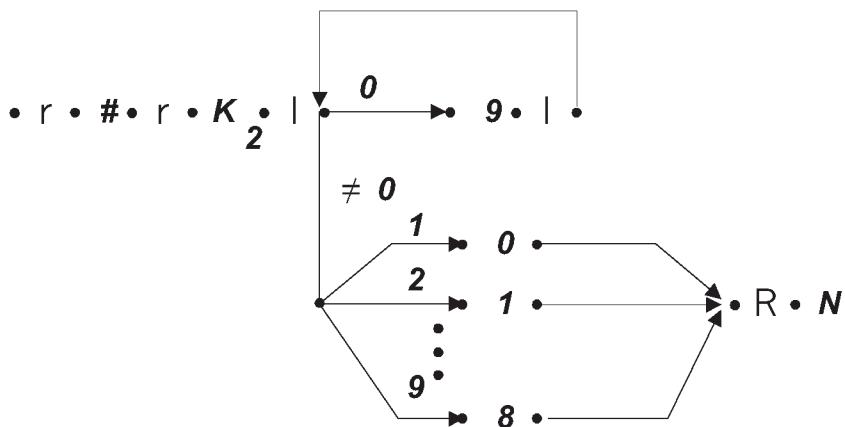
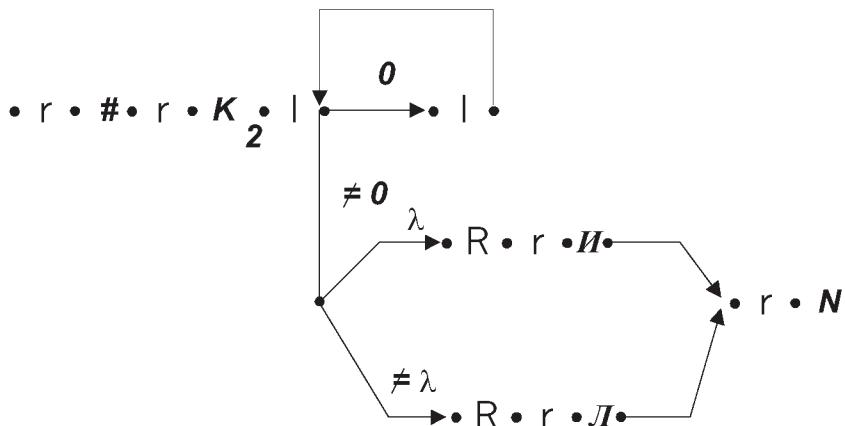


Диаграмма машины  $T_{\neq 0}$ :



Последние две диаграммы содержат только символы элементарных МТ и ранее сконструированных МТ. Поэтому построение машины  $X$ , вычисляющей НОД двух натуральных чисел, на этом завершается.

Из примера 2.7.1 видно, что, применяя метод конструирования МТ «сверху вниз», можно на каждом этапе рассматривать достаточно простые диаграммы. МТ, символы которых используются при составлении диаграмм более высокого уровня, определяются в процессе разработки этих диаграмм. Вводя эти символы, мы сводим исходную задачу к нескольким независимым задачам конструирования более простых МТ. Таким образом,

техника описания программ МТ с помощью диаграмм, нормирование вычислений и метод нисходящей разработки («сверху вниз») открывают возможность систематического конструирования достаточно сложных МТ. При этом теоремы о сочетаниях алгоритмов позволяют обосновать процесс конструирования с тем, чтобы по завершении конструирования МТ иметь уверенность в том, что разработанная МТ выполняет именно ту обработку сообщений, которая требовалась в задаче.

## 2.7.2 Определение схемы машины Тьюринга

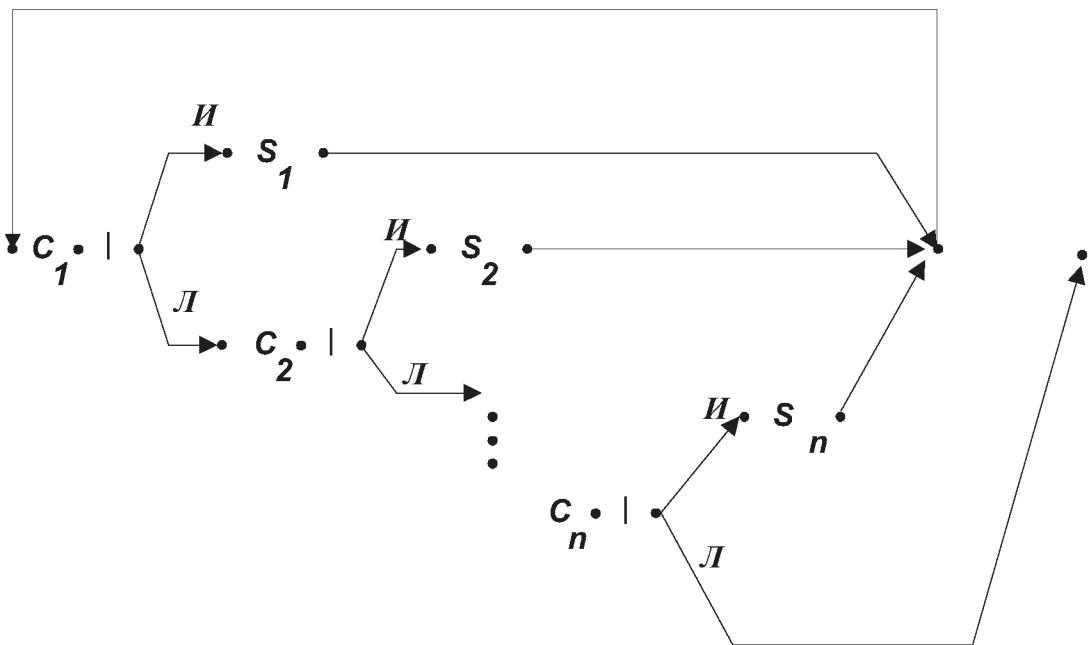
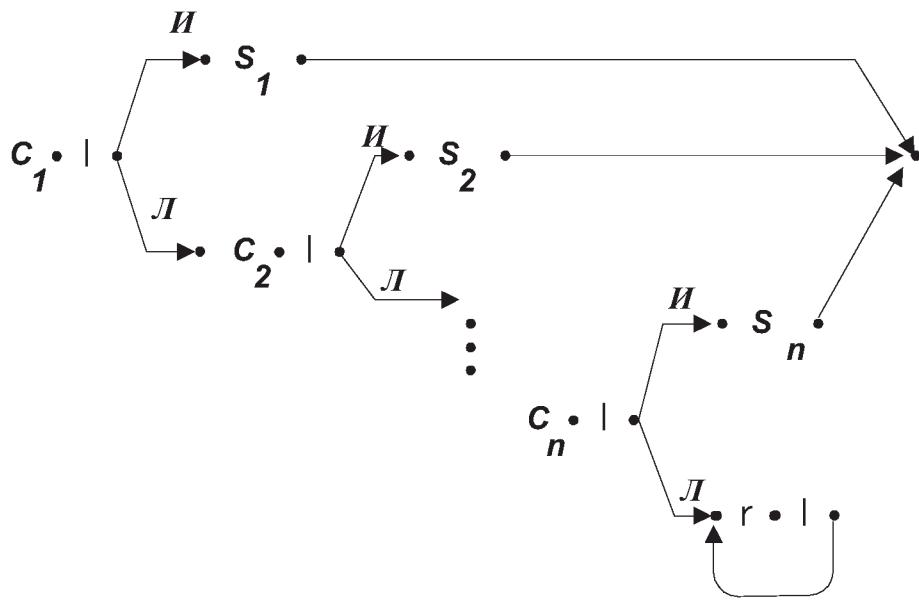
Процесс конструирования МТ «сверху вниз» состоит в последовательной разработке диаграмм все более и более простых МТ, причем каждая из последующих диаграмм описывает работу МТ, символ которой содержится по крайней мере в одной из ранее составленных диаграмм. Следовательно, каждая из диаграмм, составленных в процессе конструирования «сверху вниз», должна быть такой, чтобы ее можно было заменить символом описываемой ею МТ. Этому требованию удовлетворяют не все диаграммы, а только такие, которые описывают МТ, имеющие одно начальное и только одно конечное состояние. Дело в том, что каждый символ элементарной МТ на диаграмме Тьюринга соответствует какому-нибудь одному состоянию МТ, описываемой этой диаграммой; если же в диаграмме Тьюринга встречается символ неэлементарной МТ, то это означает, что на уровне указанной диаграммы начальное и конечное состояния указанной неэлементарной МТ как бы «склеиваются» в одно. В случае, если конечных состояний несколько, такая «склейка» (или «отождествление») состояний становится невозможной.

Отметим, что МТ с несколькими конечными состояниями сконструировать нетрудно. В п. 2.4.2 при доказательстве II теоремы Шеннона была построена МТ, подсчитывающая количество палочек над алфавитом  $\{ \mid \}$ , которая имеет  $2i+1$  конечных состояний (здесь  $i$  — номер буквы алфавита МТ, которая моделируется над алфавитом  $\{ \mid \}$ ). Эта МТ входит в состав диаграммы, описывающей МТ, которая имеет только одно конечное состояние. МТ, подсчитывающая количество палочек, не может быть заменена одним символом в диаграмме, в состав которой она входит.

Всякую МТ, которая имеет одно начальное и одно конечное состояние, будем называть **собственной**. Выше было показано, что в состав диаграммы, описывающей собственную МТ, могут входить несобственные МТ. Если это не имеет места, т. е. если диаграмма, описывающая некую МТ, содержит только собственные МТ, то МТ, описываемая этой диаграммой, называется МТ со структурированным управлением, или **структурной** МТ. Легко видеть, что в процессе конструирования «сверху вниз» мы применяем только такие способы декомпозиции, при которых получаются именно структурные МТ. Диаграммы структурных машин мы будем называть **схемами**. Таким образом, схема МТ — это частный случай диаграммы Тьюринга.

**Определение 2.7.2.** Понятие схемы машины Тьюринга определим следующим образом, отражающим нисходящий процесс конструирования:

1. символ  $\bullet$  составляет схему;
2. если  $S_1, S_2, \dots, S_n$  — схемы, то их последовательная композиция  $S_1 S_2 \dots S_n$  является схемой;
3. если  $C_1, C_2, \dots, C_n$  — схемы МТ, вычисляющих предикаты  $P_1, P_2, \dots, P_n$  соответственно, и  $S_1, S_2, \dots, S_n$  — схемы, то и конструкции



являются схемами;

4. символы элементарных машин Тьюринга составляют схемы;
5. никакие другие объекты схемами не являются.

Все диаграммы, построенные в примере 2.7.1, являются схемами.

### 2.7.3 Эквивалентность схем и диаграмм

Из определения 2.7.2 следует, что всякая схема МТ является диаграммой Тьюринга, но не всякая диаграмма является схемой. Так, диаграммы, составленные при доказательстве I теоремы Шеннона, не являются схемами. Поскольку при конструировании МТ «сверху

вниз» получаются обязательно схемы, возникает вопрос: для любого ли алгоритма существует МТ, которую можно сконструировать нисходящим образом? Ответ на этот вопрос дает следующая теорема.

**Теорема 2.7.3. (Бойма-Якопини-Миллса)** Для любой машины Тьюринга  $T$  можно эффективно построить машину Тьюринга  $S$ , которая является структурной (т. е. диаграмма которой является схемой) и которая моделирует машину  $T$ .

Доказательство этой теоремы состоит в преобразовании каждой части диаграммы машины  $T$  в одну из трех структур, приведенных в определении 2.7.2. Каждое такое преобразование (мы будем называть его структурирующим преобразованием) уменьшает неструктурную часть диаграммы. После достаточного количества структурирующих преобразований неструктурная часть диаграммы исчезает. Возможность проведения структурирующих преобразований вытекает из первой теоремы Шеннона, так как каждое такое преобразование сводится к уменьшению числа состояний моделируемой МТ. При этом, естественно, рабочий алфавит моделируемой МТ расширяется (в него вводятся дополнительные буквы). Полного доказательства теоремы 2.7.3 мы приводить не будем из-за его громоздкости.

Теорема 2.7.3 была сформулирована Боймом и Якопини в 1966 г., однако, как было впоследствии установлено сотрудником фирмы IBM Миллсом, их доказательство было неполным. Полное доказательство было опубликовано Миллсом в 1972 году [42, 38].

Ниже следует полученное Д. В. Рисенбергом чисто тьюринговское, диаграммное доказательство теоремы, основанное на идеях Клода Шеннона о сокращении числа состояний произвольной машины Тьюринга за счет увеличения числа букв рабочего алфавита.

### 2.7.3.1 Доказательство

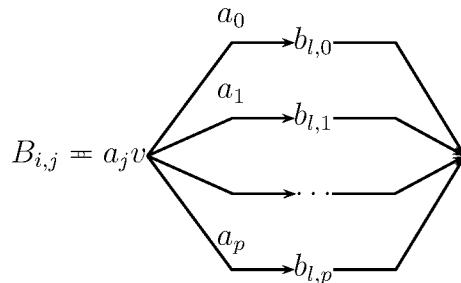
Пусть машина  $T$  имеет алфавит  $A = \{a_1, \dots, a_p\}$  и набор состояний  $Q = \{q_0, \dots, q_s\}$ . Тогда алфавит машины  $T'$  будет содержать помимо алфавита  $A$  еще  $(p+1)(s+1)$  знаков  $b_{i,j}$ .

Каждой команде из программы машины  $T'$ , не являющейся терминальной  $(q_i, a_j, s, q_l)$ , поставим в соответствие некоторую структурную машину  $B_{i,j}$  по следующему правилу:

- Если команда машины  $T$  имеет вид  $(q_i, a_j, a_k, q_l)$ , то ей соответствует машина перезаписи буквы

$$B_{i,j} = b_{l,k}.$$

- Если команда машины  $T$  имеет вид  $(q_i, a_j, v, q_l)$ ,  $v \in \{l, r\}$ , то ей соответствует машина движения

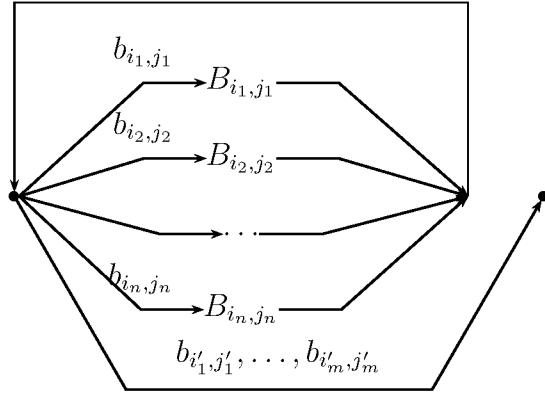


- Если команда машины  $T$  имеет вид  $(q_i, a_j, s, q_l)$  ( $l \neq i$ ), то ей соответствует стоп-машина

$$B_{i,j} = b_{l,j}.$$

Таким образом, каждый знак  $b_{i,j}$  соответствует определяющей паре  $(q_i, a_j)$  исходной машины  $T$  и неявно содержит в себе информацию о номере состояния, в котором она находится. Таким образом, сгенерирована матрица инциденций  $B_{ij}$ , отражающая наличие связей всех состояний (точкомест) в диаграмме.

Теперь можно построить диаграмму машины  $T'$ :



Здесь введена сплошная (от 1 до  $n$ ) нумерация для пар  $(i, j)$  таких, что пара  $(q_i, a_j)$  определяет некоторую команду машины  $T$ , не являющуюся терминальной. Аналогично была введена сплошная (от 1 до  $m$ ) нумерация для пар  $(i, j)$ , для которых пара  $(q_i, a_j)$  определяет терминальную команду машины  $T$ . Сам порядок, в котором перенумерованы пары, не важен.

Исходя из построенной диаграммы, можно заметить, что машина  $T'$  является структурной, т. к. структурными являются все машины  $B_{i,j}$ .

Осталось показать, что машина  $T'$  моделирует машину  $T$ .

Конфигурации

$$\begin{array}{c} [\lambda a_{l_1} \dots a_{l_k} \dots a_{l_n} \lambda] \\ q_i \end{array}$$

машины  $T$  поставим в соответствие конфигурацию

$$\begin{array}{c} [\lambda a_{l_1} \dots b_{i,l_k} \dots a_{l_n} \lambda] \\ q^* \end{array}$$

машины  $T'$ , где состояние  $q^*$  может быть любым. Тогда начальной конфигурацией машины  $T'$  будет

$$\begin{array}{c} [\lambda a_{l_1} \dots b_{i_{\text{нач}}, l_{k_{\text{нач}}} \dots a_{l_n} \lambda], \\ q \end{array}$$

соответствующая начальной конфигурации машины  $T$  (здесь  $q$  — начальное состояние  $T'$ ).

Рассмотрим выполнение нетерминальной команды  $(q_i, a_j, v, q_l)$ , где  $v \in A \cup \{\lambda\} \cup \{l, r, s\}$ , а  $i \neq l$ .

Если  $v \in A \cup \{\lambda\}$ , то машина  $T'$  просто заменяет в своей рабочей ячейке знак  $b_{i,j}$  на  $b_{l,k}$ . Если  $v \in \{l, r\}$ , то машина машина  $T'$  сначала считает из ячейки букву  $b_{i,j}$ , восстановит в ней букву  $a_j$ , сдвигается в направлении  $v$  и в новой рабочей ячейке записывает букву  $b_{l,j'}$ , если до этого в ней была записана буква  $a_{j'}$ . Если  $v = s$ , то машина  $T'$  заменяет в рабочей ячейке букву  $b_{i,j}$  на  $b_{l,j}$ . Наконец, если команда  $(q_i, a_j, v, q_l)$  является терминальной ( $l = i, v = s$ ), то машина  $T'$  сразу же выходит из цикла и завершает работу.

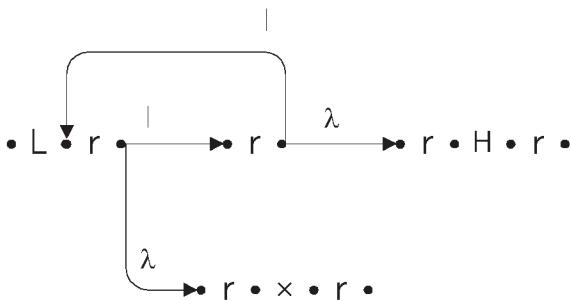
Таким образом, теорема Бойма-Якопини-Миллса доказана.

**Замечание.** Данное доказательство неконструктивно, но весьма миниатюрно и изящно как будто бы его делал не программист, а математик. Харлан Миллс, популяризируя результаты Бойма и Якопини для программистов-практиков из IBM, дал конструктивное доказательство теоремы на базе блок-схем программ.

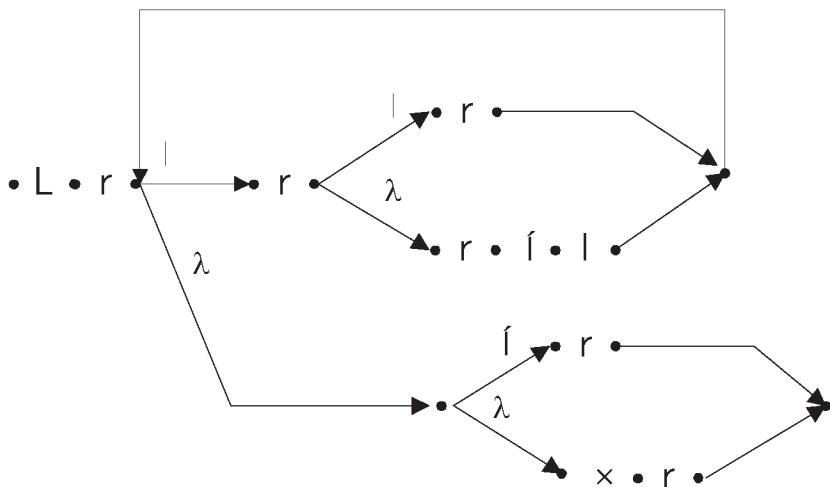
Из теоремы Бойма-Якопини-Миллса вытекает важное

**Следствие.** Всякая диаграмма может быть преобразована к виду, не использующему произвольные передачи управления — стрелки, соединяющие элементы диаграммы неструктурированными способами. В практике программирования это означает, всякая программа может быть преобразована в эквивалентную, не содержащую ни меток, ни операторов безусловного перехода **goto**. Программирование без **goto** вместе с нисходящей разработкой было одним из элементов технологии структурного программирования — одной из первых технологий (после процедурного и модульного программирования).

**Пример 2.7.4.** Рассмотрим пример структурированного преобразования диаграммы Тьюринга. Машина Тьюринга, просматривающая слово  $w \in \{|\}^*$  и печатающая в качестве результата своей работы букву Ч, если слово  $w$  содержит четное число палочек, и букву Н, если  $w$  содержит нечетное число палочек, описывается следующей неструктурированной диаграммой:



Эта диаграмма может быть заменена следующей схемой:



В рассмотренном примере удалось структурировать диаграмму МТ, не вводя вспомогательных букв в ее рабочий алфавит (роль такой буквы удалось возложить на Н).

В середине XX века многими отечественными исследователями были предложены свои исчисления схем программ [39].

## Лекция 13

## 2.7.4 Доказательство теоремы о нормированной вычислимости

Для доказательства теоремы 2.5.4 по МТ  $T_f$ , вычисляющей функцию  $f$ , необходимо построить МТ  $\bar{T}_f^N$ , которая нормированно вычисляет ту же функцию  $f$ . Для построения МТ  $\bar{T}_f^N$  используем прием конструирования МТ «сверху вниз».

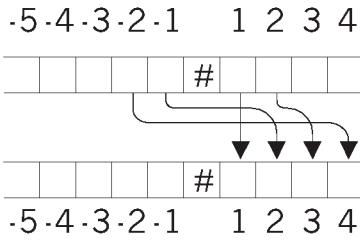
Сначала построим МТ  $\bar{T}_f^N$ , которая нормированно вычисляет функцию  $f$ , но рабочий алфавит которой содержит на один знак больше, чем рабочий алфавит МТ  $T_f$ . Этот дополнительный знак обозначим через  $\#$ .

Машина  $\bar{T}_f^N$  вычисляет ту же функцию  $f$ , что и машина  $T_f$ , но отличается от машины  $T_f$  следующими свойствами:

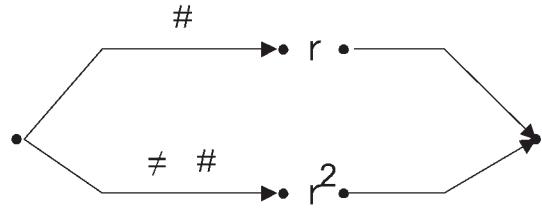
1. машина  $\bar{T}_f^N$  не портит аргументов функции  $f$ , а для машины  $T_f$  это требование не обязательно;
2. машина  $\bar{T}_f^N$  останавливается только в том случае, когда слово, полученное в результате ее работы, принадлежит множеству значений функции  $f$  (и, следовательно, является значением этой функции); в этом случае машина  $\bar{T}_f^N$  останавливается непосредственно после слова, являющегося значением функции  $f$ ; во всех остальных случаях машина  $\bar{T}_f^N$  никогда не останавливается.

Основная трудность, возникающая при построении машины  $\bar{T}_f^N$ , состоит в том, что в процессе своей работы машина  $T_f$  может изменить содержимое, вообще говоря, любой ячейки ленты, так что на ленте оказывается невозможным найти «безопасное» место для сохранения аргументов функции  $f$ . Эту трудность можно преодолеть следующим образом.

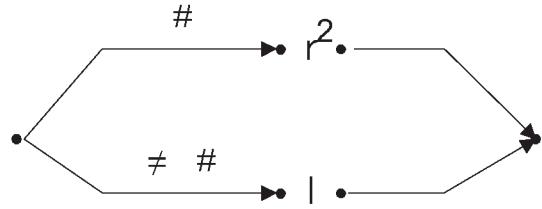
Выберем произвольную ячейку ленты и пометим ее, записав в нее знак  $\#$ . Все ячейки ленты, расположенные правее ячейки, помеченной  $\#$ , занумеруем слева направо последовательными натуральными числами, начиная с 1. Ячейки ленты, расположенные левее помеченной ячейки, занумеруем справа налево последовательными отрицательными числами, начиная с  $-1$  (см. схему).



«Раздвинем» ячейки правой части ленты, перенеся каждый знак, записанный в ячейку с номером 1, в ячейку с номером  $2i - 1$  (при этом, как видно из схемы, знак, записанный в ячейке с номером 1, останется в этой ячейке, знак, записанный в ячейке с номером 2, будет перенесен в ячейку с номером 3 и т. д.). Левую часть ленты отобразим на правую, перенеся буквы, записанные на левой части ленты, в образовавшиеся «зазоры» между знаками правой части, т. е. символ, записанный в ячейке  $-j$ , перенесем в ячейку с номером  $2j$  (см. схему). Заменим машину  $T_f$  машиной  $T'_f$ , которая моделирует машину  $T_f$  на новой ленте. Для этого в диаграмме машины  $T_f$  заменим каждое вхождение символа  $r$  диаграммой:

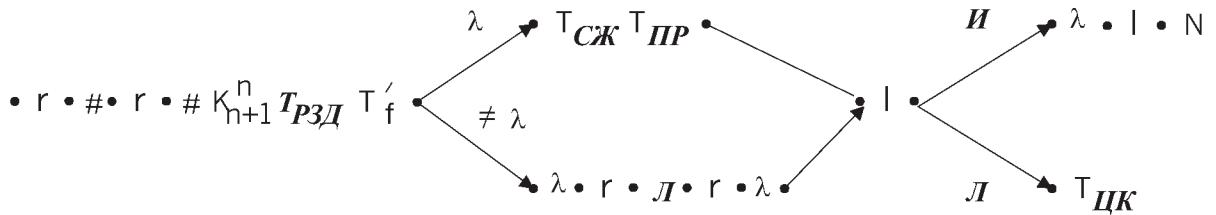


а каждое вхождение символа  $l$  диаграммой



Машине  $T'_f$  моделирует машину  $T_f$  и обладает тем свойством, что ее головка никогда не заходит левее ячейки, помеченной знаком  $\#$ . Это дает нам возможность использовать для сохранения аргументов функции  $f$  часть ленты, расположенную левее ячейки со знаком  $\#$ .

Итак, схему машины  $\bar{T}_f^N$  можно представить в виде:



Машина  $\bar{T}_f^N$  начинает работать в ситуации  
 $[\lambda w_1 \lambda w_2 \lambda \dots \lambda w_n(\lambda) \lambda >$ ,

где  $w_1, w_2, \dots, w_n$  — аргументы функции  $f$ . После работы  $r\#r\#rK_{n+1}^n$  получается ситуация  
 $[\lambda w_1 \lambda w_2 \lambda \dots \lambda w_n \lambda \# \# \lambda w_1 \lambda w_2 \dots \lambda w_n(\lambda) \lambda >$

Машина  $T_{P3D}$  раздвигает запись на правой части ленты, представив ее в виде

$[\lambda w_1 \lambda w_2 \lambda \dots \lambda w_n \lambda \# \# \lambda a_1 a_2 \dots a_{k_1} \lambda w_1 \lambda w_2 \dots \lambda w_n(\lambda) \lambda > \Rightarrow^*$

$[\lambda w_1 \lambda w_2 \lambda \dots \lambda w_n \lambda \# \# \lambda a_1 a_2 \dots a_{k_1} \lambda \lambda \lambda \dots (\lambda) \lambda >$

Далее начинает работать машина  $T'_f$ , которая либо никогда не останавливается, либо останавливается в ситуации

$[\lambda w_1 \lambda w_2 \lambda \dots \lambda w_n \lambda \# \# W(a) a' >$

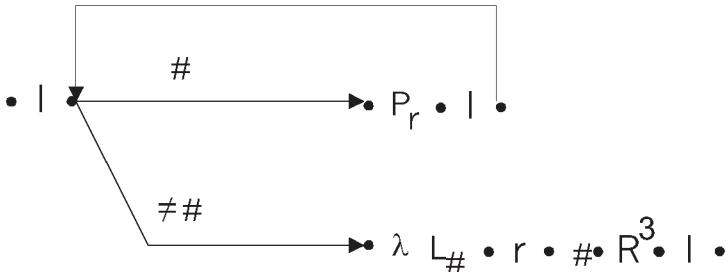
Если  $a \neq \lambda$ , то это значит, что  $T'_f$  не вычислила значения функции  $f$ ; если  $a = \lambda$ , то включается машина  $T_{CK}$ , которая «сжимает» слово, являющееся результатом работы машины  $T'_f$ , располагая его буквы в ячейках, идущих подряд, а не через одну, после чего машина  $T_{PR}$  убеждается, что слово, полученное в результате работы  $T'_f$ , содержит лишь знаки *выходного* алфавита машины  $T_f$ . Далее проверяется запись, произведенная машиной  $T_{PR}$ ; если машина  $T_{PR}$  записала знак И (истина), то он стирается, после чего включается машина  $N$ , формирующая заключительную ситуацию

$$[\lambda w_1 \lambda w_2 \lambda \dots \lambda w_n \lambda f(w_1, w_2, \dots, w_n)(\lambda) \lambda >$$

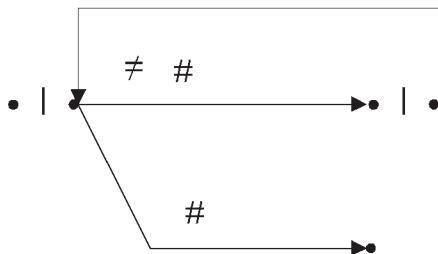
Если машина  $T_{\text{ПР}}$  записала знак Л (ложь), то включается машина  $T_\infty$ , которая никогда не останавливается.

Для завершения конструирования машины  $\bar{T}_f^N$  осталось разработать схемы машин  $T_{\text{РЗД}}, T_{\text{СЖ}}, T_{\text{ПР}}, T_\infty$  и  $\bar{N}$ . При конструировании этих машин мы будем предполагать, что рабочий алфавит машины  $T_f$  содержит буквы  $a_1, a_2, \dots, a_t$ .

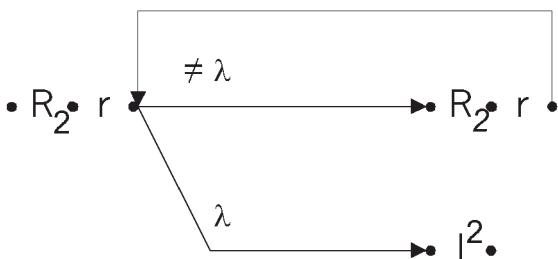
Машина  $T_{\text{РЗД}}$  описывается следующей схемой:



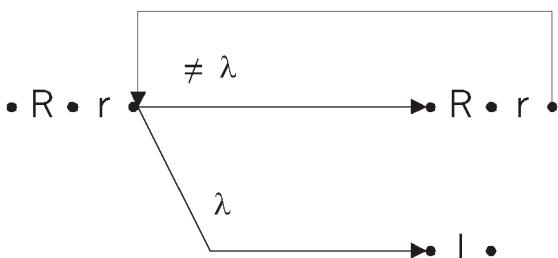
При составлении схемы машины  $T_{\text{РЗД}}$  использованы символы машин  $L_\#$ ,  $R_3$  и  $P_r$ . Машина  $L_\#$  сдвигает головку влево до тех пор, пока не встретится ячейка со знаком  $#$ ; ее схема имеет вид



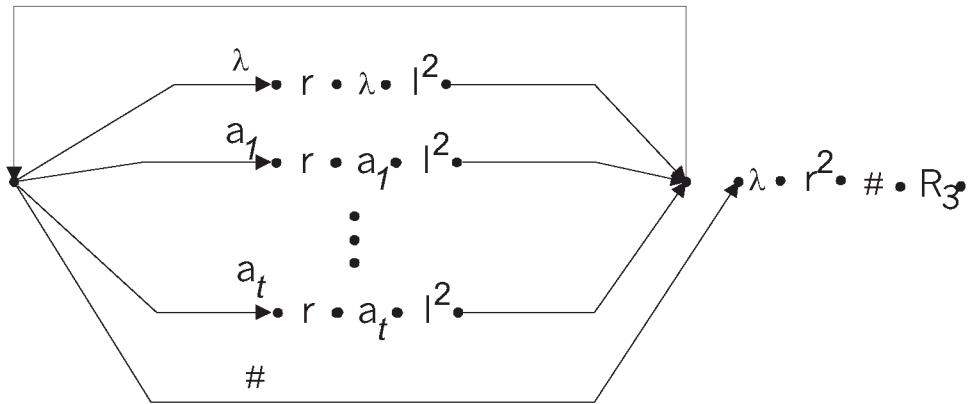
Машина  $R_3$  сдвигает головку вправо до тех пор, пока не встретятся три подряд идущие ячейки со знаком  $\lambda$ ; схема этой МТ имеет вид



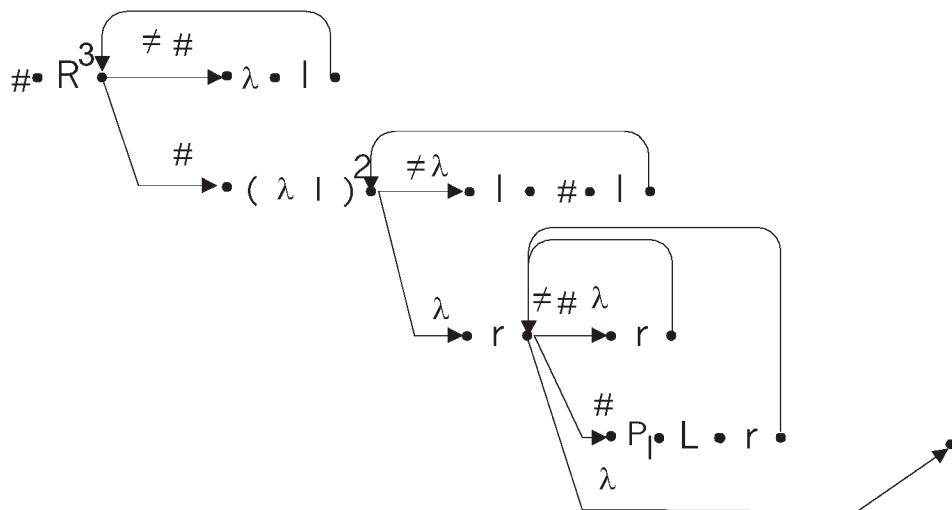
Машина  $R_2$  сдвигает головку вправо до тех пор, пока не встретятся две подряд идущие пустые ячейки; она описывается схемой



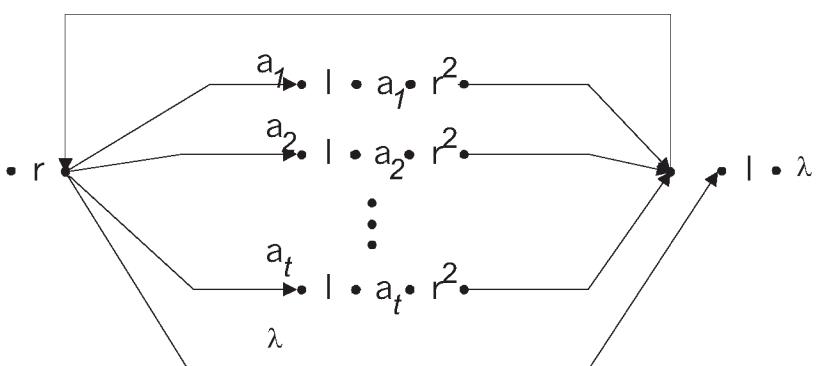
Машина  $P_r$  «освобождает» ячейку за очередной буквой; ее схема записывается в виде



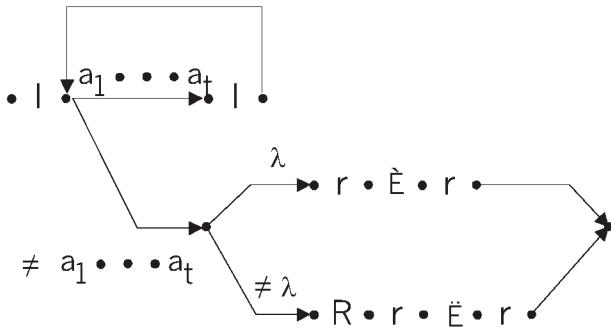
Машина  $T_{СЖ}$  производит действие, обратное действию машины  $T_{РЗД}$ . Ее схема может быть представлена в виде



Машина  $T_{СЖ}$  сначала стирает все буквы на ленте, расположенные правее значения функции  $f$ , вычисленного машиной  $T'_f$ , потом заменяет знаки, расположенные между буквами слова, являющегося значением функции  $f$ , символом  $\#$  и в заключение придвигает буквы, составляющие значение функции  $f$ , вплотную одну к другой, пока не будут стерты все  $\#$ . Схема машины  $P_l$ :

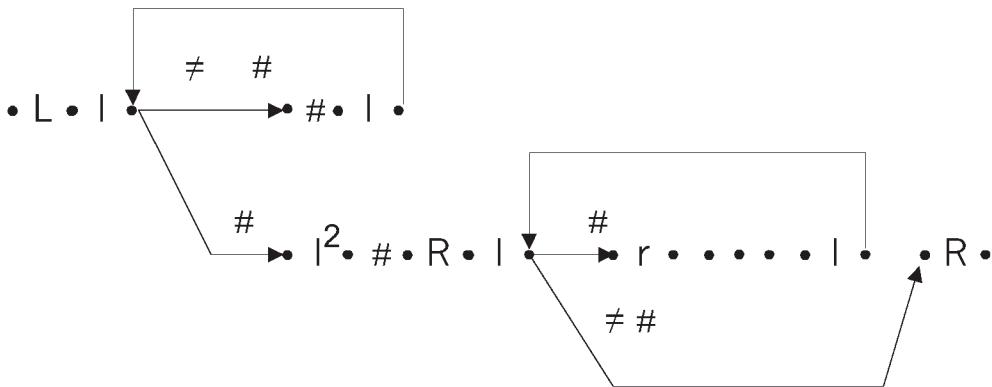


Машина  $T_{ПР}$  просматривает слово, вычисленное машиной  $T'_f$  в качестве значения функции  $f$ , и убеждается в том, что оно не содержит букв, не входящих в рабочий алфавит машины  $T_f$ . Приведем схему машины  $T_{ПР}$ :



Машина  $T_\infty$  реализует бесконечный цикл. Она начинает работать в тех случаях, когда аргументы не принадлежат области определения функции  $f$ . Машину  $T_\infty$  можно задать безусловным зацикливанием пары  $rl$ .

Действие машины  $N$  состоит в том, чтобы поместить вычисленное значение функции  $f$  непосредственно вслед за ее аргументами, расположенными на левой части ленты (т. е. левее  $\# \#$ ), убрав с ленты все «лишние» знаки. Похожая машина была сконструирована при рассмотрении примера 2.7.1. Построение схемы машины завершает процесс конструирования машины  $\bar{T}_f^N$ .



В заключение сконструируем машину  $T_f^N$ , которая нормированно вычисляет функцию  $f$ , не используя при этом дополнительного знака  $\#$ . Для этого воспользуемся второй теоремой Шеннона. Применив конструкцию рассмотренную при доказательстве этой теоремы (см. п. 2.4.4) к машине  $\bar{T}_f^N$ , получим машину  $\bar{T}_f^N$ , моделирующую машину  $\bar{T}_f^N$  над алфавитом  $A_1 = \{| \}$ .

Отметим, что машина  $\bar{T}_f^N$  использует в своей работе всего две буквы:  $|$  и  $\lambda$ . Машину  $T_f^N$  можно описать с помощью следующей схемы:

$$V_n \bar{T}_f^N E$$

где  $V_n$  — кодирующая машина, которая по исходной последовательности слов на ленте строит последовательность кодов этих слов над алфавитом  $A_1$ :

$$[\lambda w_1 \lambda \dots \lambda w_n(\lambda) \lambda] \xrightarrow{V^n}^* [\lambda w_1 \lambda \dots \lambda w_n \lambda \overline{\lambda w_1 \lambda \dots \lambda w_n(\lambda)} \lambda],$$

а  $E$  — декодирующая МТ, которая по коду значения функции  $f$  восстанавливает это значение в исходном алфавите:

$$[\lambda w_1 \lambda \dots \lambda w_n \lambda \overline{\lambda w_1 \lambda \dots \lambda w_n \lambda w(\lambda)} \lambda] \xrightarrow{E}^* [\lambda w_1 \dots \lambda w_n \lambda w(\lambda) \lambda]$$

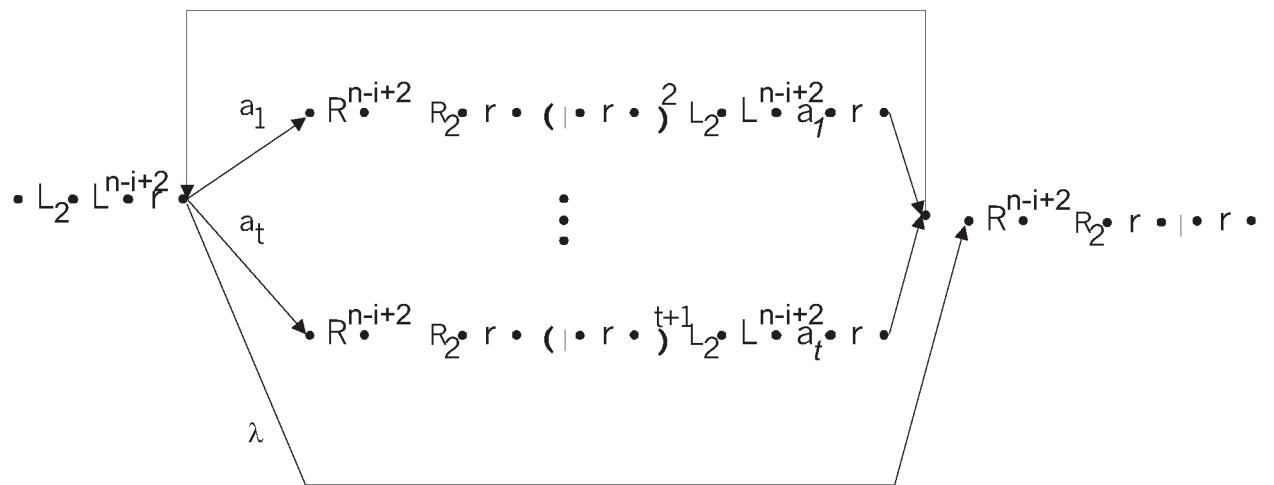
Схема машины  $V_n$  имеет вид:

$$r^2 | r V_{n_1} V_{n_2} \dots V_{n_n} l^2$$

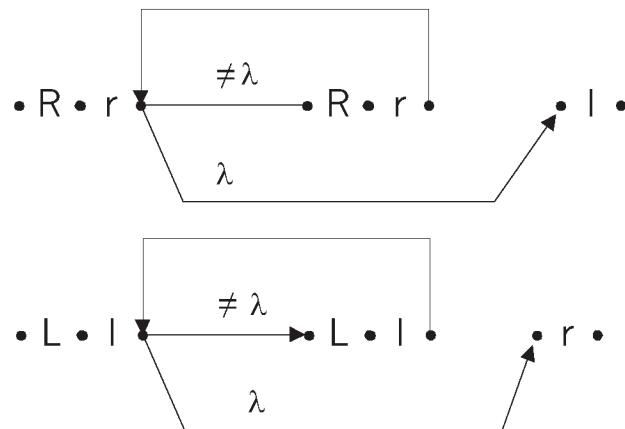
где каждая из машин  $V_{n_i}$  кодирует  $i$ -й аргумент:

$$[\lambda w_1 \lambda \dots \lambda w_n \lambda \lambda W(\lambda) \lambda] \xrightarrow{*} [\lambda w_1 \dots \lambda w_n \lambda \lambda W \bar{w} \bar{\lambda}(\lambda) \lambda]$$

Схема машины  $V_{n_i}$  такова:



В схеме машины  $V_{n_i}$  символы  $R_2$  и  $L_2$  обозначают МТ, которые сдвигают головку вправо (соответственно влево) до тех пор, пока не встретятся две подряд идущие пустые ячейки. Схемы этих машин соответственно имеют вид:



Построим схему декодирующей машины  $E$ . Ее можно представить в виде

$$E \quad \overline{\sqsubseteq} \quad r^2 R_{\lambda\lambda} L \bar{L} \bar{r} L_{\lambda\lambda} r | D N_1$$

В схеме машины  $E$  символы  $R_{\lambda\lambda}$  и  $L_{\lambda\lambda}$  представляют МТ, стирающие последовательность слов справа (соответственно слева) от головки:

$$[\lambda W_1 \bar{w} \bar{\lambda}(\lambda) W_2 \lambda] \xrightarrow{R_{\lambda\lambda}} [\lambda W_1 \bar{w} \bar{\lambda} \lambda \dots \lambda(\lambda) \lambda],$$

$$[\lambda W_1 \bar{w}(\lambda) \lambda] \xrightarrow{L_{\lambda\lambda}} [\lambda(\lambda) \lambda \dots \lambda \bar{w} \lambda \lambda]$$

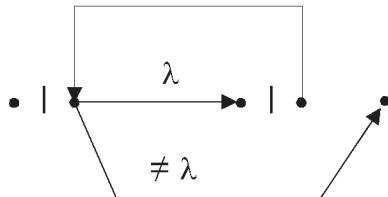
$D$  — символ МТ, восстанавливающей буквы рабочего алфавита машины  $T_f$  по их кодам над алфавитом  $A_1$ :

$$[\lambda(|) \lambda \dots \lambda w \lambda | \lambda] \xrightarrow{D} [\lambda | w(\lambda) \lambda]$$

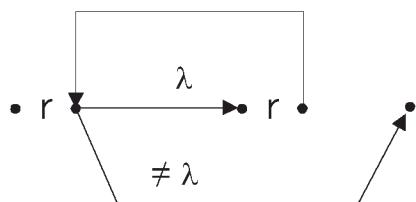
$N_1$  – символ МТ, помещающей декодированное слово  $w$ , являющееся значением функции  $f$ , непосредственно вслед за ее аргументами:

$$[\lambda|w(\lambda)\lambda > \stackrel{N_1}{\Longrightarrow}^* [\lambda w(\lambda)\lambda >$$

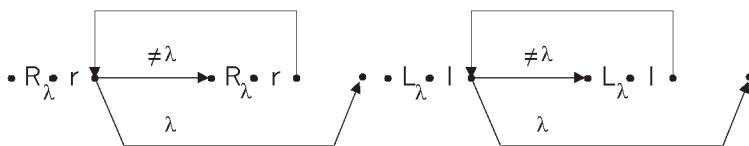
Машина  $\hat{L}$  описывается схемой



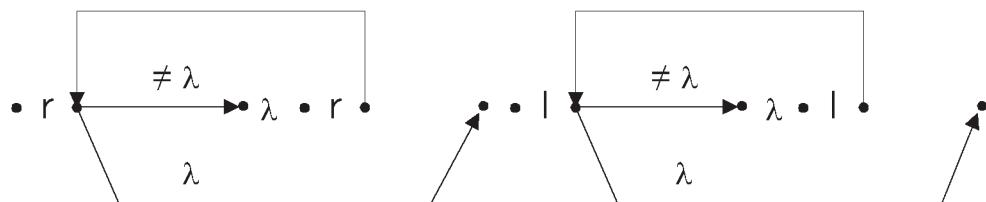
Она сдвигает головку влево до первой непустой ячейки. Аналогично можно определить машину  $\hat{R}$  с помощью схемы



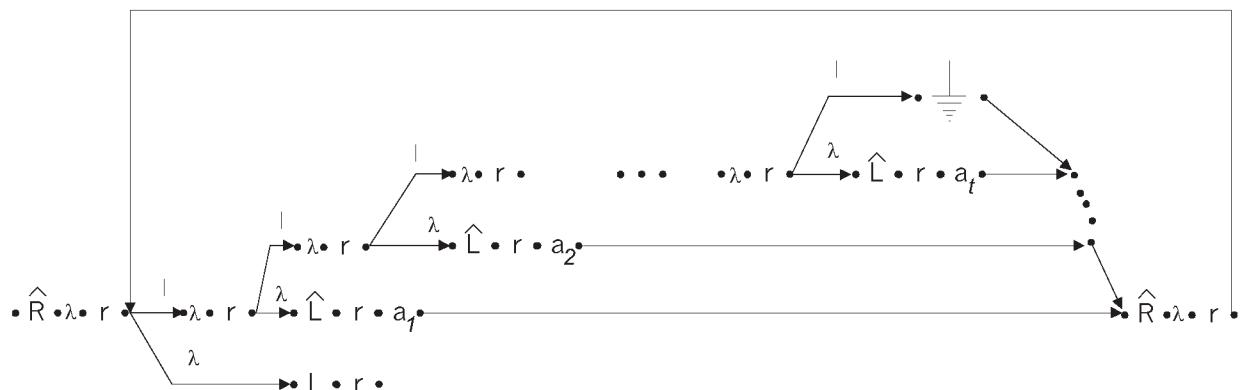
Схемы машин  $R_{\lambda\lambda}$  и  $L_{\lambda\lambda}$  соответственно имеют вид:



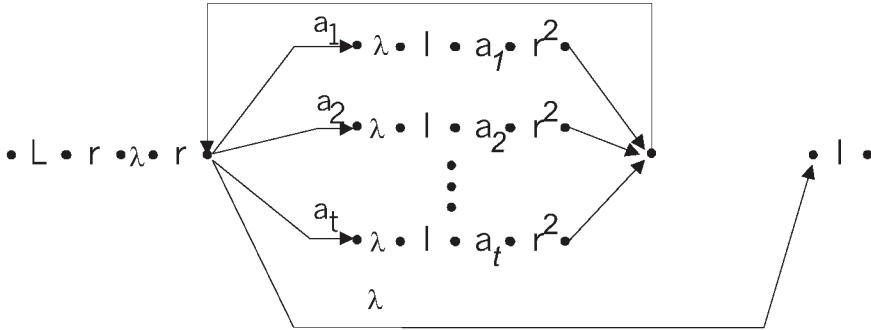
где через  $R_\lambda$  и  $L_\lambda$  обозначены МТ, стирающие одно слово справа (соответственно слева) от головки; эти МТ описываются схемами:



Для того, чтобы закончить описание машины  $E$ , приведем ее схему:



и схему машины  $N_i$ :



На этом завершается построение машины  $T_f^N$  и доказательство теоремы о нормированной вычислимости.

## 2.7.5 Некоторые фундаментальные результаты теории алгоритмов

Результаты, полученные Харелом, Цейтиным, Колмогоровым и др. [7, 4, 31]

# Лекция 14

## 2.8 Развитие алгоритмической модели Тьюринга

### 2.8.1 Понятие об универсальной машине Тьюринга

Ранее нами рассматривались МТ, каждая из которых выполняла свой строго определенный алгоритм: разные алгоритмы выполнялись разными машинами Тьюринга. Однако можно построить МТ, способную выполнить любой алгоритм, симулировав работу соответствующей МТ. Такая машина Тьюринга называется **универсальной**. Построение **универсальной машины Тьюринга** — один из важнейших для программирования результатов теории алгоритмов [7].

До начала работы универсальной МТ на ее ленту записывается последовательность четверок, представляющая программу машины, работу которой необходимо выполнить, и начальные данные. Универсальная МТ просматривает программу, записанную на ее ленте, и выполняет команду за командой этой программы, получая на ленте вслед за аргументами результат вычислений. Текст программы (в виде последовательности четверок) помещается на ленте слева от аргументов и, следовательно, при нормированных вычислениях сохраняется неизменным в процессе выполнения программы.

Реализация универсальной машины Тьюринга представляет собой весьма непростую задачу. Фактически она была осуществлена при доказательстве теорем Шеннона, когда строились машины Тьюринга, моделирующие любые другие МТ. Исследователем низкоуровневых алгоритмических моделей Дж. Тромпом [96], предложившим комбинаторно-логическую модель (CL), объем двоичного микрокода УМТ оценивается в 5495 бит (против 371 бит для CL-модели). Придуманная программистом Тромпом универсальная вычислительная машина UM реализует комбинаторную логику S-K, базируясь только на двух командах.

Построение универсальной МТ показало принципиальную возможность аппаратной реализации автоматического устройства, выполняющего любые алгоритмы по их описаниям. (Аппаратной реализацией какого-либо устройства мы называем построение этого устройства не на бумаге — в виде описания или проекта, а в виде реально работающей аппаратуры, т. е. механизмов и приборов). Построив аппаратуру универсальной МТ, мы получили бы возможность автоматического решения задач обработки данных: для решения каждой такой задачи было бы достаточно составить соответствующую программу (алгоритм) и поместить ее в виде последовательности четверок вместе с начальными данными на ленту универсальной МТ. Однако, поскольку мы уже убедились в неудобстве низкоуровневого программирования МТ, было бы желательно построить более удобную для человека диаграммную (точнее, схемную) версию УМТ. В п. 2.7 было показано, что программу МТ удобно составлять «сверху вниз» в виде последовательности схем. Каждой такой последовательности схем соответствует программа МТ в виде совокупности четверок, причем эта программа может быть эффективным образом построена по соответствующей последовательности схем [4]. Однако выполнять такое построение вручную очень сложно, да и не нужно, поскольку нетрудно составить алгоритм, реализующий это построение. Для составления указанного алгоритма необходимо прежде всего научиться записывать схемы МТ на ленте (ведь схемы, точнее говоря, их описания, представляют собой начальные данные рассматриваемого универсального алгоритма). Подобная процедура уже рассматривалась при доказательстве эквивалентности диаграмм и программ.

В общем случае под универсальностью понимают способность некоторой системы моделировать работу любой другой системы, когда описание последней подается ей на вход в закодированном виде. Это определение не является вполне строгим, поскольку не указывается, какие способы кодирования являются допустимыми. Можно определить кодирование таким образом, что оно будет включать в себя все вычисления, производимые моделируемой системой — разумеется, такой способ кодирования следует признать недопустимым. Однако в большинстве случаев допустимость выбранного способа кодирования можно оценить с точки зрения здравого смысла. В следующем параграфе будет приведен пример кодирования машин Тьюринга.

Доказать универсальность некоторой системы — значит показать, что она может моделировать поведение некоторой системы, универсальность которой доказана, либо целого класса систем, который является универсальным (например, всех машин Тьюринга).

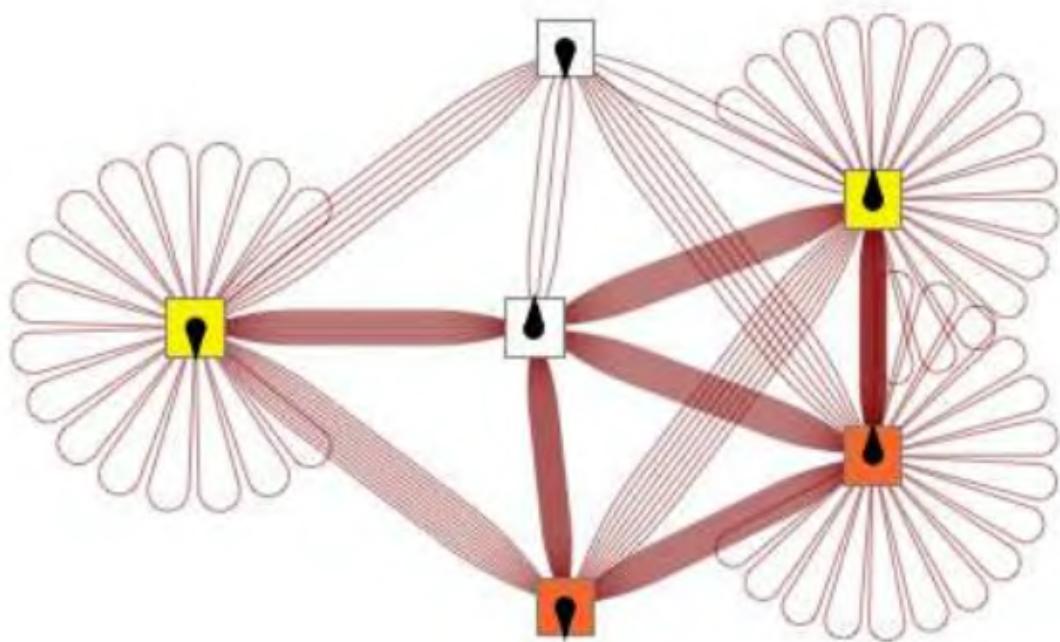
В 50-60-х годах прошлого века был сделан ряд открытий, связанных с универсальными машинами Тьюринга. Было доказано, что никакая машина Тьюринга с 2 состояниями и 2 буквами алфавита не может являться универсальной. Марвин Минский построил универсальную машину Тьюринга с 7 состояниями и 4 буквами алфавита. В течение долгого времени не было получено серьезных усилений этого результата.

Новый толчок изучению машин Тьюринга с малым числом состояний и букв алфавита придали исследования Стивена Вольфрама [106]. Он заметил, что системы, описываемые весьма простыми правилами, могут вести себя достаточно сложным образом. В частности, он предположил, что всякая простая система, поведение которой не является достаточно простым, является универсальной. Вольфрам построил универсальную машину Тьюринга с 2 состояниями и 5 буквами алфавита. Он также нашел 14 машин Тьюринга с 2 состояниями и 3 буквами алфавита, имеющих эквивалентное поведение, которое ему не удалось проанализировать до конца. Стивен Вольфрам и его единомышленники в 2007 году предложили приз в 25000 долларов за ответ на вопрос, является ли одна из

этих машин универсальной. Она имеет множество состояний  $(q_0, q_1)$ , алфавит  $(a_0, a_1, a_2)$  и описывается следующими правилами:

$$\begin{aligned} & (q_0, a_0, a_1, r, q_1) \\ & (q_0, a_1, a_2, l, q_0) \\ & (q_0, a_2, a_1, l, q_0) \\ & (q_1, a_0, a_2, l, q_0) \\ & (q_1, a_1, a_2, r, q_1) \\ & (q_1, a_2, a_0, r, q_0) \end{aligned}$$

Довольно быстро этот приз оказался востребованным: в том же 2007 году двадцатилетний британский студент Алекс Смит (Alex Smith), третьекурсник Бирмингемского университета, изучающий электротехнику, узнав о конкурсе, сразу же взялся за работу. Сведя задачу к эквивалентной, но более простой, Смит доказал универсальность "вольфрамовской" машины, за что и получил 25 тысяч долларов.-[109]. О сложности решения можно судить по диаграмме, отображающей первые две сти переходов:



На диаграмме ориентация "капелек" (вверх/вниз) символизирует состояния, а цвет квадратика (белый, желтый, оранжевый) указывает знак алфавита.

### 2.8.2 Линейная запись схем машин Тьюринга

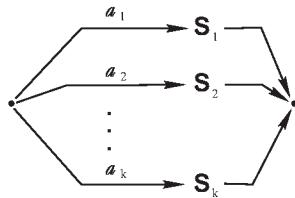
Трудность записи схем МТ на ленту состоит в том, что схемы двумерны, в то время как на ленту можно поместить лишь линейную запись. Таким образом, возникла задача линеаризации записи схем. Главным в этой задаче является не запись на ленту «цифровой

«фотографии» двумерной схемы МТ, а получение структурно-топологического, «векторного» представления. Элементы такого представления мы уже получали при доказательстве эквивалентности диаграмм и программ.

В п. 2.7.2 было отмечено, что схемы допускают всего три вида сочетаний составляющих их МТ: композицию, ветвление и цикл. Рассмотрим, как можно линеаризовать каждое из этих сочетаний.

*Композиция* состоит в последовательном выполнении нескольких действий, представленных в схеме символами соответствующих МТ. Эта часть записи схемы и так линейна, так как в случае композиции символы указанных МТ просто выписываются один за другим.

*Ветвление* изображается фрагментом схемы, который имеет вид



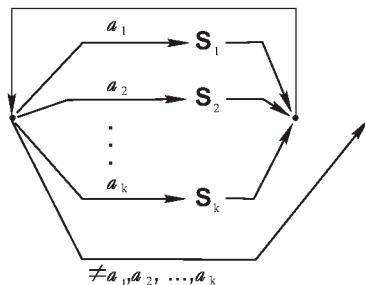
Здесь левая точка описывает состояние  $q$ , в котором происходит ветвление. Буквы  $a_1, a_2, \dots, a_k$  вместе с состоянием  $q$  образуют определяющие пары, указывающие, какая из ветвей должна быть выбрана. Символы  $S_1, S_2, \dots, S_k$  представляют собой фрагменты схемы, описывающие, какие действия должны быть выполнены в каждой из ветвей. Правая точка определяет состояние  $q'$  после ветвления.

В линейной записи рассматриваемый фрагмент схемы будем представлять в виде

**IF**    $a_1?S_1$     $\square$     $a_2?S_2$     $\square$     $\dots$     $\square$     $a_k?S_k$    **FI**

**IF** и **FI** соответственно обозначают начало и конец ветвления (состояния  $q$  и  $q'$  соответственно). Знак «?» отделяет букву, надписанную над стрелкой, от фрагмента схемы, который описывает действия в соответствующей ветви. Символ  $\square$  отделяет одну ветвь от другой.

*Цикл* изображается следующим фрагментом схемы:



Первая и последняя точки в этом фрагменте соответствуют состояниям  $q$  (начало цикла) и  $q'$  (конец цикла). Буквы  $a_1, a_2, \dots, a_k$  вместе с состоянием  $q$  составляют определяющие пары, по которым цикл должен быть продолжен (при этом выполняется одно из действий, описываемых фрагментами схемы  $S_1, S_2, \dots, S_k$ ). Буквы, отличные от  $a_1, a_2, \dots, a_k$ , вместе с состоянием  $q$  формируют определяющую пару, задающую выход

из цикла (переход в состояние  $q'$ ). Средняя точка тоже соответствует состоянию  $q$  (она введена в схему лишь для удобства).

В линейной записи описание цикла имеет вид

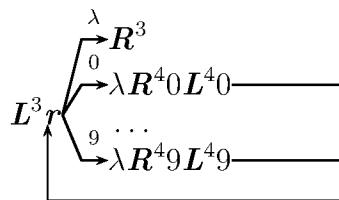
**DO**    $a_1?S_1$     $\square$     $a_2?S_2$     $\square$    ...    $\square$     $a_k?S_k$    **OD**

**DO** и **OD** обозначают соответственно начало и конец цикла, а знаки  $?$  и  $\square$  имеют тот же смысл, что и в случае ветвления.

### 2.8.3 Язык описания схем машин Тьюринга

Правила линейной записи схем МТ составляют основу *языка описания схем*, названного ОСТ (Описание Схем Тьюринга). Рассмотрим два примера описаний конкретных тьюринговых схем на языке ОСТ, вводя попутно некоторые конструкции этого языка.

**Пример 2.8.1.** В качестве первого примера рассмотрим описание программы машины  $K_3$ , выполняющей копирование третьего слова слева от головки. Будем считать, что слова на ленте МТ заданы над алфавитом  $A = \{0, 1, 2, 3, \dots, 9\}$ .



Название МТ на языке ОСТ может быть любым словом из русских и латинских букв, арабских цифр и знаков  $*$ ,  $+$ ,  $-$ ,  $/$ ,  $($ ,  $)$ ,  $\square$ ,  $@$ . Требование линейности записи исключает возможность использования индексов, так что машину  $K_3$  назовем К3. Также будем писать  $L^{**}4$  вместо  $L^4$ . Программа машины К3 на языке ОСТ имеет следующий вид:

МТ К3; "ПОСЛЕДОВАТЕЛЬНОСТЬ ЗНАКОВ, ЗАКЛЮЧЕННАЯ МЕЖДУ КАВЫЧКАМИ,  
НАЗЫВАЕТСЯ КОММЕНТАРИЕМ К ПРОГРАММЕ И МОЖЕТ БЫТЬ УДАЛЕНА  
ИЗ ПРОГРАММЫ БЕЗ ИЗМЕНЕНИЯ ЕЕ ЭФФЕКТА.  
МТ ПЕРЕД ИМЕНИЕМ ПРОГРАММЫ - УКАЗАТЕЛЬ НАЧАЛА ОПИСАНИЯ,  
КОНЕЦ ОПИСАНИЯ УКАЗЫВАЕТСЯ КАК END."

```

BEGIN
  ALPHABET: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9;
  "ОПИСАНИЕ ВНЕШНЕГО АЛФАВИТА МАШИНЫ К3"
  MT L; "ОПИСАНИЕ МТ, СИМВОЛ КОТОРОЙ ИСПОЛЬЗУЕТСЯ В ПРОГРАММЕ МАШИНЫ
  К3"
  BEGIN
    ALPHABET: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9;
    1; DO () ≠ _? 1 OD; "ЗНАК _ ПРЕДСТАВЛЯЕТ В ОСТ ЗНАК ПРОБЕЛА
    λ"
    END L; "КОНЕЦ ОПИСАНИЯ МАШИНЫ L"
  MT R;
  BEGIN
    ALPHABET: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9;
    r ; DO () ≠ _? r OD;
  
```

```

    END R;
L**3 ; r;
DO 0? a(_) ; R**4 ; a(0) ; L**4 ; a(0) ; r
    1? a(_) ; R**4 ; a(1) ; L**4 ; a(1) ; r
    2? a(_) ; R**4 ; a(2) ; L**4 ; a(2) ; r
    3? a(_) ; R**4 ; a(3) ; L**4 ; a(3) ; r
    4? a(_) ; R**4 ; a(4) ; L**4 ; a(4) ; r
    5? a(_) ; R**4 ; a(5) ; L**4 ; a(5) ; r
    6? a(_) ; R**4 ; a(6) ; L**4 ; a(6) ; r
    7? a(_) ; R**4 ; a(7) ; L**4 ; a(7) ; r
    8? a(_) ; R**4 ; a(8) ; L**4 ; a(8) ; r
    9? a(_) ; R**4 ; a(9) ; L**4 ; a(9) ; r
OD;
R**3;
END K3

```

**Пример 2.8.2.** Описание на языке ОСТ программы машины СЛОЖДР, выполняющей сложение двух обыкновенных дробей, в десятичной позиционной системе счисления. Каждая из дробей-слагаемых линейно записывается на ленте в виде слова  $w = u/v$ , где  $u$  и  $v$  — непустые слова над алфавитом  $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ , представляющие соответственно числитель и знаменатель дроби, слово  $v$  содержит хотя бы одну цифру, отличную от 0. Для описания программы машины СЛОЖДР потребуются МТ, выполняющие арифметические действия над целыми числами, МТ, копирующие слова на ленте, и некоторые другие МТ. Описания программ этих МТ может быть включено в состав рассматриваемой программы, либо заранее помещено на свободную (левую) часть ленты. В последнем случае описание соответствующей МТ в тексте программы заменяется описателем **LIB**, который означает, что соответствующую МТ нужно искать в левой части ленты. Программа СЛОЖДР может быть записана следующим образом:

МТ СЛОЖДР;

```

BEGIN
    АЛФАВЕТ: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, / ;
    МТ ЧИСЛ ; LIB ; "ВЫДЕЛЯЕТ ЧИСЛИТЕЛЬ ДРОБИ"
    МТ ЗНАМ ; LIB ; "ВЫДЕЛЯЕТ ЗНАМЕНАТЕЛЬ ДРОБИ"
    МТ ДРОБЬ ; LIB ;
        "СОСТАВЛЯЕТ ДРОБЬ ПО ЧИСЛИТЕЛЮ И ЗНАМЕНАТЕЛЕМ"
    МТ K2 ; LIB ;
    МТ K3 ; LIB ;
    МТ K4 ; LIB ;
    МТ K5 ; LIB ;
    МТ K6 ; LIB ;
    МТ K10 ; LIB ;
    МТ K ; LIB ; "КОПИРУЕТ СЛОВО"
    МТ НН ; LIB ; "ВМЕСТЕ С МАШИНОЙ НН НОРМИРУЕТ ВЫЧИСЛЕНИЯ"
    МТ КН ; LIB ; "ВМЕСТЕ С МАШИНОЙ КН НОРМИРУЕТ ВЫЧИСЛЕНИЯ"
    МТ + ; LIB ;
    МТ * ; LIB ;

```

```

МТ ÷ ; LIB ;
МТ НОД ;
BEGIN
    АЛФАВЕТ: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9;
    МТ - ; LIB ;
    МТ ≠ ; LIB ;
    МТ > ; LIB ;
    НН ; ≠ ; DO И? К2**2;
        IF И? К2**2; - ; К2; ≠
            Л? К; К3; - ; ≠
        FI
    OD;
    КН;
END НОД;
НН; К2; ЧИСЛ; К2; ЗНАМ; К5; ЧИСЛ;
К2; ЗНАМ; К5; К2; * ; К10; К4;
* ; К6; К10; * ; К4; К2; + ; К10;
НОД; ÷ ; К4; К3; ÷ ; К4; ДРОБЬ;
КН;
END СЛОЖДР

```

## Лекция 15

### 2.9 Модель фон Неймана

#### 2.9.1 Критика модели вычислений Тьюринга

Рассмотренные примеры программ на языке ОСТ показывают, что этот язык вряд ли можно считать удобным и эффективным средством описания алгоритмов.

Описание программы, выполняющей несложный алгоритм сложения двух обыкновенных дробей, заняло страницу, а ведь оно было приведено далеко не полностью: почти все МТ, через которые описывается алгоритм, были объявлены «библиотечными» (описатель **LIB**), т. е. уже составленными при описании других алгоритмов и записанными на свободной части ленты. На самом деле такое предположение справедливо лишь для копирующих МТ (машин  $K_n$  при различных  $n$ ) и для МТ НН и КН, обеспечивающих нормирование вычислений. Описания программ всех остальных МТ должны быть наряду с описанием программы НОД включены в программу явно, что, как легко видеть, увеличит ее текст примерно втрое.

Второй существенный недостаток языка ОСТ — *необходимость многочисленных копирований*. Если внимательно просмотреть описания МТ СЛОЖДР и НОД, можно заметить, что копирования составляют около половины всех действий, выполняемых при работе соответствующих МТ. Частые копирования не только резко увеличивают время выполнения программы, но и вызывают трудности при ее составлении; каждое слово присутствует на ленте в нескольких экземплярах, что усложняет проблему поиска нужных слов при составлении программы. В программе СЛОЖДР было употреблено семь разных

копирующих машин:  $K, K_2, K_3, K_4, K_5, K_6, K_{10}$ . Ясно, что количество таких машин потенциально бесконечно.

Существенным недостатком языка ОСТ является также *необходимость при составлении программы на этом языке выписывать ситуации на ленте*, так как если этого не делать, то можно легко запутаться в расположении данных на ленте, что приведет к ошибкам при составлении алгоритма.

Все перечисленные недостатки языка ОСТ обусловлены свойствами машины Тьюринга, т. е. модели вычислений, на основе которой разработан этот язык. Громоздкость описаний обусловлена тем, что МТ осуществляет *буквальную* обработку данных, т. е. длина описания программы пропорциональна числу букв сообщения, просматриваемых при выполнении алгоритма.

Необходимость частых копирований данных на ленте обусловлена выбранным способом нормированных вычислений, согласно которому каждое неэлементарное действие выполняется *над крайними правыми словами ленты*, так что перед выполнением каждого неэлементарного действия необходимо переместить к правому краю ленты слова, над которыми оно должно быть выполнено (количество слов зависит от конкретного действия). Необходимость постоянного слежения за ситуациями на ленте при составлении программы возникает в связи с тем, что положение слов на ленте определяется *относительно текущего положения головки*, которое все время меняется.

Итак, мы показали, что недостатки языка ОСТ как средства для описания алгоритмов предопределены особенностями выбранной модели вычислений. То, что удобно для построения красивой математической теории алгоритмов, малопригодно для составления реальных алгоритмов и программирования. Поэтому для разработки более удобных и эффективных средств описания алгоритмов необходимо прежде всего построить более совершенную модель вычислений. Такая модель была построена Дж. фон Нейманом в середине 40-х годов. Машина фон Неймана, к рассмотрению которой мы переходим, была получена путем усовершенствования машины Тьюринга.

## 2.9.2 Адреса и имена

Рассмотрим нормированное вычисление функции  $f(w_1, w_2, \dots, w_n)$  на универсальной машине Тьюринга (УМТ). До начала работы УМТ на ее ленту записывается программа  $P_f$  нормированного вычисления функции  $f$  и начальные данные. Головка УМТ помещается непосредственно за начальными данными. Для удобства будем помещать программу на ленте между специальными символами  $\boxed{\text{н}}$  и  $\boxed{\text{к}}$ . Будем считать, что программы внешних МТ, используемых в программе  $P_f$ , расположены слева от ячейки с символом  $\boxed{\text{н}}$  и ограничены слева ячейкой с символом  $\boxed{\text{д}}$ . Тогда начальная ситуация на ленте будет иметь вид

$$[\lambda \boxed{\text{д}} \dots \boxed{\text{н}} P_f \boxed{\text{к}} \lambda w_1 \lambda w_2 \dots \lambda w_n(\lambda) \lambda >$$

где многоточие между символами  $\boxed{\text{д}}$  и  $\boxed{\text{н}}$  обозначает часть ленты, занятую внешними программами.

Нормированность вычисления функции  $f$  программой  $P_f$  означает, что в процессе выполнения программы  $P_f$  часть ленты, расположенная левее ячейки с символом  $\boxed{\text{д}}$ , будет оставаться пустой. Более того, головка никогда не будет попадать в эту часть

ленты, и если, например, эту часть ленты «отрезать» и убрать, то УМТ при выполнении программы этого даже «не заметит». Все данные, появляющиеся на ленте в результате работы программы  $P_f$ , размещаются в ячейках ленты, расположенных правее ячейки, содержащей символ  $\boxed{\text{к}}$ .

Воспользуемся этим обстоятельством, чтобы избавиться от пресловутой необходимости постоянного слежения за ситуациями на ленте при составлении программы. Для определения положения слова на ленте не будем использовать рабочую ячейку, указываемую текущим положением головки, поскольку оно все время меняется. В качестве начала отсчета примем некоторую фиксированную ячейку ленты, например, содержащую символ  $\boxed{\text{к}}$ : слову, расположенному непосредственно вслед за этой ячейкой, сопоставим номер 1, следующему слову — номер 2 и т. д. Номер, сопоставленный некоторому слову, назовем *адресом* этого слова. Таким образом, каждому значению (слову), расположенному на ленте, ставится в соответствие его *адрес*, который не меняется в процессе выполнения программы. Подчеркнем, что приписанные словам адреса, нигде в явном виде не записываются: ни в самих словах, ни где-либо отдельно. Чтобы получить адрес конкретного слова, надо «проехать со счетчиком в руках» от первого слова к заданному. То есть понятие адреса вторично; оно задано конструктивным образом, так как адрес каждого слова может быть сгенерирован по известному алгоритму.

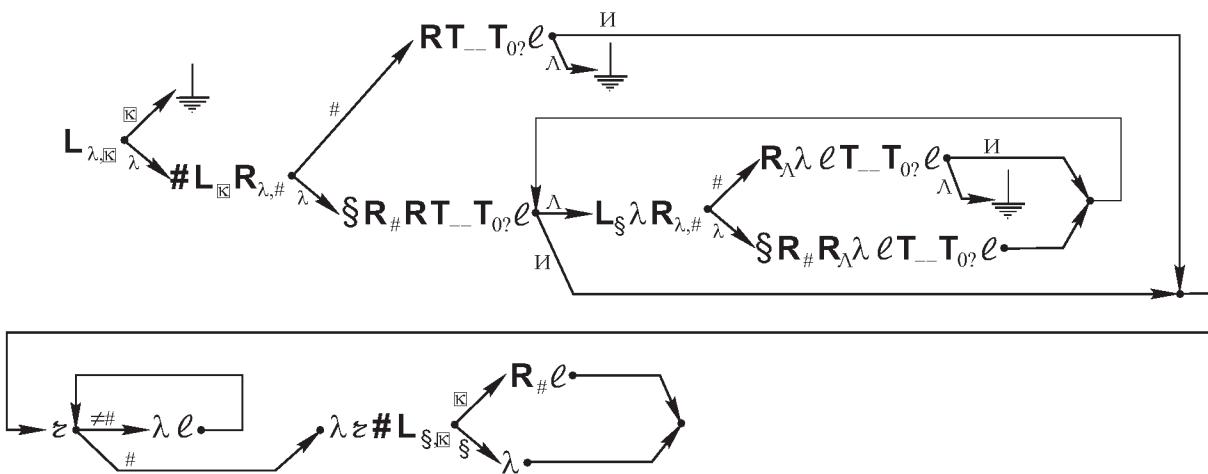
Наличие постоянных адресов у данных позволяет сконструировать МТ, которая по адресу некоторого слова, записанному на ленте в виде целого десятичного числа, устанавливает головку на пробеле, расположенному непосредственно вслед за этим словом. Мы назовем эту МТ именем  $@$  (поиск по адресу). Действие этой машины описывается следующей парой ситуаций:

$$\begin{aligned} & [\lambda \boxed{\text{д}} \dots \boxed{\text{н}} P_f \boxed{\text{к}} w_1 \lambda w_2 \lambda \dots \lambda w_k \lambda w_{k+1} \dots \lambda w_n \lambda A_k(\lambda) \lambda] > \\ \xrightarrow{@}^* & [\lambda \boxed{\text{д}} \dots \boxed{\text{н}} P_f \boxed{\text{к}} w_1 \lambda w_2 \lambda \dots \lambda w_k(\lambda) w_{k+1} \dots w_n \lambda \# \lambda] > \end{aligned}$$

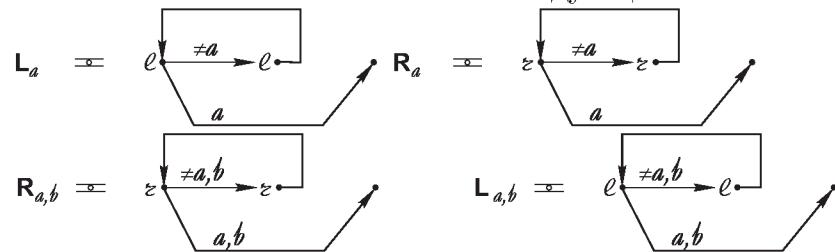
где  $A_k$  — адрес слова  $w_k$ , т. е. десятичная запись целого числа  $k$ ,  $\#$  — некоторый специальный символ, используемый для разметки ленты. В заключительной ситуации машины  $@$  адрес  $A_k$  оказывается стертым, а пометка  $\#$  сохраняется, чтобы облегчить работу машин, использующих поиск по адресу, т. е. подобно собаке-пойнтеру машина  $@$  «делает стойку» у искомого слова.

Схема машины  $@$  может быть сконструирована из машин  $R_a, L_a, R_{a,b}$ , десятичного декрементора  $T_{-1}$  и предиката  $T_0?$  (состоит ли слово из нулей?).

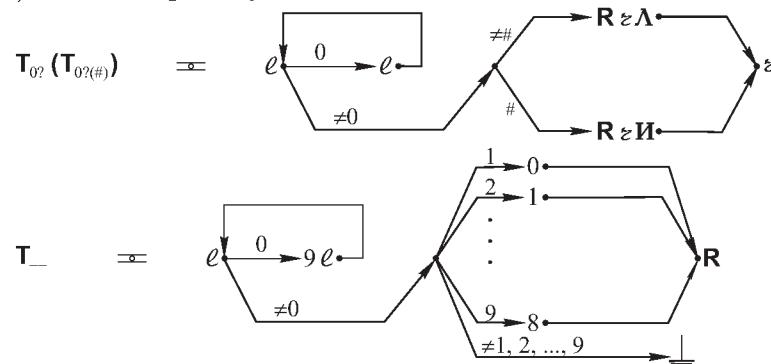
Схема машины  $@$  имеет вид



В схеме машины @ использованы следующие МТ:



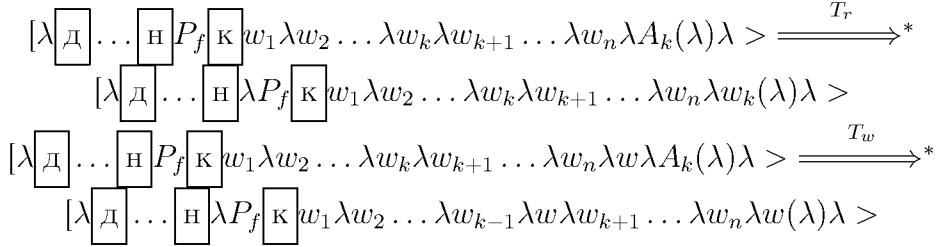
$(a, b)$  — некоторые буквы.



Из приведенных схем видно, что машина  $T_0?$  выясняет, состоит ли слово, к которому она применяется, из одних нулей или нет, т. е. проверяет число, изображаемое этим словом на равенство нулю; машина  $T_-$  преобразует слово, к которому она применяется, в новое слово, изображающее число на единицу меньшее, чем исходное слово, т. е. заменяет число предыдущим.

Используя машину @, нетрудно сконструировать машины  $T_r$  и  $T_w$ , первая из которых записывает вместо адреса слово, расположенное по этому адресу, а вторая заменяет слово, записанное по адресу, указанному на ленте, на слово, непосредственно предшествующее

слову, представляющему адрес:



Заметим, что машины  $T_r$  и  $T_w$  моделируют последовательное запоминающее устройство — хранилище слов на участке ленты машины Тьюринга, фактически реализуя их копирование. При этом они значительно мощнее машин копирования типа  $K_n$  с их постоянным плечом ( $n = \text{const!}$ ), поскольку чтение и запись осуществляются в ячейки с переменными номерами.

На основе машин  $T_r$  и  $T_w$  можно сконструировать машины  $T_{r@}(k)$  и  $T_{w@}(k)$ , где  $k$  — адрес слова. В самом деле, диаграммы этих машин имеют вид

$$T_{r@}(k) = W_{A_k}; T_r$$

$$T_{w@}(k) = W_{A_k}; T_w$$

где машина  $W_{A_k}$  записывает на ленту слово, изображающее десятичное число  $k$ , т. е. изображение числа из текста программы переносится на ленту МТ. Машины  $T_{r@}(k)$  и  $T_{w@}(k)$  позволяют существенно упростить программы МТ, сделав их более понятными. Например, они дают возможность записать программу машины НОД в виде

**МТ НОД**

**BEGIN**

**ALPHABET:** 0, 1, 2, 3, 4, 5, 6, 7, 8, 9;

**МТ** - ; **LIB**;

**МТ** ≠ ; **LIB**;

**МТ** > ; **LIB**;

**НН** ; ≠ ; **DO И?**  $T_{r@}(3)$ ;  $T_{r@}(4)$ ; > ;

**IF И?**  $T_{r@}(3)$ ;  $T_{r@}(4)$ ; - ;  $T_{w@}(3)$

□ **Л?**  $T_{r@}(4)$ ;  $T_{r@}(3)$ ; - ;  $T_{w@}(4)$

**FI**;

$T_{r@}(3)$ ;  $T_{r@}(4)$ ; ≠

**OD**;

КН;

**END НОД**

В этой программе все преобразования осуществляются над словами с адресами 3 и 4, причем в процессе выполнения программы значения этих слов меняются (в словах с адресами 1 и 2 сохранены исходные значения). В дальнейшем такие слова будем называть *переменными*, а их адреса — *адресами переменных*. Слова, значения которых в процессе выполнения программы не меняются (в программе машины НОД это слова с адресами 1 и 2), будем называть *константами*, а их адреса — *адресами констант*.

Текст программы становится еще более понятным, если вместо адресов переменных и констант использовать имена<sup>2</sup>. Именем переменной или константы мы будем назы-

<sup>2</sup>Обозначение неизвестных величин и коэффициентов именами было предложено известным математи-

вать слово над произвольным алфавитом, сопоставленное адресу этой переменной или константы. Сопоставление может быть осуществлено с помощью *таблицы имен*, т. е. последовательности слов, в которой перечислены все используемые в программе имена, причем вслед за каждым именем помещен соответствующий адрес. Таблицу имен мы расположим на ленте УМТ непосредственно перед программой  $P_f$ , отметив ее начало ячейкой, в которую записан специальный символ  $\boxed{t}$  (начало таблицы имен). Таким образом, программы внешних МТ располагаются на ленте между ячейками с символами  $\boxed{d}$  и  $\boxed{t}$ . Мы будем считать, что таблица имен составляет часть программы  $P_f$ . При использовании имен машины  $T_{r@}(k)$  и  $T_{w@}(k)$  заменяются соответственно машинами  $\uparrow(n)$  и  $\downarrow(n)$ , которые сначала находят имя  $n$  в таблице имен, затем по этому имени определяют адрес  $k$ , после чего работают как машины  $T_{r@}(k)$  и  $T_{w@}(k)$  соответственно. Знаки  $\uparrow(n)$  и  $\downarrow(n)$  символизируют загрузку значений из именованной памяти машины фон Неймана на регистры и, соответственно, их запоминание под указанными именами. Использование имен позволяет записать программу машины НОД в более понятном виде

**МТ НОД;**

**BEGIN**

**ALPHABET:** 0, 1, 2, 3, 4, 5, 6, 7, 8, 9;

**NAMES:** u, v, x, y;

**МТ - ; LIB;**

**МТ ≠ ; LIB;**

**МТ > ; LIB;**

$\uparrow(u); \downarrow(x); \uparrow(v); \downarrow(y); \uparrow(x); \uparrow(y);$

$\neq; \text{DO И? } \uparrow(x); \uparrow(y); >;$

**IF И?  $\uparrow(x); \uparrow(y); -; \downarrow(x)$**

$\quad \boxed{\square} \text{ Л? } \uparrow(y); \uparrow(x); -; \downarrow(y)$

**FI;**

$\uparrow(x); \uparrow(y); \neq$

**OD;**

**NORM(u, v, x)**

**END НОД**

В программу добавляется строка NAMES, по которой составляется таблица имен; машины НН и КН заменяются машиной NORM, которой сообщаются имена аргументов  $u, v$  и результата  $x$ ; машина NORM нормирует вычисление, реализуемое машиной НОД, сохраняя на ленте лишь слова с именами  $u, v, x$  в указанном порядке.

Итак, мы показали, что использование имен существенно проясняет текст программы и позволяет читать его без параллельного рассмотрения ситуаций на ленте. Однако программы все равно остаются громоздкими и неудобочитаемыми, что побуждает к дальнейшему усовершенствованию языка программирования.

### 2.9.3 Построение процессора фон Неймана

В программах, рассмотренных в качестве примеров, широко использовались внешние МТ, программы которых считались заранее записанными на специально отведенном для

---

ком Франсуа Виетом еще в 1591 г.

этого участке ленты универсальной МТ (назовем этот участок *D-лентой*). Использование внешних МТ позволило существенно упростить указанные программы, так как они записывались уже не через элементарные действия  $r$ ,  $l$ ,  $a$ , а через более содержательные действия, определяемые внешними МТ.

Внешние МТ программируются и записываются на *D*-ленту до того, как они будут применены при составлении очередной программы. Следовательно, они должны реализовывать действия, которые будут применимы при составлении достаточно широкого класса программ. Отметим, что такие действия существуют. Это, например, простейшие математические действия над целыми числами ( $T_{+1}$ ,  $T_{-1}$ ) и специальные служебные отношения-предикаты ( $T_0?$ ).

Универсальная МТ, *D*-лента которой содержит программы, реализующие арифметические действия, имеет *качественно новый уровень* по сравнению с универсальной МТ с пустой *D*-лентой: алгоритмы, связанные с выполнением арифметических действий над целыми числами (например, алгоритм Евклида), описываются на ней гораздо более простыми программами. Чтобы подчеркнуть это обстоятельство, назовем такую универсальную МТ *арифметическим процессором*.

Таким образом, предварительно разработав и записав на *D*-ленте набор внешних МТ, реализующих действия, используемые при составлении некоторого класса тьюринговых программ, мы получаем *процессор* для этого класса программ, позволяющий существенно упростить описания программ рассматриваемого класса и сократить процесс их разработки. Действия, реализуемые МТ, записанными на *D*-ленту, будем называть *элементарными действиями процессора* или *операциями*. Все внешние МТ, реализующие операции, имеют одинаковый рабочий алфавит, называемый *рабочим алфавитом процессора*. Этот алфавит является рабочим алфавитом всех программ, выполняемых на процессоре.

Каждая операция применяется к одному или нескольким словам над рабочим алфавитом процессора, называемых *операндами*. Количество operandов определяет *местность* операции (например, операция сложения имеет два операнда и потому является двуместной). Операция выполняется нормированно: *результат* помещается за operandами. Перед выполнением операции ее operandы необходимо поместить в требуемом порядке непосредственно перед свободным краем ленты, так как иначе результат операции и, возможно, ее промежуточные результаты (после выполнения операции они стираются с ленты) запишутся на месте слов, которые должны быть сохранены. Это существенно усложняет функции РИМ и АДР, аналогичные машинам языка ОСТ  $\uparrow$  и  $\downarrow$ , вводя в них такие действия как поиск свободного конца ленты, а также приводит к необходимости иметь полный набор копирующих МТ среди операций процессора. Отметим, что сами функции РИМ и АДР тоже являются операциями процессора.

Упрощение достигается путем использования свободного края ленты, расположенного левее ячейки с символом  $\boxed{д}$ . Если operandы поместить на эту часть ленты, то в состав функций РИМ и АДР будет входить всего по одной копирующей МТ. Таким образом, процессор наряду с *D*-лентой, на которой хранятся программы его операций, наделяется также *R*-лентой (так мы назовем часть ленты, расположенную левее ячейки с символом  $\boxed{д}$ , на которую помещаются operandы перед выполнением операции), и *S*-лентой, на которой записываются программа и ее данные. Удобно перенормировать операции таким образом, чтобы их результат также помещался на *R*-ленте *не вслед* за operandами, а *вместо* них. Обмен данными между *R*-лентой и *S*-лентой осуществляется операциями РИМ( $n$ )

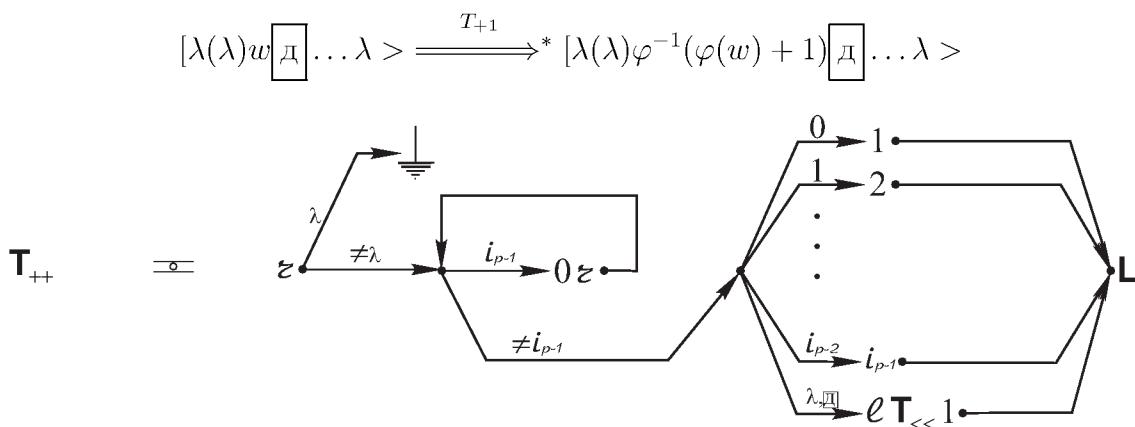
и АДР( $n$ ), где  $n$ -имя операнда.

Итак, для построения процессора необходимо:

- выбрать и зафиксировать *рабочий алфавит* процессора; обычно в качестве рабочего алфавита будем брать множество  $A_p = \{0, 1, 2, \dots, p - 1\}$  первых  $p$  неотрицательных чисел (начиная с 0); слова над алфавитом  $A_p$  допускают естественную интерпретацию как неотрицательные целые числа в позиционной системе счисления с основанием  $p$ ;
- зафиксировать конечное или бесконечное *множество допустимых слов* над алфавитом  $A_p$ ; если множество допустимых слов конечно и включает лишь слова фиксированной длины  $k$  (либо слова, имеющие длину не больше, чем  $k$ ), то удобно добавить к множеству допустимых слов еще одно слово, называемое «переполнением» —  $\top$ , которое означает, что в результате выполнения операции получилось слово, имеющее длину, большую, чем  $k$ ; множество допустимых слов может также содержать слово  $\perp$ , обозначающее неопределенное значение операнда, см. п. 3.1;
- составить и записать на  $D$ -ленту программы  $MT$ , выполняющие операции процессора (включая операции РИМ и АДР);
- сообщить *обозначения операций* универсальной МТ (точнее — управляющей программе этой универсальной МТ), на основе которой строится процессор.

В число операций процессора могут входить *отношения*, т. е. такие операции, которые по одному или нескольким допустимым словам (аргументам отношения) вычисляют логическое значения **И** (истина) или **Л** (ложь); количество аргументов каждого отношения постоянно и называется его *местностью*. Строго говоря, отношения связывают числовые операнды, имеют истинностные значения и не являются операциями, поскольку размыкают множество допустимых операндов.

В качестве примера процессора можно рассмотреть процессор с множеством допустимых слов  $A_p^*$  ( $p$  — фиксированное натуральное число), двумя одноместными операциями  $T_{+1}$  и  $T_{-1}$  и одним одноместным отношением  $T_0?$ . Схемы МТ, вычисляющих  $T_{+1}(a)$  и  $T_0?(a)$ , аналогичны упомянутым ранее (поскольку  $R$ -лента является зеркальным отражением ленты, рассматривавшейся выше, в этих схемах необходимо все действия  $l$  заменить на  $r$ , все  $r$  — на  $l$ , все  $R$  — на  $L$ ). Схема машины  $T_{+1}$  (эта машина увеличивает слово, трактуемое как число в позиционной системе счисления с основанием  $p$ , на единицу) такова:

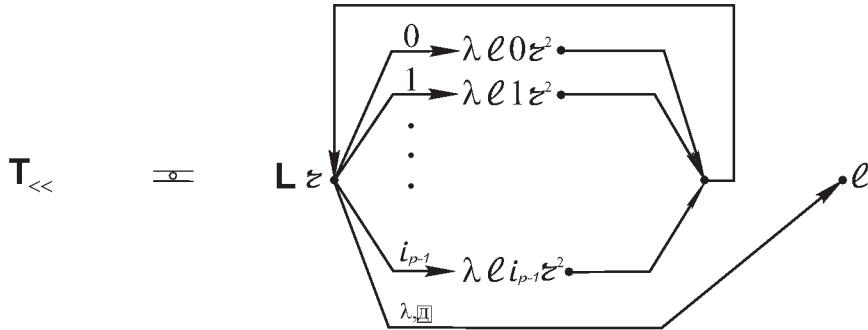


В последней схеме использована машина  $T_{<<}$ , выполняющая действие

$$[\lambda a_1 a_2 \dots a_{l-1} (a_l) \boxed{d} \dots \lambda] > \xrightarrow{T_{<<}}^* [\lambda a_1 a_2 \dots a_{l-1} a_l (\lambda) \boxed{d} \dots \lambda] >$$

где  $a_i \in \overline{A}_p$ .

Эта машина описывается следующей схемой:



## 2.9.4 Согласование процессоров

Построенные процессоры могут использоваться по отдельности, или вместе. В последнем случае необходимо уточнить понятие *согласованности*.

**Определение 2.9.1.** Два процессора называются **согласованными**, если определены функции (операции) перевода всех допустимых слов одного процессора в допустимые слова другого процессора и обратно.

При согласовании двух процессоров их  $S$ -ленты объединяются в одну. В результате получается конструкция, состоящая из управляющего устройства,  $S$ -ленты, двух  $R$ -лент и двух  $D$ -лент. Назовем ее двуленточным процессором. Отметим, что каждая из двух лент двуленточного процессора содержит операцию переписи слов с «чужой»  $R$ -ленты на «свою» с одновременным их преобразованием к «своему» допустимому виду.

Несколько попарно согласованных процессоров образуют *многоленточный* или  $n$ -ленточный процессор.

Если управляющую программу и все  $D$ -ленты многоленточного процессора заменить новым управляющим устройством, которое может выполнять все операции рассматриваемых  $D$ -лент, получится новый тип процессора, называемый **процессором фон Неймана**.

Таким образом, процессор фон Неймана состоит из полубесконечной  $S$ -ленты, на которую записывается программа и ее данные, одной или нескольких  $R$ -лент, на которые вызываются операнды соответствующих процессоров перед выполнением операций, и управляющего устройства, просматривающего и интерпретирующего программу, записанную на  $S$ -ленте. Исполнение программы состоит в переносе операндов с  $S$ -ленты на одну из  $R$ -лент или с одной  $R$ -ленты на другую, в выполнении требуемых операций над операндами, вызванными на одну из  $R$ -лент, и в записи результатов операций на  $S$ -ленту.

## 2.9.5 Машина фон Неймана

*Машиной фон Неймана* называется аппаратная реализация процессора фон Неймана. Машина фон Неймана состоит из управляющего устройства, устройства памяти (аппаратная реализация  $S$ -ленты) и одного или нескольких регистров (аппаратная реализация  $R$ -лент).

Далее в нашем курсе будет изложен язык описания программ для четырехленточного процессора фон Неймана и соответствующей машины фон Неймана. Описание языка ориентировано на программиста. Оно полностью определяет процессор (машину) фон Неймана как некоторое абстрактное устройство, обеспечивающее выполнение (интерпретацию) программ.

Прежде чем переходить к описанию конструкций языка программирования, необходимо рассмотреть один принципиальный вопрос. Устройство памяти машины фон Неймана имеет два существенных различия с *S*-лентой. Во-первых, в отличие от *S*-ленты, устройство памяти содержит лишь *конечное* число ячеек *ограниченного размера*, так что не любая вычислимая по Тьюрингу (Маркову, Посту, Черчу, Флойду, Тромпу и т. д.) функция вычислима на машине фон Неймана. Во-вторых, данные, записанные в ячейки устройства памяти, недоступны для непосредственного восприятия человеком. Следовательно, в состав машины фон Неймана должны входить устройства записи сообщений в память (устройство ввода) и вывода данных из памяти на носитель, доступный для восприятия органами чувств человека (устройство вывода). Все устройства ввода так или иначе преобразуют сообщения разного вида во внутримашинные данные (двоичные слова: именно фон Нейман предложил использовать двоичную систему счисления при аппаратной реализации ЭВМ!). Например, микрофон преобразует поперечные звуковые колебания воздуха в релевантные аналоговые электрические сигналы. Для их обработки цифровыми устройствами, эти сигналы должны быть преобразованы в дискретную форму звуковой платой. Современные устройства вывода способны воздействовать не только на зрение, но и на слух, обоняние, осязание, вкус. Кроме того, используя инерцию органов зрения, возможно создание движущихся изображений. По этой же причине даже если бы мы смогли чувствовать данные во внутримашинном представлении, мы не поспевали бы за их слишком быстрым изменением. Следовательно, язык описания программ для машины фон Неймана должен содержать средства управления устройствами ввода и вывода, преобразующими сообщения в данные и наоборот.

Итак, в результате аппаратной реализации процессора фон Неймана мы перешли от сообщений к данным, потеряв абсолютную вычислимость и лишившись возможности непосредственно созерцать процесс обработки данных в ЭВМ и тем более участвовать в нем. Но приобретения более значительны: мы получили возможность быстрой и безошибочной автоматической обработки данных без участия человека. Абсолютная вычислимость в машине фон Неймана заменена частичной. Вычислимым по фон Нейману считается все то, что может быть получено за приемлемое время на доступных ресурсах памяти и, весьма часто, в пределах ограниченного бюджета имеющихся средств.

В настоящее время машина фон Неймана уже не имеет чисто аппаратных ассоциаций. Она чаще всего предоставляется программисту в виде многослойной надстройки программных средств [22], последним из которых является языковый процессор высокого уровня.

# Глава 3

## Концепция языка программирования для машины фон Неймана

### Лекция 16

#### 3.1 Элементы дейкстровской нотации

##### 3.1.1 Требования к структуре программ для машины фон Неймана

Машина фон Неймана состоит из управляющего устройства, включающего управляющую программу и микропрограммы операций машины, памяти и регистров. Для обеспечения работы машины фон Неймана необходимо поместить в ее память следующие *объекты*: программу, таблицу имен и все данные, которые должна обработать программа. Каждый объект размещается в одной или нескольких ячейках памяти и, следовательно, связывается с определенными адресами памяти. Мы сконструировали память машины фон Неймана как хранилище произвольных слов без какой-либо интерпретации. Это означает, что ответственность за правильное использование слов лежит на программисте. Именно он должен позаботиться о том, чтобы слова с программой и с данными не были бы перепутаны, т. е. чтобы данные ошибочно не трактовались как программа и наоборот. Также необходимо следить за правильной интерпретацией слов данных. Слово, содержащее целую единицу, при непосредственном выводе не даст изображения единицы (знак '1').

В соответствии с одним из принципов фон Неймана программа и данные помещаются в одном и том же устройстве памяти. И только на регистрах процессора слова с командами программы и данными существенно различаются. Этот принцип позволил упростить конструкцию первых ЭВМ и был назван принстонской архитектурой по названию университета, где работал фон Нейман. Кроме простоты реализации принстонская архитектура позволила немедленно выполнять только что сгенерированную в памяти программу, которая еще мгновение назад была выходными данными другой программы, облегчив тем самым построение операционных систем, систем программирования и отладчиков. Недостатком принстонской архитектуры является ее небезопасность, поскольку этими же возможностями могут воспользоваться вирусы, трояны и прочие программные паразиты. Альтернативная гарвардская архитектура основана на раздельной реализации памяти

для программ и памяти для данных, что повышает безопасность ценой дополнительных аппаратурных затрат.

Таблица имен, программа и данные могут быть записаны в виде, непосредственно переносимом в память машины. При этом все заботы об установлении соответствия между именами и адресами, размещении данных в памяти, определении начальных значений некоторых данных полностью ложатся на программиста.

Процесс составления таблицы имен и размещения данных в памяти машины может быть автоматизирован. Необходимая для этого информация обычно содержится в тексте программы. Для полной автоматизации процесса решения текст программы должен быть дополнен информацией о том, каким процессором должны быть обработаны эти данные, так как операции машины фон Неймана определены над данными, допустимыми соответствующими процессорами. Следовательно, возможна система программирования, в которой составление таблицы имен и размещение данных в памяти машины выполняются автоматически некоторой специальной программой. Для упрощения процесса составления таблицы имен имеющуюся в каждой программе обработки данных информацию о свойствах объектов будем *в явном виде* помещать *в начало программы*. Эта информация, в частности, позволяет назначить процессор для обработки поименованного объекта программы.

Каждый объект, используемый для получения значений других объектов при выполнении программы обработки данных, должен иметь *определенное* значение, отличное от  $\perp$ . Объекты программы, как правило, получают свои значения в результате вычислений над другими объектами, ранее заданными в тексте программы. Из этого следует, что могут существовать объекты, которые должны получить значения *до начала* выполнения программы. Начальные значения таких объектов могут быть заданы либо при размещении их в памяти машины (*инициализация* самоопределеными константами (в Паскале и Си) и значениями по умолчанию (в Си)), либо путем выполнения инструкций ввода данных. Кроме того, некоторые компиляторы автоматически инициализируют память, выделяемую переменным, нулями, пробелами или *nil'ами*.

Инструкции ввода данных могут, в принципе, содержаться в любом месте текста программы. Однако важно, чтобы определение начальных значений объектов логически предшествовало их использованию для получения значений других объектов. В противном случае все вычисления, последовавшие за употреблением неопределенного значения, будут некорректными и, следовательно, не должны выполняться.

Рассмотрим средства описания объектов программы и определения начальных значений некоторых из них. Языки программирования Паскаль и Си требуют обязательного описания всех объектов, кроме предопределенных.

`var i, j, k : integer;`

`int i, j, k;`

Языки Бейсик и Фортран с самого начала были ориентированы на употребление неописанных скалярных объектов; процессор для обработки таких объектов выбирался компилятором исходя из контекста или по умолчанию (правило первой буквы).

Одним из способов задания начальных значений объектов является их инициализация непосредственно в тексте программы самоопределенными термами — словами, изображающими конкретные значения, допустимые соответствующими процессорами.

```
const i = 0;  
j = 0;  
k = 0;
```

```
const int i = 0, j = 0, k = 0;
```

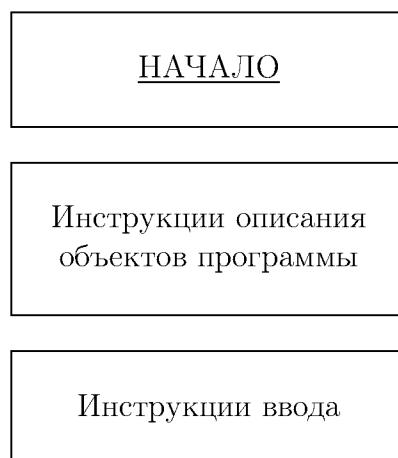
```
var i, j, k : integer;  
i := 0;  
j := 0;  
k := 0;
```

```
int i = 0, j = 0, k = 0;
```

или

Вспомним также, что данные в памяти машины фон Неймана непосредственно недоступны восприятию человеком (такое представление будем для краткости называть *внутренним*). Для непосредственного использования данных необходимо преобразовать их во *внешнее представление*. Процесс преобразования внутреннего представления информации во внешнее представление и реализацию этого внешнего представления на доступном для человека носителе (например, печать на листе бумаги или изображение на экране монитора) будем называть *выводом данных*. В результате вывода данные преобразуются в форму текстовых, графических, звуковых или иных сообщений, непосредственно доступных органам чувств человека. Для указания необходимого вывода данных в языке описания программ имеются специальные инструкции (инструкции вывода). Инструкции вывода данных могут, в принципе, так же как и инструкции ввода данных, содержаться в любом месте текста программы. Однако важно, чтобы получение значений объектов (как результат обработки данных) логически предшествовало их выводу. Инструкции вывода также будут рассмотрены позже.

Таким образом, программа для машины фон Неймана представляет собой *текст*, обязательно включающий в себя инструкции описания объектов (данных, которые обрабатывает программа), инструкции ввода данных (они могут отсутствовать, если необходимые для вычислений начальные значения объекты получают при размещении их в памяти машины), инструкции обработки данных и инструкции вывода значений объектов-результатов. В данном разделе мы рассмотрим простые программы, которые в своем большинстве имеют структуру, представленную на следующей схеме.





Символы НАЧАЛО и КОНЕЦ указывают начало и конец **текста** программы и границы действия описаний и объектов.

### 3.1.2 Обобщенная инструкция присваивания

Согласно одному из принципов фон Неймана вычислительная машина должна осуществлять покомандную обработку данных. Операнды каждой команды вызываются на регистры из памяти, после чего эта команда выполняет необходимое действие над ними, оставляя результат также на регистрах. Для сохранения результата он должен быть скопирован в память, поскольку регистры требуются для следующей команды. Поэтому программная версия машины фон Неймана должна содержать аналог цепочки «загрузка операндов — выполнение команды — сохранение результата». В любом языке программирования фон Неймановского типа основной элементарной инструкцией обработки данных является *инструкция присваивания*, которая и является таким аналогом. Инструкция присваивания имеет вид

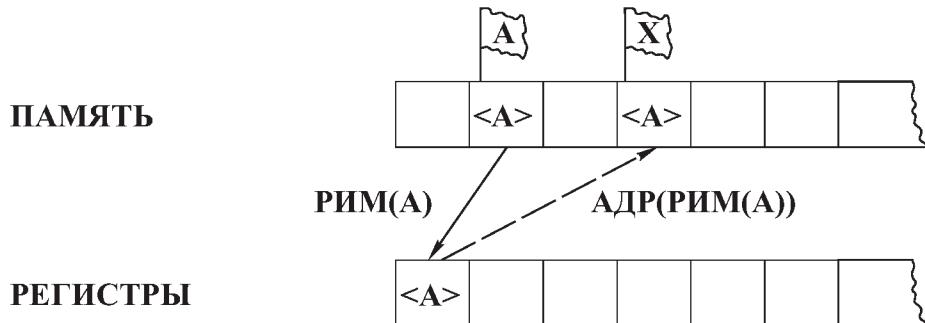
$X := A;$

где символ  $:$ = является знаком присваивания;  $X$  обозначает имя объекта (простой, индексированной или квалифицированной переменной, либо, иногда, массива),  $A$  — выражение, задающее последовательность действий над операндами. В языке Си все, что может стоять в левой части присваивания, называется *lvalue*. В простейшем случае, когда  $X$  и  $A$  — простые переменные или константы, действие инструкции присваивания сводится к перезагрузке значения переменной  $A$  в переменную  $X$  через регистр. В явном виде это выражается так

$X := \text{АДР}(\text{РИМ}(A))$

что имеет следующий *императивный* смысл: сделать адрес объекта с именем  $X$  адресом значения объекта  $A$ .

Графически выполнение инструкции присваивания в этом случае можно изобразить так (здесь и далее на схемах запись  $X$  означает имя объекта, а запись  $\langle X \rangle$  — значение объекта с именем  $X$ ):



Значение  $\langle A \rangle$  переменной с именем А из памяти машины сначала переносится (копируется) на свободный регистр, а затем с регистра копируется в ячейку памяти, сопоставленную с именем X. Переменная A должна иметь определенное значение; применение функции РИМ к неопределенному значению  $\perp$  должно приводить к отказу машины (исключительной ситуации выполнения программы). На практике одним из либерализмов фон Неймана стало отсутствие проверки на неопределенность как на аппаратном, так и на программном уровнях. Это было сделано для обеспечения быстроты исполнения программы, ответственность за корректность использования данных перекладывалась на программиста. Действительно, аппаратная поддержка даже одного дополнительного бита определенности значения многобайтового машинного слова усложняла бы оборудование и замедляла бы выполнение программ не на несколько процентов, как могло бы казаться, а в несколько раз. Чисто программная поддержка проверки неопределенности значений используется некоторыми компиляторами для предварительной инициализации неопределенными начальными значениями, которые вылавливаются в процессе выполнения программы специально расставленными контрольными инструкциями. В результате выполнение программы существенно замедляется и становится фактически интерпретативным. Объем программы также существенно возрастает. Кроме того, под неопределенное значение приходится занимать одну из допустимых кодовых комбинаций слова, что тоже не есть хорошо. В некоторых случаях цена такого контроля является оправданной. В частности, Паскаль-компиляторы обычно имеют ключи типа -C, задающие такого рода проверки.

Указанный порядок выполнения инструкции присваивания определяется требованием нормированного выполнения вычислений.

Рассмотрим более общий случай инструкции присваивания, когда А — выражение. Если это выражение содержит одну операцию машины фон Неймана с необходимыми операндами, то инструкция присваивания в точности повторяет вышеописанный порядок выполнения одиночной машинной операции. В программных версиях машины фон Неймана обычно позволяет строить более сложные, аналогичные математическим, выражения. Эти выражения содержат целые последовательности операций и, как правило, снабжены скобочными структурами. Пусть, например, выражение в правой части имеет вид  $B * C + D$  и X — имя объекта, получающего значение результата вычисления выражения. Действия машины фон Неймана при выполнении этой инструкции присваивания могут быть описаны так:

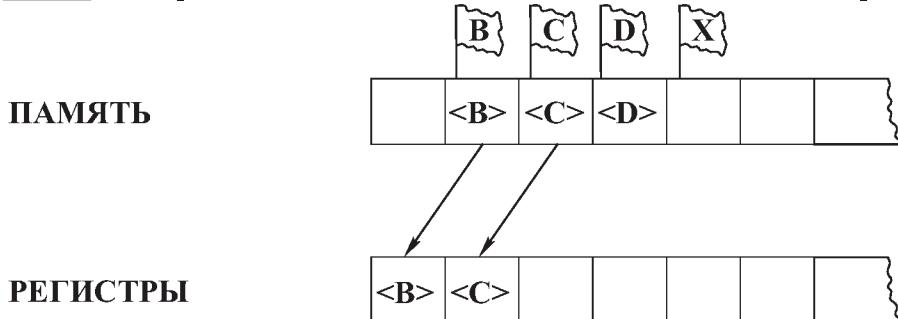
$$X := \text{АДР}(\text{РИМ}(B) * \text{РИМ}(C) + \text{РИМ}(D))$$

т. е. перемножаются не имена, а значения B и C.

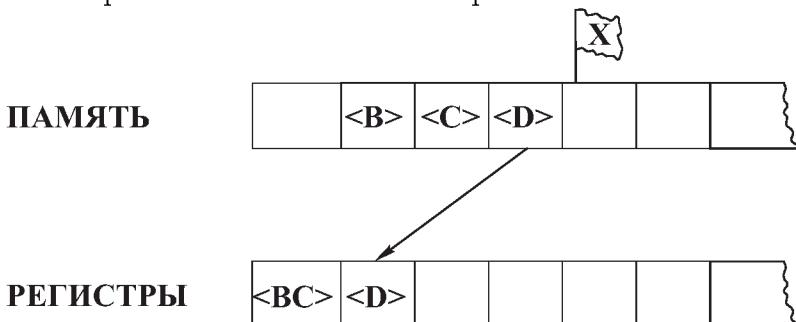
Подчеркнем императивный смысл инструкции присваивания, побуждающей к вычислению выражение в правой части.

Графически выполнение этой инструкции присваивания можно проиллюстрировать следующим образом.

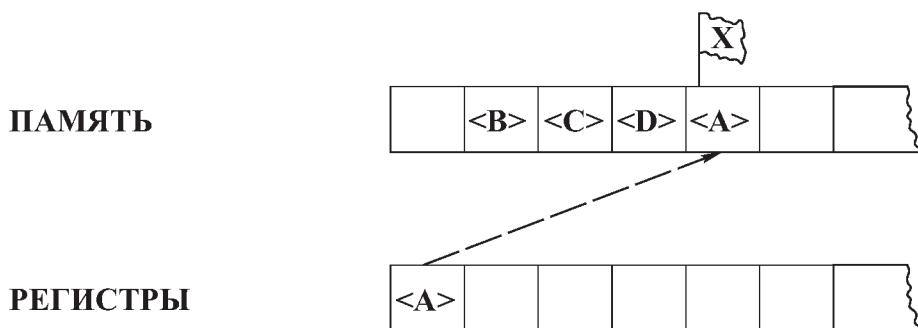
1 этап: копирование значений объектов B и C на свободные регистры.



2 этап: в результате нормированного выполнения операции умножения на одном из регистров остается полученное произведение, значение которого на схеме обозначим через <BC>; другой регистр становится опять свободным. После этого осуществляется копирование значения объекта на свободный регистр. Таким образом вычисление сложных выражений программной машины фон Неймана производится по частям, аналогичным простейшим командам аппаратуры, а само выражение должно быть предварительно декомпозировано на такие элементарные компоненты.



3 этап: в результате нормированного выполнения операции сложения на одном из регистров остается сумма, значение которой обозначим через <A>. Полученное значение <A> с регистра копируется в ячейку памяти, сопоставленную с именем X. Если выражение содержит скобки, то оно помимо декомпозиции на элементарные операции должно быть переведено в линейную бесскобочную форму, обеспечивающую ее последовательное выполнение. Скобочная запись выражения нелинейна и не может быть выполнена однократным линейным просмотром. Для перевода в бесскобочную запись потребуется еще одна рабочая лента ( $M$ -лента, стек) для хранения промежуточных неименованных результатов вычисления выражения. Альтернативой этому способу является рекурсивная интерпретация выражения, которая, как правило, не имеет прямой аппаратной поддержки и, в конечном счете, все равно сводится к использованию дополнительной рабочей ленты для стека. Перевод выражения в бесскобочную запись обычно выполняется языковым процессором (компилятором или интерпретатором).



Наши приемы, раскрывающие суть инструкции присваивания, могут оказаться полезными для понимания индуктивного присваивания  $i := i + 1$ , бессмысленного как статическое соотношение  $i = i + 1$ . Действительно, разыменование операндов правой части приведет к загрузке на регистры значения  $i$  и константы 1. Далее будет выполнено сложение, результат которого заместит на регистровой ленте операнды. После чего, согласно семантике оператора присваивания, этот результат будет поименован указанным в левой части именем  $i$ . Прежнее значение  $i$  будет замещено вновь пришедшем (разрушающей записью).

Все операнды выражения и результат должны изображаться допустимыми словами одного процессора. Если же в одном выражении присутствуют разнотипные операнды, обрабатываемые разными процессорами, то их типы должны быть согласованными, и во внутреннее представление выражения вставляются необходимые согласующие действия, включая возможные согласования левой и правой части инструкции присваивания.

Выполнение инструкции присваивания в этом случае производится таким образом: вычисляется значение выражения в правой части инструкции; к полученному значению выражения (оно находится на регистре) применяется указанная функция согласования; производится копирование преобразованного значения (оно — на соответствующем регистре!) с регистра в память машины.

*Обобщением инструкции присваивания является инструкция, задающая одновременную замену значений нескольких объектов.* Левая часть такой инструкции представляет собой список именованных объектов. Правая часть инструкции должна содержать столько же выражений, сколько имен объектов в списке левой части инструкции, или одно выражение. В первом случае инструкция одновременного присваивания имеет вид

$$X_1, X_2, \dots, X_N := A_1, A_2, \dots, A_N;$$

При выполнении этой инструкции  $i$ -я переменная ( $i = 1, 2, \dots, N$ ) ее левой части получает значение  $i$ -го выражения ее правой части. В случае, когда в правой части инструкции только одно выражение, все объекты в левой части получают одно и то же значение — значение этого выражения. Например, инструкция

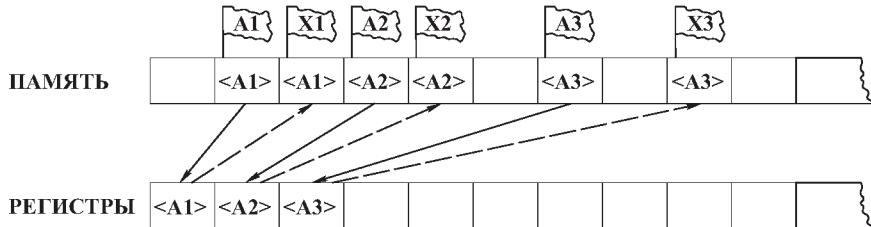
$$X_1, X_2, X_3, X_4 := 0;$$

обеспечивает получение каждой из четырех переменных списка левой части значения 0.

Выполняется инструкция одновременного присваивания аналогично обычной инструкции присваивания. Первоначально на свободные регистры копируются (если выражения справа — простые переменные или константы) или (в общем случае) с использованием регистров вычисляются все выражения правой части инструкции присваивания, а затем результаты с регистров копируются в соответствующие ячейки памяти. Случай, когда

в обеих частях инструкции приведен список простых переменных или именованных констант, можно проиллюстрировать следующим образом. Для иллюстрации возьмем инструкцию

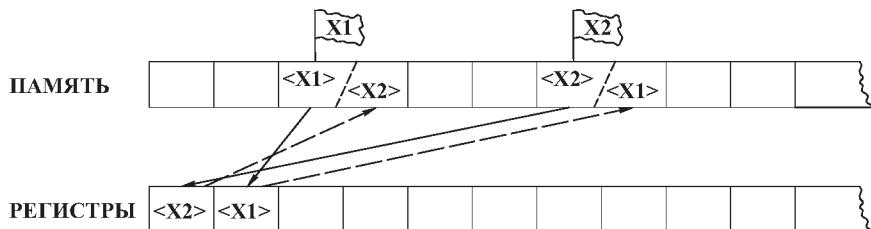
$X1, X2, X3 := A1, A2, A3;$



Следует обратить внимание на то, что все инструкции, обеспечивающие выполнение инструкции одновременного присваивания, а именно  $X := \text{АДР(РИМ}(A))$ , исполняются **одновременно** (практическая реализация должна обеспечить эффект их одновременного независимого выполнения). Это позволяет лаконично записать обмен значениями двух объектов; кстати, такой обмен значениями реализован в небезызвестном языке *Python*! Более того, вышеупомянутая инструкция присваивания списка значений списку переменных тоже реализована:

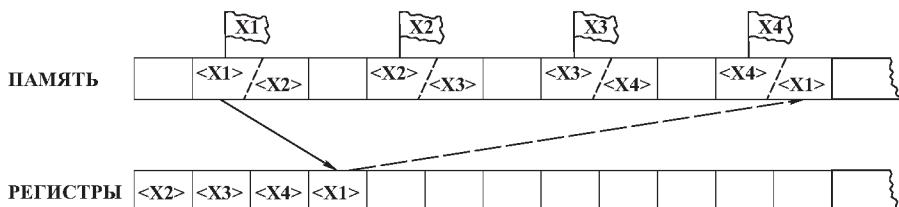
$X1, X2 = X2, X1$

$X1, X2, X3 = A1, A2, A3$



И в дейкстровской нотации, и в Python допустима циклическая перестановка значений объектов:

$X1, X2, X3, X4 = X2, X3, X4, X1$



Обычными операторами свопинг реализуется гораздо длиннее и требует вспомогательной переменной:

```
t := X1;
X1 := X2;
X2 := t;
```

Это чисто последовательное «однорукое» решение (для обмена значениями достаточно одной руки!) В C++ имеется шаблон встраиваемой (inline) библиотечной (STL) функции для свопинга:

```

template<typename T>
inline void std::swap<T>(T& lhs, T& rhs)
{
    T t = lhs;
    lhs = rhs;
    rhs = t;
}

```

Инструкция присваивания применима и в том случае, когда в левой части стоит имя массива. Тогда в правой части инструкции присваивания должно находиться имя массива той же структуры или скалярная константа. В последнем случае все элементы массива получат значение этой константы. На современных скалярных ЭВМ присваивание массивов, конечно же, реализуется программным путем, поэлементной пересылкой компонент. Отсутствие аппаратной поддержки присваивания массивов является одной из причин отсутствия этой операции в языке Си. С другой стороны, операция присваивания и отношения равенства должны быть определены над любыми данными, кроме бессмысленных вещей вроде `input := output`. Это наполовину реализовано в Паскале: присваивание и сравнение для массивов допустимы, а для файлов — нет.

Символы присваивания в Паскале и Си различаются: в Паскале он составной (`:=`), а в Фортране, Си, Бейсице, Яве и Python — атомарный (`=`). Во всем остальном больше сходств, чем различий.

`x := b * c + d;`

`x = b * c + d;`

Нельзя не упомянуть условного присваивания, имеющего место в Алголе и Си (но не в Паскале!):

`x := if b > c then b else c;`

`x = b > c ? b : c;`

Условное присваивание также реализовано в Python’е, оно более читабельно, чем в Си:  
`x = a if a > b else b.`

## Лекция 17

### 3.1.3 Обобщенная инструкция композиции

Общепринятый и широко используемый способ организации вычислений громоздких выражений — *функциональная композиция*, состоящая в том, что значения одной или нескольких функций используются в качестве аргументов для других функций.

Поясним это на **примере**.

Пусть требуется вычислить корни квадратного уравнения  $ax^2+bx+c=0$ . Формулы для вычисления корней квадратного уравнения  $x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$  удобно для вычислений представить в виде  $x_{1,2} = \frac{-b \pm d}{e}$ , где  $d = \sqrt{b^2 - 4ac}$  и  $e = 2a$ . (Будем пока считать, что проблемы с вычислением квадратного корня из дискриминанта не существует.) Таким образом, вместо исходного соотношения для вычисления корней квадратного уравнения

используется другое соотношение, аргументами которого являются значения двух выделенных выражений.

Заметим также, что запись  $x_{1,2} = \frac{-b \pm d}{e}$  представляет в действительности два соотношения. По этой причине и для рациональной организации вычислений (например, для минимизации числа операций) целесообразно иметь возможность разбивать процесс вычисления выражения на ряд последовательных процессов вычисления подвыражений, полученные значения которых можно было бы использовать в качестве аргументов при последующих вычислениях.

Составную инструкцию, состоящую из последовательности инструкций  $S_i$ , разделенных знаком «точка с запятой», будем называть композицией инструкций или коротко — *композицией*:

$$S_1; S_2; S_3; \dots; S_N$$

В композиции каждая инструкция  $S_{k+1}$  выполняется после нормального завершения выполнения инструкции  $S_k$  ( $k = 1, 2, \dots, N - 1$ ) и может через соответствующие имена использовать для вычислений значения, полученные в результате выполнения предшествующих инструкций. Таким образом, композиция носит сугубо последовательный характер.

Использование композиции позволяет написать следующую программу вычисления корней квадратного уравнения:

*BEGIN*

Инструкции описания объектов программы: коэффициентов уравнения <b>a</b> , <b>b</b> , и <b>c</b> , корней уравнения <b>x1</b> и <b>x2</b> , дискриминанта <b>d</b> и переменной <b>e</b> для хранения значения удвоенного первого коэффициента
Инструкции ввода начальных значений коэффициентов уравнения: <b>a</b> , <b>b</b> и <b>c</b>
<pre> <b>e</b> := 2.0 * <b>a</b>; <b>d</b> := SQRT(<b>b</b> * <b>b</b> - 4.0 * <b>a</b> * <b>c</b>); <b>x1</b> := (-<b>b</b> + <b>d</b>) / <b>e</b>; <b>x2</b> := (-<b>b</b> - <b>d</b>) / <b>e</b>; </pre>
Инструкции вывода: <b>a</b> , <b>b</b> , <b>c</b> , <b>x1</b> , <b>x2</b>

*END*

В языке Паскаль последовательная композиция инструкций скрепляется т. н. операторными скобками **begin** и **end** в Паскале и символами **{** и **}** в Си. Полученная таким образом составная инструкция может быть использована всюду, где дозволяется простая инструкция.

В настоящее время, когда параллельное выполнение, многопроцессорность и много задачность реальны и доступны, нельзя не упомянуть и о параллельной композиции. Композируемые элементы в этом случае выполняются параллельно и независимо, что должно отражаться в форме записи: вместо точки с запятой для разделения параллельно выполняемых ветвей могут использоваться знаки **||** или **[]**.

### 3.1.4 Охраняемые инструкции

Во многих задачах не существует единого списка инструкций, выполняющих требуемые вычисления во всех случаях. Рассмотрим простой **пример**. Пусть требуется вычислить значение функции

$$f = \begin{cases} x & \text{при } x \leq 0; \\ x + x^2 & \text{при } x > 0 \end{cases}$$

Очевидно, что выражение, которое необходимо использовать для вычисления значения функции, зависит от значения аргумента. Было бы желательно организовать вычисления следующим образом:

- присвоить переменной  $f$  значение переменной  $x$ ;
- если  $x > 0$ , то прибавить  $x^2$  к  $f$ ;
- если же  $x \leq 0$ , то инструкцию, по которой осуществляется прибавление  $x^2$ , пропустить.

Таким образом, надо иметь возможность разрешать или запрещать выполнение той или иной инструкции в зависимости от выполнения определенных условий. Инструкцию, выполнение которой зависит от текущего состояния вычислительной системы, будем называть *охраняемой*. Охрана инструкции осуществляется *предохранителем*. Общий вид предохранителя

<логическое выражение>?

Логическое выражение принимает одно из двух возможных значений — **И** (*Истина*) или **Л** (*Ложь*). Если логическое выражение принимает значение **И** (*истина*), то инструкция, непосредственно записанная за символом ?, выполняется. В противном случае инструкция, следующая за символом ?, не выполняется (пропускается).

В простейшем случае предохранитель используется для охраны инструкции присваивания, входящей в композицию, в следующей форме:

<логическое выражение> ? <инструкция присваивания>

Если логическое выражение принимает значение **И**, то выполняется инструкция присваивания, следующая за предохранителем. В противном случае эта инструкция пропускается.

С использованием охраняемой инструкции программа для вычисления значения функции может быть записана следующим образом:

```
BEGIN
.....
.....
f := X;
X > 0 ? f := f + X * X;
.....
END
```

Предохранитель используется в двух других составных инструкциях: ветвлении и цикле.

### 3.1.5 Обобщенная инструкция ветвления

Инструкцию ветвления (**IF–FI**) определим следующим образом:

```
IF <предохранитель1> <охраняемая инструкция1>
  ┌ <предохранитель2> <охраняемая инструкция2>
  . . .
  ┌ <предохранительm> <охраняемая инструкцияm>
FI
```

Символы **IF** и **FI** играют роль открывающей и закрывающей скобок инструкции ветвления, символ **└** играет роль разделителя охраняемых инструкций, входящих в состав ветвления.

По инструкции ветвления из списка охраняемых инструкций, заключенных в скобки **IF–FI**, должна быть выбрана одна и только одна инструкция (для выполнения), и причем та, предохранитель которой принимает значение **И** (истина).

С использованием инструкции ветвления программа для вычисления значения функции (см. п. 3.3.2) может быть записана следующим образом:

```
BEGIN
  . . .
  IF X ≥ 0 ? f := X;
  ┌ X < 0 ? f := X + X * X;
  FI;
  . . .
END
```

Правила выполнения инструкции ветвления следующие:

- Одновременно и независимо вычисляются все предохранители.
- Среди вычисленных предохранителей инструкции ветвления должен быть хотя бы один, принимающий значение **И**. Если среди предохранителей инструкции ветвления нет ни одного, принимающего значение **И**, то происходит ОТКАЗ — выполнение программы прекращается (аварийно!). Таким образом, отсутствие среди предохранителей инструкции **IF–FI** предохранителя, принимающего в момент выполнения инструкции значение **И**, воспринимается как грубая ошибка периода выполнения, препятствующая дальнейшему хоть сколько-нибудь осмысленному выполнению программы. Дело в том, что при планировании ветвления программист должен позаботиться о «полноте» системы предикатов в охраняемых инструкциях, и предусмотреть дополнительную ветвь, отрицающую все предикаты.
- Допускается, чтобы среди предохранителей инструкции ветвления было более одного предохранителя, принимающего значение **И** при выполнении инструкции **IF–FI**.

При этом не предполагается, что охраняемые инструкции каким-либо образом упорядочены. Следовательно, если среди предохранителей инструкции **IF–FI** окажется, например, два предохранителя, принимающих значение **И** (истина), то выбор инструкции для выполнения осуществляется недетерминированным образом и не определяется порядком записи

охраняемых инструкций в инструкции **IF–FI**. Недетерминированное ветвление, вообще говоря, неалгоритично и отражает тот факт, что программисту все равно, по какой из открывшихся ветвей действительно произошло ветвление. Недетерминированность не следует считать случайностью, т. к. случайные величины подчиняются вероятностным законам, что противоречит духу недетерминированности. Недетерминированное ветвление отражает стремление к повышению семантического уровня языка программирования до пропозициональной семантики.

Все известные конструкции ветвления в языках программирования являются частными случаями обобщенного ветвления Э. Дейкстры.

Например, двузвенное ветвление

$$\text{IF } p ? s1 \quad \square \quad \text{not } p ? s2 \text{FI}$$

в Паскале и Си реализуются практически одинаково:

**if** p **then** s1 **else** s2;

**if**(p) s1(); **else** s2();

а многозвездный переключатель (избыточный по теореме Бойма-Джакопини-Миллса):

$$\begin{array}{c} \text{IF } e = e_1 ? s_1 \\ \quad \square e = e_2 ? s_2 \\ \quad \quad \vdots \\ \quad \quad \square e = e_n ? s_n \\ \text{FI} \end{array}$$

записывается на этих языках несколько по-разному:

**case** e **of**  
  e1 : s1;  
  e2 : s2;  
  {...}  
  en : sn;  
**end;**

**switch**(e)  
{  
  e1: s1(); **break**;  
  e2: s2(); **break**;  
  // ...  
  en: sn();  
}

Естественно, здесь все детерминировано. Пользуясь конкретными вариантами инструкции ветвления, необходимо внимательно изучать их особенности и не только по описанию стандарта языка. Например, в Паскале в случае несовпадения текущего варианта ни с одной из предусмотренных меток вместо благородного и шумного отказа от неудачного ветвления втихую происходит продолжение работы программы. Исправление такой ситуации веткой, выполняемой по умолчанию (*otherwise* или *else*), увы, не стандартизовано.

В С/C++ есть стандартная опциональная метка **default**, которая получает управление в случае, когда не сработал ни один из **case**'ов.

### 3.1.6 Обобщенная инструкция цикла

Пусть требуется вычислить значение  $f(n) = n!$  для заданного значения  $n$ . Поставим целью составить программу вычисления  $n!$  для любого задаваемого значения  $n$ . Очевидно, что

это можно сделать по следующим соотношениям:

$$\begin{aligned} \text{при } n = 0 \quad f(n) &= 1; \\ \text{при } i > 0 \quad (i = 1, 2, \dots, n) \quad f(i) &= f(i - 1) * i, \end{aligned}$$

дающим правила вычисления каждого последующего значения функции через предыдущее. Следовательно, программа должна содержать: определение начального значения функции  $f(0)$  и завершение вычисления, если  $n = 0$ ; последовательное умножение значения функции на  $i = 1, 2, \dots, n$  до тех пор, пока  $i \leq n$ , где  $n$  — задаваемое значение.

Из сказанного следует, что есть необходимость в специальном программном средстве для организации многократного выполнения умножения текущего значения искомой функции на значение аргумента, увеличившееся на единицу. Повторяться эти вычисления должны до тех пор, пока значение аргумента не превысит некоторого заранее заданного значения.

Такой инструкцией является инструкция *цикла*, которую мы определим для общего случая следующим образом:

```
DO <предохранитель1> <охраняемая инструкция1>
  ┌ <предохранитель2> <охраняемая инструкция2>
  . .
  ┌ <предохранительm> <охраняемая инструкцияm>
OD
```

Что русскому хорошо, то немцу смерть

Русская пословица

Несмотря на то, что инструкция цикла **DO-OD** внешне похожа на инструкцию ветвления, правила выполнения этой инструкции определяются иначе:

- Если среди предохранителей инструкции **DO-OD** *один и только один* принимает значение **И**, то соответствующая охраняемая инструкция выполняется, и после этого вновь осуществляется *одновременная и независимая проверка всех предохранителей* инструкций цикла. Следует обратить внимание на то, что охраняемой инструкцией может быть композиция или ветвление.
- Количество предохранителей, принимающих значение **И**, также *может быть более одного*. В этом случае не предполагается, что охраняемые инструкции упорядочены и выбор инструкции для выполнения очередного повторения цикла осуществляется вне связи с порядком их написания в инструкции **DO-OD**, т. е. недетерминированным образом.
- Если среди предохранителей инструкции **DO-OD** нет *ни одного*, принимающего значение **И**, то, *в отличие от ветвления*, выполнение инструкции заканчивается естественным образом в связи с отсутствием инструкции, открытой для продолжения работы цикла.
- Выполнение каждой охраняемой инструкции должно приводить к изменению аргументов предохранителей. В противном случае выполнение инструкции **DO-OD** может никогда не закончиться.

С использованием инструкции **DO-OD** программа для вычисления может быть записана следующим образом:

```
begin
    . . .
    f, i := 1, 2;
    do i ≤ n ? f := f * i;
        i := i + 1
    od
    . . .
end
```

Для сравнения рассмотрим другой вариант программы с использованием инструкций **IF-FI** и **DO-OD**.

```
BEGIN
    . . .
    f, i := 1, 2;

    IF n = 0 ? ПРОПУСК
    IF n > 0 ? DO i ≤ n? f := f * i;
                    i := i + 1
                OD
            FI;
    . . .
END
```

Второй вариант программы выглядит более громоздким, однако обладает важным достоинством по сравнению с первым вариантом. Дело в том, что в инструкции **IF-FI** отсутствует предохранитель, принимающий значение **И** при отрицательном значении  $n$ . Поэтому попытка использовать значение  $N$  из недопустимой области значений будет «наказана» ОТКАЗом в выполнении инструкции **IF-FI**, и тем самым программа будет защищена от неправильного использования.

<pre>i := 1 <b>DO</b> i ≤ n ? s; i := i + 1 <b>OD</b></pre> <pre><b>for</b> i := 1 <b>to</b> n <b>do</b>     s;</pre> <pre><b>DO</b> p ? s <b>OD</b></pre> <pre><b>while</b> p <b>do</b>     s;</pre>	<pre>for(int i = 0; i &lt; n; i++)     s();</pre> <pre>while(p)     s();</pre>
--	---

s DO p ? s OD

или так

flag := true DO flag ? s OD

repeat  
  s  
until not p;

do  
  s();  
while(p);

## Лекция 18

### 3.2 Типы данных

#### 3.2.1 Определение типа данных

**Тип данных** — это *множество изображений* (слов над некоторым алфавитом), для которых определено правило их интерпретации, позволяющее каждому изображению сопоставить его значение, и *множество атрибутов*, которые позволяют одному или нескольким элементам типа данных сопоставить либо изображения данных того же типа, либо изображения данных другого типа. В частности, к атрибутам типа данных относятся минимальное и максимальное значения, количество значений типа, операции, отношения и функции, определенные над значениями этого типа. Т. е. определение типа имеет теоретико-множественный и алгебраический аспекты.

Интерпретация задается с помощью отображения:

$$\varphi : W \longrightarrow X$$

где  $W$  — множество изображений, а  $X$  — множество значений.

В отличие от человека машина не может непосредственно работать ни с абстрактными математическими объектами, такими как числа, множества, логические значения и литеры, ни с предметами, представляющими их материальные физические реализации (счётными палочками, оловянными литерами из типографской кассы, карточками, кубиками и прочими суррогатами из набора первоклассника). Машина всегда работает со словами небольшой и фиксированной длины, представляющими собой допустимые значения, которые по отношению к тем абстрактным объектам являются *изображениями*. Интерпретация этих внутримашинных изображений  $\varphi$  осуществляется неявно через алгоритмы выполнения операций и вычисления отношений, связанных с соответствующим типом данных. Например, операция  $+$ , являющаяся атрибутом целого типа, по двум заданным изображениям операндов получает слово, являющееся изображением целого числа, представляющего их сумму. Если же изображения объектов подлежат ручной обработке человеком, они могут быть записаны на бумаге и в таком осязаемом органами чувств виде непосредственно обработаны им.

Множество изображений, составляющих тип данных, может быть задано несколькими способами.

1. *Непосредственным перечислением изображений*. Когда их немного, удобно задать список слов, как, например, в Паскале и в Си для типа «Светофор»:

```
type TrafficLights = (red,
                      yellow, green);
```

```
enum TrafficLights {red, yellow,
                     green};
```

Перечислимые типы в конечном счете сводятся к начальному отрезку натурального ряда и, следовательно, имеют эффективную машинную реализацию. Поэтому в языках системного программирования нет перечислимого типа: без лишних условностей в этом качестве используется целый или адресный типы. Перечислимый тип не может быть первичным, поскольку он «паразитирует» на целом типе; непосредственная аппаратная реализация перечислимого типа физически непроста и нецелесообразна в связи с эффективной компиляцией на любую аппаратную платформу машины фон Неймана.

2. *Заданием характеристической функции  $\chi(w)$*  (предиката), определенной на некотором множестве изображений  $L \subset W$ , которая принимает значение *Истина*, если изображение относится к данному типу. Нередко такие функции позволяют весьма эффективно устанавливать принадлежность конкретного изображения множеству допустимых слов  $L$ . Например, легко проверить принадлежность значения отрезку (диапазону). Этот тип также не может быть первичным т. к. он базируется на некотором надмножестве  $W$ , которое должно иметь аппаратную или приравненную к ней низкоуровневую программную или микропрограммную поддержку.
3. С помощью системы аксиом, определяющих изображения через их свойства. Этот математически красивый способ позволяет легко задать потенциально бесконечное множество значений. Однако он неконструктивен, не позволяет в явном виде порождать значения типа или работать с ними. Создание аппаратуры или программного обеспечения, конструктивно реализующих удовлетворяющие аксиоматике атрибуты такого типа, дает возможность создания первичного типа, не основанного ни на каком другом. Тьюринговская модель вычислений в качестве такого элементарного базового аппарата поддерживаемого типа использует набор литер рабочего алфавита, над которым создаются надстройки, позволяющие работать со словами, числами и т. д.

Задание типа данных первыми двумя способами предполагает, что уже имеется множество изображений, из которых выделяется этот тип. Следовательно, в языке программирования обязательно должен быть хотя бы один тип, задаваемый способом 3. Отсутствие аппаратной реализации базового типа препятствует автоматическому выполнению программ, использующих как этот тип, так и все надстройки над ним.

Тип, который задается аксиоматически и имеет физическую реализацию, называется *базовым типом*. Атрибуты базовых типов машины фон Неймана реализуются управляющим устройством этой машины и процессорами типов данных.

Итак, в языке программирования должен быть определен по крайней мере один базовый тип. Даже при наличии только одного типа данных можно описывать любые алгоритмы. Однако на языке, в котором имеется только один тип данных, удобно описывать алгоритмы, связанные только с этим типом данных. Что касается остальных

алгоритмов, то при их описании на рассматриваемом языке мы фактически находимся в условиях, близких к МТ.

Например, в первой отечественной ЭВМ БЭСМ был реализован двоичный процессор целого типа. Поэтому при составлении программ, реализующих вычисления с вещественными числами на языке этой машины, возникали дополнительные и весьма серьезные трудности, связанные с необходимостью реализовывать процессор вещественного типа программным путем. Такие же проблемы возникали при реализации первых бортовых ЭВМ.

В языке программирования удобно иметь несколько базовых типов данных. Несмотря на избыточность такого решения с точки зрения абсолютной вычислимости, практическая (эффективная) вычислимость, как правило, осуществляется с помощью нескольких типов данных. Основными базовыми типами языков программирования являются логический (**BOOLEAN**), целый (**INTEGER**), вещественный (**REAL**) и литерный (**CHAR**). Они позволяют описать алгоритмы решения широкого класса задач автоматической обработки данных, поскольку имеют либо прямую аппаратную поддержку, либо эффективно компилируются.

Полный набор типов данных, при наличии которых в языке программирования существенно упрощается процесс написания программ, а сами программы становятся компактными, наглядными, понятными, а потому более надежными, зависит от поставленной задачи. Например, при разработке программы решения задачи, связанной с электрическими цепями, нужны комплексные числа. Конечно, можно написать программу на языке, где нет такого типа данных, и использовать вместо каждого комплексного числа массив из двух вещественных чисел, первое из которых представляет действительную часть, а второе — мнимую, и заменить все операции над комплексными числами операциями над их действительными и мнимыми частями. Однако при этом пропадут все преимущества использования комплексных чисел, и программы существенно усложняются. При наличии комплексного типа достаточно указать, какие переменные и константы (по именам) этого типа используются в программе, и знать, какие операции определены над данными этого типа. Наглядность использования комплексных чисел сохранится.

Рассмотрим примеры реализации комплексных чисел на Паскале, Си и С++.

Лучшей реализацией на Паскале и Си следует считать запись. Операции надо выполнять вручную, покомпонентно. В стандартной библиотеке языка С++ есть тип `complex`, реализующий, помимо представления чисел, также операции и отношения в стандартной (инфиксной) записи, приближенной к математической нотации. В последнем стандарте языка Си ISO/IEC 9899:1999 ([C99]) можно писать `complex z = 5 + 3*I;`

```
type complex = record
  Re, Im : real
end;

var c1, c2, c3, c4 : complex;
{ Покомпонентное сложение с
  присваиванием }
c3.Re := c1.Re + c2.Re;
c3.Im := c1.Im + c2.Im;
```

```
#include<complex> // C++
std :: complex c1, c2, c3, c4;
c3 = c1 + c2;
c4 = c3;
```

```
{ Обычное присваивание }
c4 := c3;
```

В некоторых языках программирования (Фортран, Бейсик) нет записей и структур. Поэтому комплексные числа приходится размещать в двухэлементных массивах. Неудобство также заключается в том, что в большинстве таких языков присваивание массивов покомпонентно. Здесь Паскаль — приятное исключение.

```
type complex = array[1..2] of real;
var c1, c2, c3, c4 : complex;
c3[1] = c1[1] + c2[1];
c3[2] = c1[2] + c2[2];
c4 := c3;
```

```
typedef float complex[2];
complex c1, c2, c3, c4;
c3[0] = c1[0] + c2[0];
c3[1] = c1[1] + c2[1];
c4[0] = c3[0];
c4[1] = c3[1];
```

Предыдущая запись плоха тем, что компоненты комплексного числа не поименованы, а пронумерованы. Если в нашем распоряжении имеется перечислимый тип, программа может иметь более ясный вид:

```
type part = (Re, Im);
type complex = array[part] of real;
var c1, c2, c3, c4 : complex;
c3[Re] := c1[Re] + c2[Re];
c3[Im] := c1[Im] + c2[Im];
c4 := c3;
```

```
enum part {Re = 0, Im = 1};
typedef float complex[2];
complex c1, c2, c3, c4;
c3[Re] = c1[Re] + c2[Re];
c3[Im] = c1[Im] + c2[Im];
c4[Re] = c3[Re];
c4[Im] = c3[Im];
```

Наконец, можно реализовать комплексное число парой вещественных скалярных переменных, представляющих вещественную и мнимую части. Но из-за раздельного хранения компонент комплексного числа все операции, отношения, включая присваивание, также придется выполнять вручную, что совсем далеко от математической формы записи.

```
var Re1, Re2, Re3, Re4 : real;
      Im1, Im2, Im3, Im4 : real;
Re3 := Re1 + Re2;
Im3 := Im1 + Im2;
Re4 := Re3;
Im4 := Im3;
```

```
float Re1, Re2, Re3, Re4;
float Im1, Im2, Im3, Im4;
Re3 = Re1 + Re2;
Im3 = Im1 + Im2;
Re4 = Re3;
Im4 = Im3;
```

Другой пример. Пусть имеются два массива из вещественных чисел: массив A и массив

В. И пусть требуется всем элементам массива A дать значения соответствующих элементов массива B, т. е. переслать массив B в массив A. Для описания этих действий, например, на языках Си, Бейсик и Фортран придется написать оператор цикла (типа FOR в Паскале), предусмотрев для его организации специальную переменную (*параметр цикла*) и (в Фортране и Бейсике) метку конца цикла:

```
var A, B : array[1..100] of integer;  
{ ... }  
A := B;
```

```
int i, A[100], B[100];  
// ...  
for(i = 0; i < 100; i++)  
    A[i] = B[i];
```

Наличие же в языке Паскаль типа массив дает возможность записать в таком случае один оператор присваивания A := B. И это не просто короткая запись. Когда имеется процессор типа массив (пусть и с ограниченным набором операций и отношений (только присваивание и только равенство)), он естественным образом будет проверять выход за границы массива, так как процессор «знает», из скольких элементов состоит каждый массив и не допустит, чтобы один массив непредусмотренным образом наложился на другой. Итак, наличие в языке нужного типа данных упрощает программы и исключает ошибки программирования.

Предусмотреть в множестве базовых типов универсального языка программирования все те типы данных, которые могут быть полезными при решении задач из разных предметных областей, невозможно. Поэтому в языке необходимо предусмотреть возможность введения новых типов и дать средства их конструирования на основе базовых типов данных. Используя эту возможность, программист может существенно улучшить и упростить свои программы.

Мы уже касались простейших способов введения новых типов, таких как перечислимые типы и типы диапазонов (отрезков).

Во многих задачах целые числа используется не в качестве арифметических операндов, а для нумерации элементов конечных множеств. Нумерация элементов дает удобство их упорядочивания и обработки, но затрудняет смысловую идентификацию нумерованных значений. В таких случаях в программе удобно иметь возможность привести список всех возможных значений. Например, нумерация состояний машины Тьюринга удобна для представления выполнимой Тьюринговской программы, но затрудняет составление алгоритма (см. пример первой Тьюринговской программы): используя мнемоничные имена состояний мы легко и безошибочно составили таблицу переходов. Итак, в основе *перечислимого типа* лежит идея явного задания слов — изображений значений, как правило, перенумерованных в порядке написания, и несущих некоторую семантическую нагрузку. При определении этого типа данных следует привести изображения всех констант, обозначающих все возможные значения этого типа, а также указать множества операций и отношений, применимых к данным этого типа. Как минимум, это множество содержит операцию присваивания и отношение равенства. В общем случае операции и отношения, а также алгоритмы их выполнения могут быть заданы процедурно. Но этого вопроса мы пока касаться не будем [43].

Поскольку элементы перечислимого типа упорядочены и пронумерованы, то соответствующее множество значений может быть задано отрезком или диапазоном. И наоборот, конечное упорядоченное множество подряд идущих целых чисел тоже перечислимо

и может быть задано отрезком  $i \dots j$ . Такой тип данных называется **отрезком типа** или диапазоном, чтобы подчеркнуть его происхождение от какого-нибудь надтипа (супертипа).

Если нумерация и порядок элементов не важны а интересно лишь их наличие или отсутствие, то может быть введен тип данных *множество*. Он также имеет простую реализацию (и частичную аппаратную поддержку) в виде битовых шкал, в которых каждый бит отражает наличие или отсутствие соответствующего элемента множества. Длина шкалы равна максимальной мощности множества. Ввиду жесткого соответствия бита и представляемого им элемента множества нумерация элементов неизбежна на уровне внутреннего представления. Таким образом, для задания конкретного типа множество программист должен указать тип его элементов, перечислимый, небольшой мощности. В современных языках программирования существуют гораздо более развитые средства поддержки множеств.

Все упомянутые выше типы данных: базовые (**BOOLEAN**, **INTEGER**, **REAL**, **CHAR**), тип перечисления и отрезок типа — являются *неструктурными* (скалярными) *типами* данных.

Однако на практике часто используются данные, имеющие так называемый *структурный тип*. Примером структуры может служить студенческая группа. Во многих случаях речь о группе ведут как о едином целом (при составлении расписания практических занятий и экзаменов, при подведении итогов сессии и т. д.). Однако группа, естественно, состоит из ее членов — студентов, при этом каждому студенту сопоставлен его номер по списку, которым пользуется как сам студент (например, формируя имя пользователя в системе UNIX или выбирая вариант задания), так и система КАДРЫ для накопления различных сведений о каждом студенте (например, о текущей успеваемости по предметам, об успехах и достижениях в научной работе и т. д.)

Для представления студенческой группы в ЭВМ используется структурный тип данных, называемый **массивом**. Данные, имеющие структуру массива, являются совокупностью компонентов одного и того же скалярного типа, причем каждый конкретный компонент массива может быть явно обозначен и к каждому компоненту имеется доступ по одному или нескольким **индексам**, т. е. определена функция, которая по значениям соответствующего количества индексов определяет адрес элемента массива. При этом заранее не делается никаких предположений относительно расположения массива в памяти; оно определяется лишь *функцией поиска по индексам*. Тип каждого индекса, однозначно указывающий желаемый элемент массива, может быть произвольным, но в нем обязательно должно быть определено отношение порядка (*succ/pred*, а не  $<$  и  $>$ , что в частности не позволяет типу **real** быть индексным).

Рассмотрим подробнее четыре базовых типа данных.

### 3.2.2 Тип логический

Значениями логического типа являются *Истина*, *Ложь* и *Неопределенность* (изображаемые словами **И**, **Т**, **true**, либо **Л**, **F**, **false** и  $\perp$ ). Цифровые изображения истинностных значений (1 или 0) также могут использоваться, но пока применяться не будут, чтобы не смешивать их с числовыми омонимами. К тем логическим значениям, которые рассматривались в математической логике (**И** и **Л**), добавлено еще одно значение —  $\perp$ , присваивание которого любому объекту приводит к возникновению особой ситуации. Таким образом, значение  $\perp$  позволяет описать класс особых ситуаций, которые могут возникнуть

при вычислении отношений и логических выражений в случаях, когда результат не имеет никакой осмысленной интерпретации. Неопределенное значение применяется не только в теории, но и на практике: в СУБД, удовлетворяющих стандарту SQL-92 (Oracle, DB2, Postgres, MS SQL Server и др.),  $\perp$  обозначается словом **NULL**. Функция интерпретации, связывающая значения и изображения, в данном случае очевидна. Хотя внутримашинное представление значений может отличаться: например, в языке Си машинное слово трактуется как *Ложь*, если в нем все его биты нулевые, и как *Истина* в противном случае. С другой стороны, в некоторых реализациях Паскаля *Истина* изображается единицей только в младшем разряде.

Над значениями логического типа определены ([105])

- одноместная операция  $\neg$  (отрицание, NEG или «!»);
- двуместные операции  $\&$  (конъюнкция, AND,  $\wedge$ ) и  $\vee$  (дизъюнкция, OR или  $\mid$ );
- отношения  $\equiv$  (эквиваленция, тождественность) и  $\not\equiv$  (нетождественность, исключающее ИЛИ,  $\text{^}$ , XOR).

Так как результатом проверки отношений являются логические значения, то отношения тоже можно считать логическими операциями, не размыкающими логического типа (не выводящими за его пределы).

Результаты выполнения логических операций определяются с помощью следующих таблиц:

Конъюнкция				Дизъюнкция				Тождественность				Отрицание			
$\wedge$	И	Л	$\perp$	$\vee$	И	Л	$\perp$	$\equiv$	И	Л	$\perp$	$\neg$	И	Л	$\perp$
И	И	Л	$\perp$	И	И	И	$\perp$	И	И	Л	$\perp$	Л	Л	И	$\perp$
Л	Л	Л	$\perp$	Л	И	Л	$\perp$	Л	Л	И	$\perp$	Л	Л	И	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	Л	И	$\perp$	$\perp$

Операции логического типа обладают следующими свойствами.

1. Если хотя бы один operand имеет значение  $\perp$ , то и результат имеет значение  $\perp$ .
2. Если оба операнда дизъюнкции (конъюнкции) истинны (ложны), то истинен или ложен и результат, то есть  $X \& X = X$ ,  $X \vee X = X$ .
3. Если  $X$  истинно, а  $Y$  имеет произвольное логическое значение, отличное от  $\perp$ , то  $X \vee Y$  также является истинным:  $X \vee Y = X$ .
4. Если  $X$  ложно, а  $Y$  имеет произвольное логическое значение, отличное от  $\perp$ , то  $X \& Y$  также является ложным:  $X \& Y = X$ .
5. Дизъюнкция и конъюнкция коммутативны:  $X \vee Y = Y \vee X$ ,  $X \wedge Y = Y \wedge X$ .
6.  $X \vee \neg X$  всегда истинно, если только  $X \neq \perp$ .
7.  $X \& \neg X$  всегда ложно, если только  $X \neq \perp$ .
8.  $\neg\neg X = X$  ( $\neg\neg X$  можно заменить на  $X$ ).

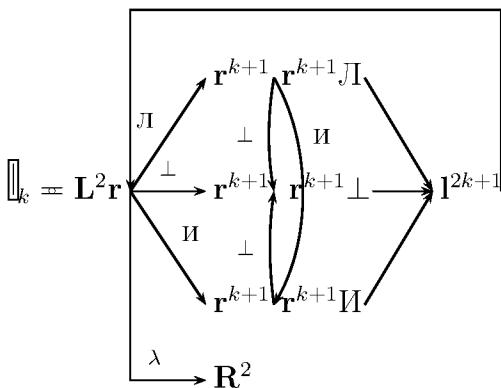
Названные выше операции и отношения могут быть распространены и на вектора длины  $k$  из логических значений следующим образом: конъюнкции (дизъюнкции, и т. д.) подвергаются значения соответствующих компонент векторов и результатом является новый вектор. Покомпонентные булевские операции подобны побитовым операциям над двоичными разрядами машинного слова. Такие поразрядные логические операции реализуются аппаратно над машинными словами фиксированной длины в регистрах.

Условимся считать, что на ленте МТ значение *Истина* изображается буквой **И**, значение *Ложь* — буквой **Л**, а все остальные буквы (кроме пустой ячейки) соответствуют значению *Неопределенность*. Таким образом мы определяем множество изображений  $W_k$  векторов из логических значений длины  $k$  в процессоре  $BOOL(k)$ .

Алгоритмы выполнения операций и вычисления отношений процессора  $BOOL(k)$  можно описать диаграммами следующих машин Тьюринга, которые являются предметом практических занятий и лабораторных работ по курсу [30]:

### Дизъюнкция

$$[\lambda w_1 \lambda w_2 (\lambda) \lambda] \xrightarrow{\parallel_k} [\lambda w_1 \lambda w_2 \lambda \bar{w} (\lambda) \lambda], \text{ где } \varphi(\bar{w}) = \varphi(w_1) \vee \varphi(w_2), \quad \bar{w}, w_1, w_2 \in W_k.$$

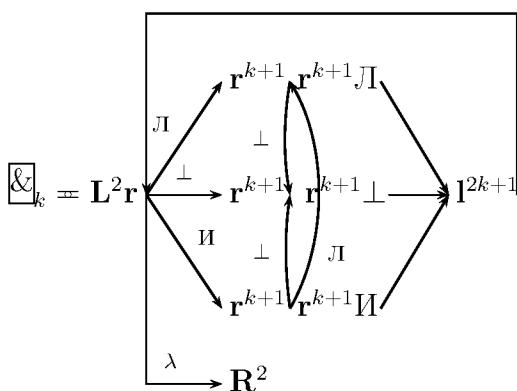


Эта диаграмма неструктурирована: дуга из ветки **Л** в ветку **И** пересекает ветку  $\perp$ , что можно оправдать ее эффективностью и компактностью.

При построении диаграммы МТ  $\parallel_k$  учитываются свойства 0 и 2 логических операций.

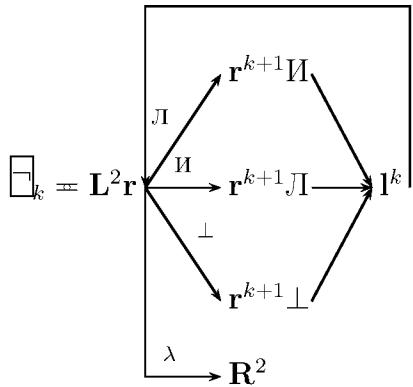
### Конъюнкция

$$[\lambda w_1 \lambda w_2 (\lambda) \lambda] \xrightarrow{\&_k} [\lambda w_1 \lambda w_2 \lambda \bar{w} (\lambda) \lambda], \text{ где } \varphi(\bar{w}) = \varphi(w_1) \& \varphi(w_2), \quad w, w_1, w_2 \in W_k.$$



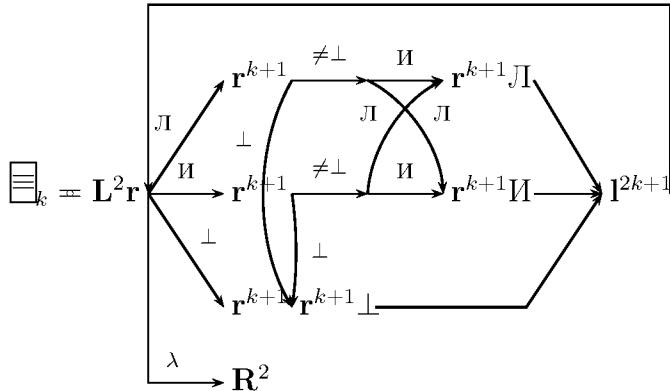
Отрицание

$$\boxed{\Box} [\lambda w(\lambda)\lambda] \xrightarrow{*} [\lambda w\lambda\bar{w}(\lambda)\lambda], \text{ где } \varphi(\bar{w}) = \neg\varphi(w), \quad \bar{w}, w \in W_k.$$



Тождественность

$$\boxed{\equiv} [\lambda w_1\lambda w_2(\lambda)\lambda] \xrightarrow{*} [\lambda w_1\lambda w_2\lambda\bar{w}(\lambda)\lambda], \text{ где } \varphi(\bar{w}) = \varphi(w_1) \equiv \varphi(w_2), \quad w, w_1, w_2 \in W_k.$$



*Задание:* структурировать эту диаграмму.

Таким образом, вычислимость по Тьюрингу логического типа доказана.

## Лекция 19

### 3.2.3 Тип целый

Из курса алгебры известно, что множество целых чисел состоит из бесконечного числа элементов вида  $0, \pm 1, \pm 2, \dots$ , удовлетворяющего системе аксиом и образующего алгебраическое кольцо относительно сложения и умножения.

Как уже было сказано выше, при аппаратной реализации процессора фон Неймана мы поступили абсолютной вычислимостью в пользу практически эффективной частичной вычислимости. Это, в частности, означало, что слова машины фон Неймана должны иметь ограниченную, небольшую и фиксированную длину, и само число слов в памяти машины также должно быть ограничено и невелико. Поэтому при реализации целого типа

в ЭВМ диапазон значений должен быть ограниченным отрезком числовой оси, включающим 0 и слегка несимметричным (см. п. 1.5). Несимметричность диапазона вызвана необходимостью однозначного представления нуля, который отнесен к положительным числам, что и приводит к сокращению диапазона положительных чисел на единицу. Поэтому наибольшее положительное число на единицу меньше модуля наименьшего отрицательного, а их коды диаметрально противоположны.

Значениями машинного целого типа являются все математические целые числа, заключенные между двумя фиксированными значениями  $MIN$  и  $MAX$ , являющимися атрибутами конкретного целого типа. К этим значениям добавляются неопределенное значение ( $\perp$ ) и переопределенное значение ( $\top$ ). Последнее представляет все математические целые, оставшиеся за пределом диапазона  $[MIN, MAX]$  (значение переполнения). Присваивание этих метазначений некоторому объекту может приводить к возникновению особых ситуаций, которые должны воспрепятствовать дальнейшему выполнению программы ввиду его некорректности.

Таким образом, множество значений целого типа, в отличие от множества целых чисел, конечно

$$\bar{\mathbb{Z}} = \{z \in \mathbb{Z} \mid MIN \leq z \leq MAX\} \cup \{\perp, \top\}$$

Над значениями целого типа определены обычные целочисленные арифметические операции, а именно: сложение (+), вычитание (-), умножение (\*), целочисленное деление (/ (для целочисленных операндов!) в Си или `div` в Паскале). То есть в Паскале операция деления для целого типа изображается отдельным символом `div`, а в Си оба деления изображаются одним символом /, что может ввести в заблуждение начинающего программиста. Кроме этих операций дополнительно определены одноместная операция изменения знака числа на противоположный (эту операцию мы будем обозначать знаком «-» либо, чтобы не путать с бинарным минусом, символом `neg` (противоположный)) и двуместная операция вычисления остатка от целочисленного деления (`mod` в Паскале или % в Си).

Для значений целого типа определены отношения порядка ( $>, \geq, <, \leq$ ) и отношения равенства ( $=, \neq$ ). Операнды отношений целочисленные, результаты — логические, т. е. отношения размыкают целый тип, отображая пары целочисленных значений в истинностные значения — значения другого (логического) типа. В отличие от логического типа, отношения операциями не являются, т. к. результат операции должен принадлежать тому же типу, что и операнд.

Целочисленное деление реализовано без остатка, чтобы не размыкать целый тип для получения дробного частного более сложного вещественного типа. Здесь и выше размыкание понимается в алгебраически-кольцевом смысле: результат операции над любыми допустимыми операндами должен оставаться в том же самом множестве (кольце).

Некоторые свойства операций и отношений:

1. коммутативность сложения и умножения:  $\forall X, Y \in \bar{\mathbb{Z}}$      $X + Y = Y + X, X * Y = Y * X;$
2. если  $X \geq Y \geq 0$  или  $X \leq Y \leq 0$ , то  $(X - Y) + Y = X$ ;
3. монотонность операций:  $\forall X, Y \in \bar{\mathbb{Z}}$  из  $0 \leq X \leq A$  и  $0 \leq Y \leq B$  следует

$$\begin{aligned} X + Y &\leq A + B; & X - B &\leq A - Y; \\ X * Y &\leq A * B; & X + B &\leq A + Y; \end{aligned}$$

Всё это верно только если  $A + B \leq MAX$  и  $A * B \leq MAX$  соответственно.

4. если хотя бы один из операндов операции целого типа имеет значение  $\perp$  или  $\top$ , то результат операции тоже имеет значение  $\perp$  или  $\top$  соответственно. Если один операнд двуместной операции имеет значение  $\perp$ , а другой  $\top$ , то результат операции имеет значение  $\perp$ ;
5. если один из operandов отношения целого типа имеет значение  $\perp$  или  $\top$ , то результат имеет значение  $\perp$ . Строго говоря, это неопределенное значение принадлежит логическому типу.

Отметим, что наличие значений  $\perp$  и  $\top$  со сформулированными выше свойствами приводит к тому, что для операций целого типа не выполняются законы ассоциативности и дистрибутивности. Это тоже жертвы эффективной вычислимости по фон Нейману. Например, ассоциативный закон  $(X + Y) + Z = X + (Y + Z)$  не выполняется, если результатом хотя бы одной из операций в левой части является  $\top$ , а в правой части нет операций, имеющих результатом значение  $\top$ , и наоборот. Если для некоторого процессора  $\text{MAX} = 32\,767$ , то  $20\,000 + (30\,000 + (-25\,000)) = 20\,000 + 5\,000 = 25\,000$ , а результат вычисления  $(20\,000 + 30\,000) + (-25\,000) = \top$ , так как  $20\,000 + 30\,000 = \top$ ,  $\top - 25\,000 = \top$ .

16-разрядный целочисленный процессор с дополнительным кодированием целых имеет следующий диапазон:  $[-32\,768..32\,767]$ ; его верхняя граница является предопределенной константой **MAXINT** для короткого целого (O, Borland!). Диапазон 32-разрядного процессора существенно шире:

$$[-2\,147\,483\,648..2\,147\,483\,647].$$

Это число является значением константы MAXINT в GNU и Compaq Pascal. Аналогичная константа INT\_MAX может быть извлечена из стандартной библиотеки языка C (файл `<limits.h>`). В 64-разрядном процессоре представимы числа в диапазоне

$$[-9\,223\,372\,036\,854\,775\,808..9\,223\,372\,036\,854\,775\,807],$$

или `[-maxint64 - 1..maxint64]`, для чего приходится применять нестандартный тип integer64 в Compaq Pascal или стандартный тип **long long int** для 64-битных версий Compaq C (cc) и GNU C (gcc). Эти типы имеют аппаратную поддержку на соответствующих процессорах. Кроме того, беззнаковая версия целого типа используется для вычисления адресов (адресная арифметика). Таким образом, 64-битный процессор имеет адресное пространство 16 777 216 терабайт, или 16 384 петабайт, или 16 экзабайт, что составляет 18 446 744 073 709 551 616 байт!

Рассмотренные операции и отношения позволяют определить двоичный целочисленный процессор  $INT(k)$  с множеством изображений  $W_k = \{a_1 a_2 \dots a_k | a_i \in \{0, 1\}, i = 1, 2, \dots, k\} \cup \{\underbrace{\perp \perp \dots \perp}_{k \text{ букв}}, \underbrace{\top \top \dots \top}_{k \text{ букв}}\}$ . Функция интерпретации  $\varphi$  следующая:

1. Слово из букв 0 и 1 является записью целого числа с использованием дополнительного кода.
2. Слово из букв  $\perp$  соответствует неопределенному значению.
3. Слово из букв  $\top$  соответствует переполнению.

При таком определении процессора  $INT(k)$  константы  $MIN$  и  $MAX$  имеют значения  $-2^{k-1}$  и  $2^{k-1} - 1$  соответственно.

Алгоритмы выполнения операций и вычисления отношений процессора  $INT(k)$  можно конструктивно описать диаграммами Тьюринга.

Опишем сначала специальные машины инкремента и декремента. Они необходимы для выполнения рутинных повторяющихся операций в более сложных арифметических машинах.

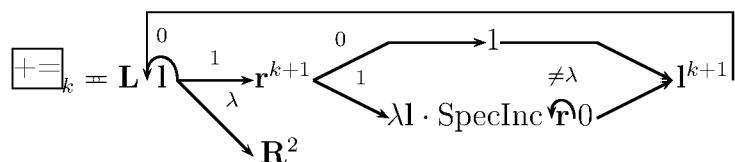


Эти машины в обычном тьюринговском смысле ненормированы. Но при построении машины фон Неймана мы уже прибегали к подобной перенормировке. На базе этих машин построим перенормированные (в смысле положения головки МТ, неизменность аргументов не соблюдается) МТ для инкремента и декремента. Они соответствуют префиксным формам этих операторов в C/C++.

$$\boxed{++}_k = 1 \cdot \text{SpecInc} \cdot R$$

$$\boxed{-}_k = 1 \cdot \text{SpecDec} \cdot R$$

Операция сложения с одновременной записью результата в правый operand аналогична операторам  $\alpha =$  в C/C++ ( $\alpha$  — одна из операций  $+, -, *, /, \%, \&, |, \wedge, \ll, \gg$ ):



Бинарное сложение теперь реализуется очень просто через уже введенную МТ  $\boxed{+=}_k$ .

$$[\lambda w_1 \lambda w_2 (\lambda) \lambda] \xrightarrow{*} [\lambda w_1 \lambda w_2 \lambda \bar{w} (\lambda) \lambda], \text{ где } \varphi(\bar{w}) = \varphi(w_1) + \varphi(w_2), w, w_1, w_2 \in W_k.$$

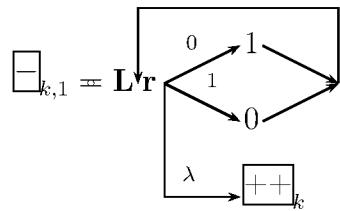
$$\boxed{+}_k = K_2 \boxed{+=}_k$$

То есть мы вручную отнормировали результат ненормированного сложения.

С целью простоты реализации переполнение не отслеживается.

Ненормированный оператор унарный минус меняет на месте знак числа в дополнитель-

ном коде:  $[\lambda w (\lambda) \lambda] \xrightarrow{*} [\lambda \bar{w} (\lambda) \lambda]$ , где  $\varphi(\bar{w}) = -\varphi(w), w \in W_k$ .



Операция вычитания с одновременной записью результата в правый operand аналогична вышеупомянутым операторам вида  $a =$  в C/C++. В данном случае левый operand является вычитаемым, в правый — уменьшаемым. При этом результат помещается в правый operand, т. к. в силу полубесконечности ленты МТ удобно (для дальнейшего использования) все результаты помещать справа от operandов. Эта машина, хотя и не создает новых слов, помещает результат именно так для однообразия и унификации с другими машинами. Ее operandы записаны в обратном порядке, т. к. в C/C++  $a = b$  означает  $a = a - b$ , то есть operand-результат является уменьшаемым.

$$\boxed{-=}_k = L \boxed{\ominus}_{k,1} \boxed{+}_k L \boxed{\ominus}_{k,1} R$$

Бинарный оператор вычитания с записью результата в новое слово.

$$\boxed{\ominus}_k = K_2 \boxed{-=}_k$$

Теперь опишем принцип работы машины двоичного умножения. Пусть слова фиксированной длины  $k$   $w_1$  и  $w_2 = \overline{b_k b_{k-1} \dots b_1}$  содержат два двоичных числа. Поскольку позиционная система счисления является полиномиальной интерпретацией последовательности цифр числа, то умножение чисел, по сути, есть умножение соответствующих многочленов с двоичными коэффициентами, вычисленных в точке 2. Тогда  $\varphi(w_1)\varphi(w_2) = \varphi(w_1) \sum b_i 2^{i-1} = \sum \varphi(w_1)b_i 2^{i-1}$ . Умножение на многочлен проще делать почленно, причём ввиду двоичности используемой системы счисления слагаемые этой суммы либо будут присутствовать (с коэффициентом 1), либо нет, если соответствующая двоичная цифра  $b_i$  равна 0. Таким образом, можно получить произведение двух чисел путем однократных сложений (если на данном месте в записи числа стоит 1) и сдвигов влево.

В отличие от машин сложения, машина умножения использует дополнительное слово для хранения результата. Вот схема ее работы:

$$[\lambda w_2 \lambda w_1 (\lambda) \lambda] \Rightarrow [\lambda w_2 \lambda w_1 \lambda 0 (\lambda) \lambda] \Rightarrow [\lambda w_2 \lambda w'_1 \lambda w (\lambda) \lambda] \Rightarrow [\lambda w_2 \lambda w (\lambda) \lambda].$$

Для реализации работы машины по приведенной схеме необходимо описать несколько вспомогательных машин. Первой такой машиной является машина ненормированного арифметического сдвига разрядов в машинном слове  $w$  влево на один разряд  $([(\lambda)w \lambda] \Rightarrow [\lambda w'(\lambda)])$ , где  $w = b_k b_{k-1} \dots b_2 b_1$ , а  $w' = b_k b_{k-2} b_{k-3} \dots b_2 b_1 0$ . Здесь  $b_k$  — «знаковый» разряд дополнительного кода.

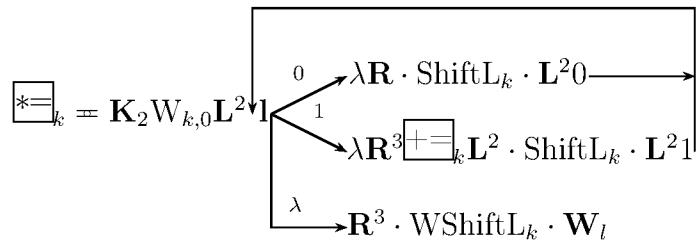
$$\text{ShiftL}_k = r \left( r^2 \begin{array}{c} 0 \\ \swarrow \searrow \\ 1 \end{array} \right)^{k-2} r 0 r$$

Действие в скобках — это перенос одного бита справа налево. Однако при этом машина находится не на той позиции, куда будет скопирован очередной бит. Можно слегка изменить диаграмму (отказавшись от двух машин  $\mathbf{r}$  в начале итерации), чтобы она копировала правый бит на текущую позицию и передвигалась влево, вместо того, чтобы сначала передвигаться влево, а потом копировать биты, как это реализовано выше.

Еще одна полезная машина —  $W_{k,0} ([(\lambda)\lambda] \Rightarrow [\lambda \underbrace{00\dots 0}_{k} (\lambda)\lambda])$ . Пустые ячейки берутся справа на свободном краю ленты:

$$W_{k,0} = (\mathbf{r}0)^k \mathbf{r}$$

Собственно реализация машины умножения:



Использованная в машине умножения машина  $WShiftL_k$  сдвигает (фактически, копирует) слово слева от головки МТ в предыдущее слово ( $[\lambda w_1 \lambda w_2 (\lambda)\lambda] \Rightarrow [\lambda w_2 \lambda w_2 (\lambda)\lambda]$ ):

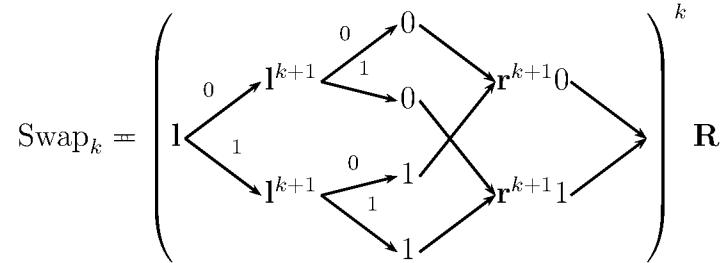
$$WShiftL_k = I \left( \begin{array}{ccc} 0 & \xrightarrow{\quad} & I^{k+1} 0 \\ 1 & \xrightarrow{\quad} & I^{k+1} 1 \end{array} \right)^k R$$

Таким образом, умножение двоичных чисел разбивается на более простые аддитивные стадии. Сначала справа от второго операнда на свободном краю ленты записывается слово из  $k$  нулей, которое будет использоваться в качестве рабочей ячейки, где и будет аккумулироваться результат умножения. Затем выполняется сложение второго операнда с этим словом ( $[\square_k]$ ), соответствующее умножению второго операнда на очередной (ненулевой двоичный!) разряд первого, и сдвиг, происходящий при любом значении этого разряда. Сложения и сдвиги происходят вплоть до полного перебора всех разрядов левого операнда. Сложность этого алгоритма *относительно сложений и сдвигов линейна* ( $k - 1$  сдвиг и не более  $k$  сложений), а *относительно длины слова* — квадратична ( $O(k^2)$ : не более  $k$  сдвигов-умножений, каждый из которых имеет линейную сложность относительно  $k$ ). На самом деле эта сложность сравнительно невелика, т. к. мы используем известную своей экономностью позиционную систему счисления, а не экспоненциально-сложностную натуральную. Заметим, что если бы это была не двоичная система счисления, то перед сложением дополнительно потребовалось бы умножение операнда на цифру, что усложнило и замедлило бы реализацию умножения.

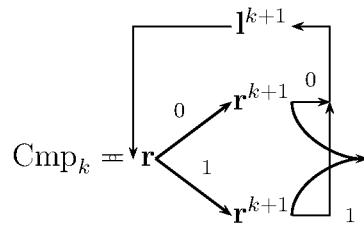
В современных процессорах применяются параллельные схемы умножения, имеющие постоянную сложность (1 такт процессора!), поскольку размер слова фиксирован [18, 79].

Для реализации машины деления потребуется несколько вспомогательных машин. (Напомним, что для удобства реализации и совместимости как с другими МТ, так

и с языками C/C++ и типичными микропроцессорами, в которых результат замещает на регистре один из операндов, первым словом является делитель, а вторым — делимое.) Первой из таких машин является машина  $\text{Swap}_k$ , которая обменивает местами два слова ( $[\lambda w_1 \lambda w_2 (\lambda) \lambda] \Rightarrow [\lambda w_2 \lambda w_1 (\lambda) \lambda]$ ):



Машина Стр тоже не нормирована. Она сравнивает поразрядно слева направо два слова длины  $k$  и останавливается во втором слове при обнаружении первого несоответствия.

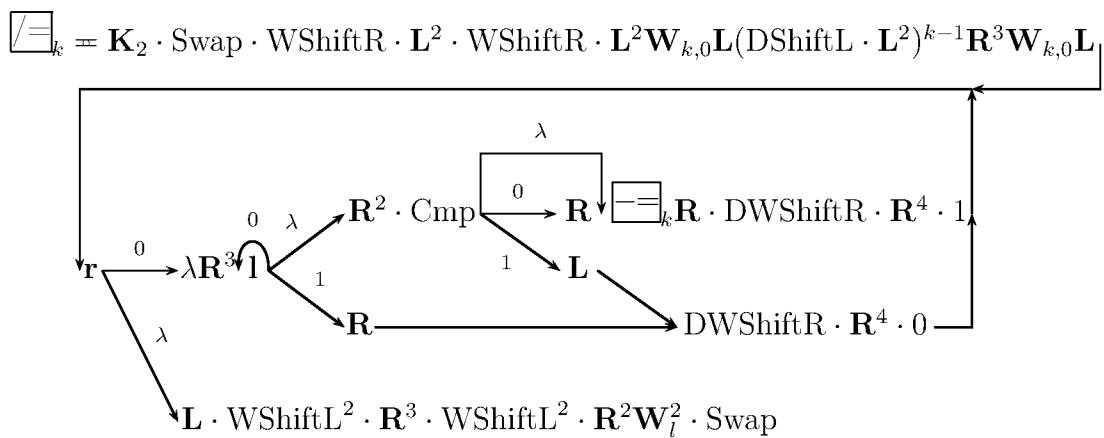


Теперь опишем принцип работы машины деления. Как и в случае умножения, двоичная система серьезно упрощает сам процесс. Дело в том, что школьное деление «столбиком» не является алгоритмом, т. к. в вычислении очередной цифры присутствует элемент догадки — подбор релевантного кратного делителю. К счастью, в двоичной системе достаточно сравнить делимое и сдвинутый влево делитель. Далее, если делимое больше, то в соответствующий разряд частного необходимо записать единицу и вычесть делитель из делимого. Иначе в этот разряд частного помещается нуль. В любом случае необходимо осуществить сдвиг делителя влево. Процесс заканчивается через  $k$  шагов. В итоге получается частное и остаток.

Однако этот алгоритм в описании выглядит просто. Его реализация сложнее. В первую очередь это связано с необходимостью манипулировать сразу двумя вспомогательными словами. Если перейти к деталям, то машина деления работает так:

1. Делитель копируется направо от делимого, после чего эти два слова меняются местами для удобства организации процесса деления.
2. Перед делителем добавляется дополнительное слово из нулей, куда будет помещаться та часть делителя, которая «вылезает» из слова при его сдвиге влево. После делимого также записывается слово из  $k$  нулей, куда будем помещено частное.
3. Делитель сдвигается на  $k - 1$  разряд влево.
4. Первый бит частного помечается (замещается  $\lambda$ ), чтобы правильно позиционировать машину в циклах, вычисляющих биты частного.

5. Происходит сравнение делителя и делимого. При этом учитывается содержимое дополнительного слова, буферизующего сдвиг.
6. Если делитель меньше делимого, то происходит вычитание сдвинутого делителя из делимого и запись единицы в соответствующий разряд частного. В противном случае вычитания не происходит, а в этот разряд записывается нуль.
7. Делитель сдвигается на 1 разряд вправо. Помечается следующий бит. Если достигнут конец слова, то стираются лишние слова (делитель и дополнительное слово для хранения сдвинутого делителя), и остаток и частное меняются местами. Иначе деление продолжается с п. 5.



Еще не определены машины DShiftR и DShiftL, осуществляющие арифметический сдвиг двух двоичных слов подряд с переносом битов в соседнее слово вправо и влево соответственно. Схемы их работы:

$$\begin{aligned} \text{DShiftR: } & [\lambda w_1 \lambda w_2 (\lambda) \lambda] \Rightarrow [(\lambda) w'_1 \lambda w'_2 \lambda] \\ \text{DShiftL: } & [(\lambda) w_1 \lambda w_2 \lambda] \Rightarrow [\lambda w'_1 \lambda w'_2 (\lambda) \lambda] \end{aligned}$$

$$\begin{aligned} \text{DShiftL}_k &= \left( \begin{array}{ccc} 0 & \xrightarrow{\quad} & 10 \\ r^2 & \swarrow & \searrow & \xrightarrow{\quad} & 11 \\ & 1 & & & \end{array} \right)^{k-1} r^3 \left( \begin{array}{ccc} 0 & \xrightarrow{\quad} & 10 \\ r & \swarrow & \searrow & \xrightarrow{\quad} & 11 \\ & 1 & & & \end{array} \right)^{k-2} r0r \\ \text{DShiftR}_k &= \left( \begin{array}{ccc} 0 & \xrightarrow{\quad} & r0 \\ l^2 & \swarrow & \searrow & \xrightarrow{\quad} & r1 \\ & 1 & & & \end{array} \right)^{k-1} l^3 \left( \begin{array}{ccc} 0 & \xrightarrow{\quad} & r0 \\ l & \swarrow & \searrow & \xrightarrow{\quad} & r1 \\ & 1 & & & \end{array} \right)^{k-2} l0l \end{aligned}$$

Если определить дополнительную машину для записи слова  $\underbrace{0 \dots 0}_{k-1} 1$ :

$$\mathbf{W}_{k,1} = (r0)^{k-1} r1 r$$

то можно на основе машины Смр создать нормированные машины для сравнения слов, работающие по схеме  $[\lambda w_1 \lambda w_2 (\lambda) \lambda] \Rightarrow [\lambda w_1 \lambda w_2 \lambda \bar{w} (\lambda) \lambda]$ .

Таким образом, целый тип также вычислим по Тьюрингу.

Следует отметить также, что наиболее привычной системой счисления для человека является десятичная позиционная система. Как правило, именно в ней выполняются

все расчеты вне вычислительной машины. Подавляющее же большинство современных машин оперирует с числами, представленными в системе счисления с другим основанием. (например, с основанием 2, или 8, или 16). Поэтому при вводе и выводе данных возникает необходимость перевода из одной системы счисления в другую (например, из десятичной в двоичную, из десятичной в шестнадцатиричную и наоборот). Рассмотрим решение этой задачи для данных целого типа.

Задача перевода заключается в следующем. Пусть известно изображение числа  $X$  в системе счисления с основанием  $P$ :

$$X = (p_n p_{n-1} \dots p_1 p_0)_P,$$

где  $p_i$  - цифры  $P$ -ичной системы счисления ( $0 \leq p_i \leq P-1$ ). Требуется найти изображение этого же числа  $X$  в системе счисления с другим основанием  $Q$ :

$$X = (q_s q_{s-1} \dots q_1 q_0)_Q,$$

где  $q_i$  — искомые цифры  $Q$ -ичной системы счисления ( $0 \leq q_i \leq Q-1$ ). При этом можно ограничиться случаем положительных чисел, поскольку перевод любого числа сводится к переводу его модуля и приписыванию числу нужного знака.

При рассмотрении правил перевода необходимо учитывать, средствами какой арифметики должен быть осуществлен перевод, т. е. в какой системе счисления должны быть выполнены все необходимые для перевода арифметические действия. Эти действия могут быть выполнены как в системе счисления с основанием  $P$ , так и в системе счисления с основанием  $Q$ . Чтобы охватить все возможные случаи, рассмотрим перевод из  $Q$ -ичной системы в  $P$ -ичную (перевод  $Q \rightarrow P$ ) и обратный перевод (перевод  $P \rightarrow Q$ ) с выполнением действий в системе счисления с основанием  $P$ .

Перевод  $Q \rightarrow P$ . Задача перевода произвольного числа  $X$ , заданного в системе счисления с основанием  $Q$ ,

$$X = (q_s q_{s-1} \dots q_1 q_0)_Q$$

в систему счисления с основанием  $P$  сводится к вычислению значения соответствующего полинома. Действительно, в силу последней формулы

$$X = q_s \cdot Q^s + q_{s-1} \cdot Q^{s-1} + \dots + q_1 \cdot Q^1 + q_0.$$

Для получения  $P$ -ичного изображения достаточно все цифры  $q_i$  и число  $Q$  заменить их  $P$ -ичными изображениями и произвести все требуемые по формуле операции в  $P$ -ичной системе счисления. Например, пусть требуется перевести число  $X = (371)_8$  в десятичную систему счисления, пользуясь средствами десятичной арифметики; здесь  $P = 10, Q = 8$ . Для перевода запишем число  $X$  в виде  $X = 3 \cdot 8^2 + 7 \cdot 8^1 + 1$  и выполним все необходимые действия в десятичной системе счисления:  $X = 3 \cdot 64 + 7 \cdot 8 + 1 = 192 + 56 + 1 = 249$ .

Еще пример. Пусть требуется перевести число  $X = (4315)_7$  в троичную систему счисления, пользуясь средствами троичной арифметики; здесь  $P = 3, Q = 7$ . Запишем  $X = 4 \cdot 7^3 + 3 \cdot 7^2 + 1 \cdot 7^1 + 5$ . Все цифры и основание семеричной системы счисления заменим их троичными изображениями:  $X = 11 \cdot 21^{10} + 10 \cdot 21^2 + 1 \cdot 21^1 + 12$ . Выполним все требуемые по формуле операции в троичной системе счисления:  $X = 11 \cdot 110201 + 10 \cdot 1211 + 1 \cdot 21 + 12 = 1212211 + 12110 + 21 + 12 = 2002201$ . Итак,  $X = (4315)_7 = (2002201)_3$ .

Перевод  $P \rightarrow Q$ . Пусть известно изображение целого числа  $X$  в системе счисления с основанием  $P$  и требуется перевести это число в систему счисления с основанием  $Q$ . Запись этого числа в  $Q$ -ичной системе счисления будет иметь вид

$$X = (q_s q_{s-1} \dots q_1 q_0)_Q, \quad (1)$$

где  $q_i$  ( $i = 0, 1, \dots, s$ ) - цифры  $Q$ -ичной системы счисления, которые надо определить. Для определения  $q_0$  разделим обе части равенства  $X = q_s \cdot Q^s + q_{s-1} \cdot Q^{s-1} + \dots + q_1 \cdot Q^1 + q_0$  на число  $Q$ , причем в левой части равенства произведем фактическое деление, поскольку запись числа  $X$  в системе счисления с основанием  $P$  нам известна, а в правой части деление выполним аналитически:

$$\frac{X}{Q} = q_s \cdot Q^{s-1} + q_{s-1} \cdot Q^{s-2} + \dots + q_1 + \frac{q_0}{Q}.$$

Приравняем между собой полученные целые и дробные части, учитывая при этом, что  $q_i < Q$ :

$$\begin{aligned} \text{целая часть } \left(\frac{X}{Q}\right) &= q_s \cdot Q^{s-1} + q_{s-1} \cdot Q^{s-2} + \dots + q_1; \\ \text{дробная часть } \left(\frac{X}{Q}\right) &= \frac{q_0}{Q}. \end{aligned}$$

Отсюда получим, что младший коэффициент  $q_0$  в разложении (1) определяется соотношением  $q_0 = Q \cdot$  дробная часть  $\left(\frac{X}{Q}\right)$ , причем указанные здесь действия на самом деле не выполняются, так как  $q_0$  является просто остатком от деления  $X$  на  $Q$ .

Положим далее  $X_1 =$  целая часть  $\left(\frac{X}{Q}\right)$ , т.е.

$$X_1 = q_s \cdot Q^{s-1} + q_{s-1} \cdot Q^{s-2} + \dots + q_2 \cdot Q^1 + q_1.$$

Тогда  $X_1$  будет целым числом, и чтобы определить следующую искомую цифру  $q_1$ , к нему можно применить ту же самую процедуру и т.д.

Этот процесс следует продолжать до тех пор, пока не будет получено частное, равное нулю. Заметим, что поскольку все операции выполняются в системе счисления с основанием  $P$ , то в этой же системе счисления будут получены и искомые коэффициенты  $q_i$ . Для окончательной записи числа  $X$  в  $Q$ -ичной системе счисления необходимо каждый из полученных коэффициентов  $q_i$  записать одной  $Q$ -ичной цифрой. Например, пусть требуется перевести число  $X = 3060$  в шестнадцатиричную систему счисления с использованием десятичной арифметики. Применим только что обсужденное правило:

$$\begin{array}{r} 3060 \mid 16 \\ \underline{-16} \quad 191 \mid 16 \\ \underline{-146} \quad \underline{16} \quad 11 \mid 16 \\ \underline{\underline{144}} \quad \underline{31} \quad \underline{0} \quad 0 \\ \underline{-20} \quad \underline{16} \quad 11 \\ \underline{\underline{16}} \quad \underline{15} \\ 4 \end{array}$$

Таким образом,  $q_0 = (4)_{10}; q_1 = (15)_{10}; q_2 = (11)_{10}$ . Для окончательной записи числа  $X$  в шестнадцатиричной системе нужно каждый из этих коэффициентов записать одной шестнадцатиричной цифрой:  $X = (\text{BF}4)_{16}$ .

Если основание целевой системы счисления  $Q$  меньше основания объектной  $C$ , то цифрами могут служить небольшие целые числа аналогично тому, как при кодировании большего алфавита в меньший мы использовали составные знаки-слова.

Паскаль-программы интерпретации изображений вводимых и выводимых целых чисел приведены в п. 13 пособия Н. Вирта [8] и в п. 3.2.11.

Изученные нами алгоритмы работы с изображениями целых чисел могут быть полезны для реализации так называемой *длинной арифметики*. В некоторых задачах важно производить вычисления большой точности или с очень большими числами, не поддерживаемые аппаратурой. Для этого используется программные реализации работы с изображениями. Программная поддержка длинной арифметики имеется в языках Python (встроенный тип или перегрузка операторов) и Java (библиотечные функции). Позднее мы приведем реализации длинной арифметики на Паскале и на Си, так необходимые на олимпиадах по программированию.

## Лекция 20

### 3.2.4 Тип вещественный

Как известно из курса математического анализа [82, 93], следующим за целыми числами уровнем числовой абстракции стали рациональные числа, позволившие выражать отношения и пропорции. Рациональное число можно представить (не единственным образом!) в виде отношения целого и натурального

$$r = \frac{p}{q}, \quad r \in \mathbb{Q}, \quad p \in \mathbb{Z}, \quad q \in \mathbb{N}.$$

Рациональные числа используются для более точного измерения отрезков и промежутков времени. Но еще в Древней Греции последователи Пифагора обнаружили несоизмеримость диагонали квадрата и его стороны [81] и длины окружности с её диаметром, выражаемых иррациональными числами. Позже, в середине XIX века, было установлено, что множество вещественных чисел несчётно и всюду плотно. В частности, вся эта бесконечная и всюду плотная числовая ось с помощью процедуры диагонализации Кантора однозначно отображается в любой конечный отрезок этой же самой числовой оси. Из этого следует, что простым ограничением диапазона вещественных чисел мы не загоним этот отрезок в прокрустово ложе машинного слова ограниченной длины, как мы это успешно сделали для целого типа. Необходимо предпринять более кардинальные меры.

Естественным ограничением для представления одного числа в одном машинном слове является отсутствие бесконечных или конечных, но очень длинных чисел. Это означает, что мы вынуждены будем опустить при машинном представлении не только все иррациональные числа, но и подавляющее большинство рациональных чисел, за исключением лишь конечного их числа (если, конечно, мы не будем прибегать к такой экзотике, как системы счисления с иррациональными основаниями!?).

Если поручить членам этого оставшегося конечного множества «представительские» функции относительно всех невошедших в президиум соседних чисел в некоторой окрестности, то мы получим машинный суррогат великого и ужасного вещественного континуума в виде конечных рациональных приближений. Вместо обычных конкретно-числовых множеств мы получим интервальные. Складывая два таких числа-интервала своеобразной интервальной арифметикой, мы получим интервал для суммы.

Для решения проблемы машинного представления вещественных чисел осталось только предложить кодировку этих конечных рациональных приближений кодовыми комбинациями машинного слова.

**Замечание.** При использовании одного и того же машинного слова для представления целых и вещественных чисел оказывается, что машинных вещественных чисел даже меньше, чем целых ввиду неоднозначности представления.

Как же все-таки машинные вещественные числа смогут выполнять свои исторические предначертания в расширении диапазона и в удобстве представлении дробных чисел? В 1937 году, еще до создания первых компьютеров немецкий инженер Конрад Цузе, основываясь на экспоненциальной форме записи числа, предложил *полулогарифмический* формат представления вещественных чисел в ЭВМ.

Число  $-1.23$  в экспоненциальной форме может быть представлено как  $-1.23 \cdot 10^0$ , или как  $-0.123 \cdot 10^1$ , или как  $-12.3 \cdot 10^{-1}$ . То есть число разлагается на произведение (неправильной) дроби (*мантизы*) и экспоненты (*экспоненты*), являющейся целой частью *логарифма* этого числа по основанию порядка. У дроби есть целая и дробная части, которые в данной системе счисления интерпретируются полиномиальным способом, причем для вычисления дробной части используется полином по отрицательным степеням.

$$\sum_i \varphi(a_i)p^{-i}$$

При этом основания системы счисления для дроби и для экспоненты, вообще говоря, могут быть разными. Цузе предложил число  $-1.23$  перевести в нормализованную форму  $-0.123 \cdot 10^1$  и представить в ЭВМ тремя частями одного машинного слова.

—	1	123
---	---	-----

Тем самым мы получили весьма экономное представление вещественного числа, опустив ведущий 0, десятичную точку, символ порядка (основание системы счисления для представления порядка) и связывающий элементы экспоненциальной записи знак умножения. Кроме того, ранее было опущено также основание системы счисления для целой и дробной частей числа.

Мы сразу же поместили знак числа в отдельное поле машинного слова, потому что при представлении отрицательных вещественных чисел в ЭВМ дополнительное кодирование не дает таких замечательных преимуществ как в случае целого типа.

Вышеприведенные примеры записи числа  $-1.23$  показывают неоднозначность такого представления вещественных чисел в ЭВМ. Однако существует простой способ избавиться от этой неоднозначности: необходимо потребовать, чтобы дробь была правильной и старший разряд числа всегда был ненулевым, подкорректировав соответствующим образом порядок. Такое однозначное представление числа называется *нормализованным*.

Опустив точку в записи числа мы предполагаем ее наличие перед первым (ненулевым!) разрядом записи числа. При делении числа  $-1.23$  на 10 с последующей нормализацией и подразумеваемая десятичная точка, и цифры 123 останутся на месте. Чтобы правильно представить нормализованное частное при неподвижной точке надо чтобы «поплыл» порядок. Поэтому полулогарифмическое представление называется еще и представлением *с плавающей точкой* (*запятой*). Хотя этот устоявшийся термин следует признать неудачным, т. к. на самом деле плавает не точка, а порядок!

Поскольку необходимо представлять не только очень большие числа, но и малые около нуля, то порядки чисел могут быть и положительными, и отрицательными, и возникает проблема представления знака порядка. Обычно она решается с помощью специального кодирования порядка со смещением; смещённый код порядка называется *характеристикой*. Характеристика может рассматриваться как разновидность дополнительного кода порядка. В реальных компьютерах для представления вещественных чисел используются двоичные машинные слова, и основание системы счисления для мантиссы подразумевается тоже двоичное. А вот для расширения диапазона представимых чисел основание порядка (характеристики) иногда бывает шестнадцатиричным (IBM/360). Это, правда, понижает точность представления, поскольку в один и тот же диапазон при меньшем порядке попадет больше чисел.

Итак, мы представили вещественные числа в ЭВМ конечными рациональными приближениями. За пределами нашего рассмотрения пока осталась точность этого приближения — ширина отрезка, представляемого данным числом. Наиболее простым решением было бы выбрать ширину этого отрезка  $\Delta x$  постоянной для всех чисел. Однако  $\Delta x = \text{const } \forall x$  бессмысленно с физической точки зрения, т. к. абсолютная погрешность  $\Delta x$  измерения величины  $x$  должна быть обязательно соотнесена с её значением. Из этого следует, что ширина отрезков должна подчиняться соотношению  $\frac{\Delta x}{x} = \text{const}$ , иными словами постоянной должна быть относительная погрешность. Прибегая к дифференциальному исчислению  $\frac{dx}{x} = \text{const}$ , получим, что ширина отрезка меняется по экспоненциальному закону. Таким образом, для малых чисел ширина отрезка мала, а для больших — пропорционально велика и на всем множестве машинных вещественных чисел обеспечивается одинаковая относительная погрешность представления.

Теперь мы можем заняться формальным определением вещественного типа.

Множество изображений вещественного типа — это множество слов конечной длины над алфавитом  $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \cup \{+, -, ., \mathbf{E}\}$ , причем длина слова, изображающего вещественное число, в языке программирования не фиксирована и определяется с помощью соответствующих базовых атрибутов, которые имеют определенное значение для каждой реализации.

Функция интерпретации определяется следующим образом:

$$\begin{aligned} \varphi_{\text{REAL}}(s_1 a_0 a_1 \dots a_c. a_{c+1} a_{c+2} \dots a_d \mathbf{E} s_2 a_{d+1} a_{d+2} \dots a_k) &= \\ &\quad \begin{array}{l} \text{значок} \\ \text{числа} \end{array} \quad \begin{array}{l} \text{десятичная} \\ \text{точка} \end{array} \quad \begin{array}{l} \text{знак} \\ \text{порядка} \end{array} \quad \begin{array}{l} \text{символ} \\ \text{порядка} \end{array} \\ &= s_1 \left( \sum_{i=0}^c a_i \cdot 10^{c-i} + \sum_{i=c+1}^d a_i \cdot 10^{c-i} \right) \cdot 10^{\sum_{i=d+1}^k a_i \cdot 10^{k-i}} \end{aligned}$$

где  $a_0, a_1, \dots, a_c$  — цифры, изображающие целую часть вещественного числа;  $a_d, a_{d+1}, \dots, a_k$  — цифры, изображающие дробную часть вещественного числа.

Рассмотренное выше изображение получило название *представление с плавающей точкой*. Фактически это специальным образом нормализованное экспоненциальное представление числа. Ранее существовало также *представление с фиксированной точкой*, при котором точка подразумевалась в фиксированном месте машинного слова. Поскольку точность представления с фиксированной точкой также фиксирована, то оно может применяться только для ограниченного диапазона чисел, в котором фиксированное представление имеет одинаковую *абсолютную* ( $\Delta x = \text{const}$ ) погрешность. Фиксированная арифметика использовалась на бортовых и других встроенных компьютерах, работавших в реальном масштабе времени, а также используется в современных СУБД (например, для представления денежных величин). В некоторых языках программирования, ориентированных на эти сферы, есть вещественный тип с фиксированной точкой. Это не только Ada и SQL, но и Borland Delphi/Builder (тип Currency).

Итак, вещественное число с плавающей точкой фактически изображается с помощью двух целых чисел  $p$  и  $m$ , каждое из которых содержит конечное (фиксированное для каждой реализации) число цифр, так что вещественное число  $X = m * B^p$ , причем  $\minexp \leq p \leq \maxexp$ ;  $\minmant \leq m \leq \maxmant$ ;  $m$  называется *мантиссой*;  $p$  — *порядком*, а  $B$ ,  $\minexp$ ,  $\maxexp$ ,  $\minmant$  и  $\maxmant$  являются константами, характеризующими представление; число  $B$  называется *основанием* представления с плавающей точкой. Итак,  $\text{ИНТ}_{\text{вещ}}(m, p) = m * B^p$ .

Для любого числа  $X$  можно найти столько различных пар  $(m, p)$ , сколько позволяет ширина мантиссы. Например, при  $B = 10$  число  $X = 15.431$  можно с плавающей точкой представить и как  $0.15431 \cdot 10^2$ , и как  $15.431 \cdot 10^0$ , и как  $1.5431 \cdot 10^{+1}$ , и как  $154.31 \cdot 10^{-1}$ , и как  $1543.1 \cdot 10^{-2}$  и т. д. Обычно рассматривают *нормализованные* представления вещественных чисел, которые определяются дополнительным соотношением

$$\frac{\maxmant}{B} \leq |m| \leq \maxmant.$$

Таким образом, любое вещественное число может быть представлено в нормализованном виде с помощью двух целых чисел. При этом пара  $(m, p)$  дает приближенное представление вещественного числа. Дело в том, что из-за конечной длины всего слова и мантисса, и порядок представляются конечным количеством цифр. Пусть каждая мантисса содержит  $l$  цифр, а порядок —  $q$  цифр; тогда  $l$  определяет точность представления числа, а  $q$  — диапазон представляемого числа. И мантисса, и порядок могут иметь знак.

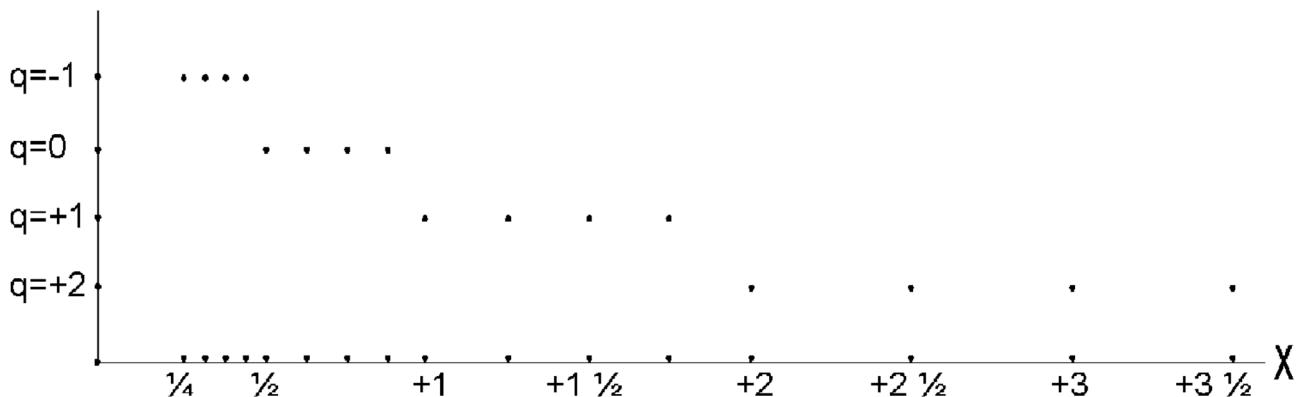
Множество вещественных чисел, каждое из которых представлено конечным числом, не является континуумом. Это конечное множество; в нем ровно  $2 \cdot (B - 1) \cdot B^{l-1} \cdot (\maxexp - \minexp + 1) + 1$  чисел. Они расположены на числовой оси неравномерно. Равномерность распределения имеет место лишь при фиксированном показателе степени (значения порядка). Для примера построим систему положительных нормализованных чисел с плавающей точкой, для которых  $B = 2$ ;  $l = 3$ ;  $\minexp = -1$ ;  $\maxexp = +2$ . Значения этих чисел определяются по формуле

$$X = \pm \left( \frac{a_1}{B} + \frac{a_2}{B^2} + \cdots + \frac{a_l}{B^l} \right) \cdot B^q = \pm \left( \frac{a_1}{2} + \frac{a_2}{4} + \frac{a_3}{8} \right) \cdot 2^q.$$

В заданном диапазоне изменения  $[\minexp, \maxexp]$  порядок  $q$  имеет значения:  $-1$ ,  $0$ ,  $+1$ , и  $+2$ . Полученную систему чисел можно представить в форме табл. 1, в которой для удобства построения графика числа записаны в виде обыкновенных дробей.

Цифры мантийсы	Порядок			
	$q = -1$	$q = 0$	$q = 1$	$q = 2$
000	0	0	0	0
100	$\frac{4}{16}$	$\frac{4}{8}$	$\frac{5}{4}$	$\frac{5}{2}$
101	$\frac{5}{16}$	$\frac{5}{8}$	$\frac{5}{4}$	$\frac{5}{2}$
110	$\frac{6}{16}$	$\frac{6}{8}$	$\frac{6}{4}$	$\frac{3}{2}$
111	$\frac{7}{16}$	$\frac{7}{8}$	$\frac{7}{4}$	$\frac{7}{2}$

Проиллюстрируем полученные результаты на рисунке 1.



При использовании только нормализованной формы представления плотность элементов множества вещественных чисел на оси вещественных чисел экспоненциально уменьшается с увеличением  $|X|$ . Например, промежуток  $[0.1..1]$  содержит при  $B = 10$  столько же элементов рассматриваемого множества, сколько промежуток  $[10000..100000]$ .

Таким образом, множество значений вещественного типа является конечным подмножеством множества действительных чисел  $\mathbb{R}$ , определенного в математике:  $\text{REAL} \subset \mathbb{R}$ . В множестве действительных чисел сначала выделяется подмножество  $\mathcal{R}$ , которое называется *диапазоном представления*. Множество  $\mathbb{R} \setminus \mathcal{R}$  называется *диапазоном переполнения*. Множество  $\mathcal{R}$  представляет собой замкнутое связное множество, т. е. промежуток:  $\mathcal{R} = [\min \text{REAL}, \max \text{REAL}]$ .

Каждому числу  $X \in \mathcal{R}$  ставится в соответствие  $\tilde{X} \in \text{REAL}$ , где  $\tilde{X}$  — представитель  $X$ . Каждое  $\tilde{X}$  является представителем бесконечного множества чисел из диапазона представления. В приведенном выше примере (при  $l = 3$ ) ясно видно, что любой промежуток числовой оси между отмеченными точками содержит бесконечное количество действительных чисел. Однако вместо любого действительного числа, попадающего внутрь указанных интервалов, приходится использовать границу (правую или левую — это зависит от принятого правила округления) соответствующего отрезка. Эта граница и является представителем действительного числа в множестве значений типа  $\text{REAL}$ . Для числа  $X \in \mathcal{R}$  справедливо неравенство [83]

$$\left| \frac{\tilde{X} - X}{X} \right| \leq \frac{1}{2} \cdot B^{1-l}.$$

В рассматриваемом примере вещественные числа 0 и 1 сами являются своими представителями (см. рис. 1).

Масштаб диапазона представления для 128-битной аппаратно реализованной на Digital Alpha вещественной арифметики IEEE Quadruple-Precision X\_floating впечатляет: от

$$\text{MINQUADRUPLE} = 3.36210314311209350626267781732175E-4932$$

и до

$$\text{MAXQUADRUPLE} = 1.18973149535723176508575932662801E+4932.$$

Серийно выпускаемые 32-х разрядные процессоры Intel и AMD имеют не более, чем 80-разрядную вещественную арифметику IEEE Double-Precision T\_floating с более скромными характеристиками: от MINDOUBLE = 2.2250738585072014E-308 и до MAXDOUBLE = 1.7976931348623157E+308. Однако, нельзя не отметить большие успехи в реализации вещественного типа в видеокартах: ещё 10 лет тому назад его там как правило, не было, а сейчас серийно выпускаются 128-битные реализации вещественного типа и 64-битные целого типа.

В связи с тем, что подавляющее количество значений типа REAL представляется приближенно, возникает вопрос о точности выполнения действий над такими числами (о точности плавающей арифметики). Эту точность можно характеризовать с помощью так называемого *машинного эпсилон*, т. е. наименьшего числа с плавающей точкой  $\varepsilon$ , такого, что  $1 \oplus \varepsilon > 1$ . Существование машинного эпсилон хорошо видно на рис. 1: за единицей на протяжении машинного эпсилон на числовой оси нет представителей. Вычисление машинного эпсилон именно в районе единицы связано с тем, что нормализация числа приводит мантиссу к правильной дроби без ведущих нулей после запятой.

Существует несколько методов для определения приближенного значения машинного эпсилон например, на основании конкретных технических характеристик аппаратной реализации вещественного типа. Кроме того, программа может сама определить точность выполнения действий над данными типа REAL в той машине, на которой она выполняется, *во время своего исполнения*. Метод, которым вычисляется приближение, иллюстрируется следующим фрагментом программы в нотации Э. Дейкстры:

```
 $\varepsilon := 1.0$ 
do  $(1.0 + \varepsilon / 2.0 > 1.0)$ ?  $\varepsilon := \varepsilon / 2.0$ 
od
```

В литературе [83] часто приводится программа вычисления величины, отличающейся от машинного эпсилон самое большое в 2 раза.

В Паскаль-программе вычисления  $\varepsilon$  для 128-битного аппаратно реализованного вещественного типа DEC Alpha тип real переопределяется:

```
program GetAlphaEpsilon(output);
type real = quadruple;

var  $\varepsilon$  : real;

begin
   $\varepsilon := 1.0$ ;
  while  $(1.0 + \varepsilon / 2.0) > 1.0$  do
     $\varepsilon := \varepsilon / 2.0$ ;
  writeln( $\varepsilon : 5 - \text{round}(\log(\varepsilon))$ );
end.
```

```

llies@axp3:~$ cat ./demo.cpp
#include <iostream>
#include <limits>
#include <iomanip>
#include <cmath>

int main(void)
{
    long double epsilon = 1.0;
    while((1.0 + epsilon / 2.0) != 1.0)
        epsilon /= 2.0;
    std::cout << std::setprecision(std::static_cast<int>(1.0 - std::log10(epsilon))) <<
    "Calculated: " << epsilon << ", STL: " << std::numeric_limits<long double>::eps
    ion() << std::endl;
    return 0;
}
llies@axp3:~$ g++ -ansi -pedantic ./demo.cpp
llies@axp3:~$ ./a.out
Calculated: 1.925929944387235853055977942584927e-34, STL: 1.92592994438723585305
5977942584927e-34
llies@axp3:~$ 

```

Рис. 3.1: Машинное эпсилон для DEC Alpha



Рис. 3.2: Машинное эпсилон для WinCalc

Результат вычисления будет выведен с 33 значащими цифрами:

1.92592994438723585305597794258493E-0034.

Впрочем, это число доступно в Compaq Pascal как предопределенная константа EPSQUADRUPLE. В Си эта константа называется LDBL\_EPSILON, в C++ — std::numeric\_limits<long double>::epsilon(). Приведенный в программе способ определения корректной ширины вывода рекомендуется применять в III задании курсового проекта, динамически конфигурируя таблицу под точность используемого вещественного типа конкретной ЭВМ.

Итак, при решении математической задачи, использующем числа типа **REAL**, по различным причинам получаются приближенные результаты. Иногда числовые данные, с которыми производятся вычисления, неточны, поэтому задача не может быть решена точно и возникает ошибка, которая называется *неустранимой погрешностью*. Кроме того, при выполнении операций вынужденно производят округления, необходимость которых вызывается представлением чисел с определенной точностью, и возникает ошибка, называемая *погрешностью округления*. Погрешность округления возникает и при переводе из одной системы счисления в другую [82]. Дело в том, что  $\hat{X}$  представляет собой *внешнее значение* представителя числа  $X$ . Имеется еще *внутреннее значение*  $\hat{X}$ , как правило, не совпадающее с внешним: только  $\hat{0} = \tilde{0} = 0$  и  $\hat{1} = \tilde{1} = 1$  при любой реализации. Например, конечная десятичная неправильная дробь 2.7 в двоичной системе счисления представляет собой бесконечную периодическую двоичную дробь 10.1011(0011). Без округления этой дроби ее помещение в машинное слово невозможно. Заметим, что эти проблемы отсутствуют лишь в системах с кратными основаниями:  $2 \Leftrightarrow 8, 16, 3 \Leftrightarrow 9$ . То есть привычная для человека десятичная система счисления вносит дополнительную неточность при переводе в неизбежную для ЭВМ двоичную систему счисления. С плавающей точкой  $\hat{X}$  представляется парой  $(\hat{m}, \hat{p})$  и  $\hat{X} = \hat{m} * b^{\hat{p}-\hat{l}+1}$ , и в качестве значения  $b$  обычно берется  $2^K$ , где  $K = 1, 2, 3, 4$  (и не больше); например в IBM/370 (mainframe)  $K = 4$ , а современный стандарт IEEE 754 предполагает двоичное основание порядка, сужая диапазон с целью увеличения точности. Различные машины одному и тому же  $\hat{X}$  могут ставить в соответствие различные  $\hat{X}$ .

Таким образом, все числа типа **REAL**, даже те, которые совпадают со своими представителями, мы должны рассматривать как приближенные. Это накладывает определенные ограничения на программирование: условия  $=$  и  $\neq$  могут не выполняться, так как приближенные значения могут сливаться (у них может быть один и тот же представитель), и наоборот. И это приводит к трудно обнаруживаемым ошибкам. Особенно опасно сравнение переменных с константами. На практике сравнение вещественных чисел на равенство (неравенство) заменяют проверкой принадлежности  $\varepsilon$ -окрестности.

```
var x, y, z : real;           var x, y, z : real;
begin                         begin
  if x = y then               if (x - y) < eps then
    ...                           ...
  else if x = z then          else if (x - z) < eps then
    ...                           ...
end;                           end;
```

```

var x, h : real;
begin
    h := (b - a) / 10;
    x := a;
    while x <> b do
        x := x + h;
    end;
var x, h : real;
begin
    h := (b - a) / 10;
    x := a;
    while x <= b do
        x := x + h;
    end;

```

Исследованием свойств вещественных чисел и оценкой полной погрешности вычислений подробно занимаются в курсе численных методов [102, 83]. См. также весьма интересное пособие [104]

К базовому множеству атрибутов вещественного типа относятся:

- изображения выделенных значений MINREAL и MAXREAL, определяющих диапазон представления, а также  $\perp$  и  $\top$ ; значение  $\top$  представляет числа из диапазона переполнения.

Заметим, что уже в стандарте IEEE 754 неопределенность(NaN, Not A Number) и переполнение ( $+/ - \infty$ ) имеют аппаратную поддержку. Там же специфицируются значащий нуль и ненормализованные значения:

s	e[52 : 62]	f[51 : 0]
63	62	52 51 0

Значения полей	Значение
$0 < e < 2047$	$(-1)^s \cdot 2^{-1023} \cdot 1.f$ (нормализованные числа)
$e = 0; f \neq 0$ (по меньшей мере один бит в $f$ не нуль)	$(-1)^s \cdot 2^{-1022} \cdot 0.f$ (ненормализованные числа)
$e = 0; f = 0$ (все биты в $f$ нули)	$(-1)^s \cdot 0.0$ (нуль со знаком)
$s = 0; e = 2047; f = 0$ (все биты в $f$ нули)	$+\infty$
$s = 1; e = 2047; f = 0$ (все биты в $f$ нули)	$-\infty$
$s = u; e = 2047; f \neq 0$ (по меньшей мере один бит в $f$ не нуль)	NaN (Not-a-Number)

- операции: сложение, вычитание, умножение и деление с плавающей точкой. Так как арифметические действия над числами с плавающей точкой являются по существу приближенными, а не точными, для обозначения соответствующих операций следовало бы использовать специальные символы  $\oplus, \ominus, \otimes, \oslash$ , чтобы отличать приближенные операции от точных. Однако обычно из контекста понятно, с какими данными производятся вычисления, и поэтому для обозначения этих операций используются те же символы, что и для соответствующих операций с данными целого типа. К операциям над данными вещественного типа относится и одноместный минус, обозначаемый так же, как и операция вычитания. Заметим, что сложение и вычитание может быть сведено к одному действию, если в случае вычитания переходить сначала к числу с противоположным знаком (выполняя операцию «одноместный минус»);

- отношения порядка;
- функции: ABS, SQRT, SIN, COS, EXP, LN, ARCTAN.

*Свойства операций над данными вещественного типа*

Пусть  $X, Y, Z \in \text{REAL}$  (символ  $\sim$  над значением писать здесь и в дальнейшем не будем, так как все остальные рассуждения относятся к представителям действительных чисел).

Операции над данными вещественного типа удовлетворяют следующим трем свойствам:

- коммутативности сложения и умножения, но только для двух слагаемых, т. е.

$$\begin{aligned} X + Y &= Y + X; \\ X * Y &= Y * X, \\ X + Y + Z &\neq Z + Y + X \neq Z + X + Y; \end{aligned}$$

но

- $(X - Y) + Y = X$ , только когда  $X \geq Y \geq 0$ ; при всех других соотношениях между  $X$  и  $Y$  равенство не гарантируется;
- симметричности операций относительно нуля:

$$\begin{aligned} X - Y &= X + (-Y) = -(Y - X), \\ (-X) * Y &\equiv X * (-Y) = -(X * Y), \\ \frac{(-X)}{Y} &= \frac{X}{(-Y)} = -\left(\frac{X}{Y}\right). \end{aligned}$$

Из этих свойств следует, что

- если  $Y \geq 0$ , то  $X + Y \geq X$ ;
- если  $X \geq Y$ , то  $X - Y \geq 0$ ;
- если  $X \geq 0$  и  $0 \leq Y \leq 1$ , то  $X * Y \leq X$ ;
- $X - X = 0$ ;
- $X + 0 = X$ ;
- $X * 0 = 0$ ;
- $X - 0 = X$ ;
- $X * 1 = \frac{X}{1} = X$ ;
- $\frac{X}{X} = 1$ .

Это очевидные свойства, которые выполняются.

Для данных вещественного типа, вообще говоря, *не выполняются*:

- правила ассоциативности для сложения и умножения:

$$(X + Y) + Z \neq X + (Y + Z); \\ (X * Y) * Z \neq X * (Y * Z).$$

Эти правила могут нарушаться не только из-за переполнения (получения результата вне диапазона представления). Нарушения могут быть вызваны округлениями и нормализацией. В частности, если в процессе вычислений производится вычитание двух близких чисел, то происходит значительная потеря точности (значимости);

- правило дистрибутивности

$$(X + Y) \cdot Z \neq (X \cdot Z) + (Y \cdot Z).$$

Рассмотренные свойства позволяют сформулировать рекомендации для практической организации вычислений над данными вещественного типа:

- Если необходимо произвести сложение-вычитание длинной последовательности чисел, то надо сначала производить действия с наименьшими числами.
- Надо по возможности избегать вычитания двух почти равных чисел; формулы, содержащие такое вычитание, часто можно преобразовать так, чтобы избежать подобной операции.
- Выражение вида  $a \cdot (b - c)$  можно записать в виде  $a \cdot b - a \cdot c$ , а выражение вида  $(b - c)/a$  в виде  $b/a - c/a$ . Если числа в разности почти равны друг другу, то следует производить вычитание до умножения или деления; при этом задача не будет осложнена дополнительными ошибками округления.
- В любом случае надо сводить к минимуму число необходимых арифметических операций над вещественными числами.

Замечание. Формула Тейлора сводит вычисление трансцендентных функций к алгебраическим (полиномам, ау, схема Горнера!). Однако этот простой способ не применяется на практике ввиду большой ресурсоёмкости и значительной погрешности.

### *Алгоритмы выполнения операций над данными вещественного типа*

Сложение и вычитание. Для общности будем считать, что основание системы счисления равно  $B$  и имеются два нормализованных числа  $X_1$  и  $X_2$ , причем  $X_1 > X_2$ :

$$X_1 = M_1 \cdot B^{P_1} \text{ и } X_2 = M_2 \cdot B^{P_2}.$$

Тогда

$$X_1 + X_2 = M_1 \cdot B^{P_1} + M_2 \cdot B^{P_2} = b^{P_1} \cdot (M_1 + M_2 \cdot B^{-(P_1-P_2)}).$$

Аналогично

$$X_1 - X_2 = M_1 \cdot B^{P_1} - M_2 \cdot B^{P_2} = b^{P_1} \cdot (M_1 - M_2 \cdot B^{-(P_1-P_2)}).$$

Пусть сначала оба числа неотрицательны. Тогда в соответствии с приведенными выше формулами выполняются следующие действия. Сначала происходит выравнивание

порядков, причем порядок меньшего числа приводится к порядку большего числа, т. е. порядок меньшего числа принимается равным порядку большего числа, а мантисса сдвигается на соответствующее число разрядов вправо (очевидно, что это число равно разности большего и меньшего порядков). Затем происходит сложение или вычитание мантисс как целых чисел, в результате чего получается мантисса результата. Порядок результата в соответствии с произведенным выравниванием порядков принимается равным порядку большего числа. Наконец, полученный результат округляется и, если нужно, нормализуется.

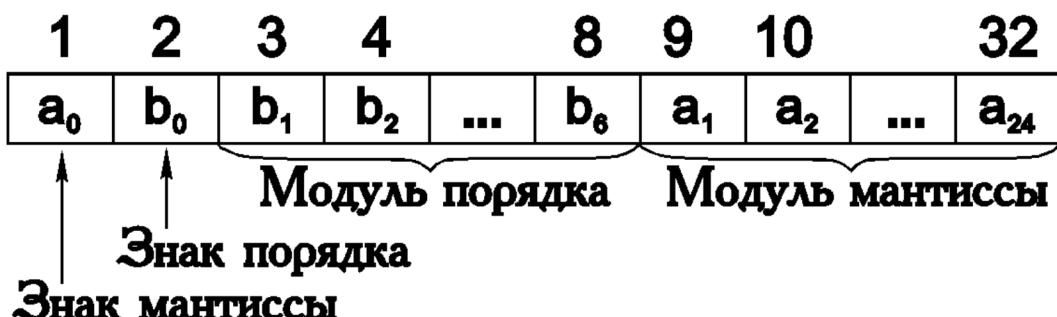
Округление результата может быть реализовано несколькими способами. Например, один из них заключается в том, что предварительный результат вычисляется с большим количеством разрядов, чем разрядная сетка чисел, над которыми производится операция. Даже если используется двоичный код, достаточно одного лишнего машинного (двоичного!) разряда. В этом случае округление заключается в прибавлении единицы к этому дополнительному младшему разряду предварительного результата. Если при выполнении действия в этом дополнительном разряде был получен нуль, то это не вызовет переноса единицы в основные разряды. Если же в дополнительном разряде при выполнении действия получена единица, то предварительный результат будет увеличен на единицу основного младшего разряда, что обусловлено переносом единицы, полученной при увеличении дополнительного разряда. После выполнения этих действий дополнительный разряд отбрасывается.

При нормализации полученного результата мантисса сдвигается влево до тех пор, пока в старшем разряде не окажется цифра, отличная от нуля, а порядок результата при этом уменьшается на число единиц, равное количеству разрядов, на которое произведен сдвиг мантиссы. Таким образом, при выполнении операций сложения и вычитания возникает проблема выравнивания порядков чисел, т.е. прежде всего должен быть проведен анализ знаков и величин порядков.

Для упрощения операций над порядками во многих машинах порядок всех чисел увеличивается на одно и то же число так, чтобы порядок наименьшего числа, представимого в машине, был неотрицательным, т. е. машинный порядок всех чисел  $P_x^m$  удовлетворял бы условию

$$P_x^m = P_x + K,$$

где  $K = |maxP_x| + 1$ . Например, для разрядной сетки



$|maxP_x| = 63$ , так что  $P_x^m = P_x + 64$ ,  $0 \leq P_x^m \leq 127$ . Таким образом, порядки всех чисел в машине становятся положительными, и анализ знака порядка становится не нужным.

Беззнаковый смещенный порядок называется *характеристикой*.

При проведении операций сложения и вычитания необходим также анализ знаков мантисс чисел, причем в зависимости от этого принципиально изменяется сама методика выполнения операции. Кроме того, операция вычитания в машине реализуется достаточно сложно, так как при этом часто возникает необходимость займа единиц в соседнем старшем разряде. Если же в соседнем старшем разряде этого сделать нельзя, необходим заем в более старшем разряде, и т. д. Если уменьшаемое меньше вычитаемого, этот процесс будет продолжаться до самого старшего разряда, а затем нужно вычитание делать заново, поменяв знак разности на противоположный, а вычитаемое и уменьшаемое — местами.

Для упрощения операции вычитания и сведения ее к операции сложения, так чтобы операция алгебраического сложения в машине выполнялась единообразно, как известно, применяют специальные способы представления отрицательных чисел.

Представление (изображение) чисел в соответствии с формулой

$$R_{np} = \begin{cases} R, & R \geq 0, \\ B - 1 + |R|, & R < 0 \end{cases}$$

называется *прямым кодом* числа  $R$ . Как видно из этой формулы, знак положительного числа кодируется нулем, а знак отрицательного числа кодируется максимальной цифрой системы счисления. Например

$$\begin{array}{lll} (R)_2 = 0.110101 & R_{np} = 0 & 110101 \\ (R)_2 = -101.1101 & R_{np} = 1 & 1011101 \\ (R)_{10} = -25.43 & R_{np} = 9 & 2543 \\ (R)_5 = -0.4321 & R_{np} = 4 & 4321 \end{array}$$

Из равенства  $X - Y = X + (B - Y) - B$  следует, что операцию вычитания  $Y$  из  $X$  можно заменить операцией сложения  $(B - Y)$  и  $X$  с последующим вычитанием из результата величины  $B$ . На этом основана работа с так называемым *дополнительным кодом* числа, который получается согласно следующей формуле:

$$R_{np} = \begin{cases} R, & R \geq 0, \\ B + R, & R < 0 \end{cases}$$

Из приведенных определений следует, что положительные числа в прямом и дополнительном коде совпадают. В прямом коде нуль имеет два представления: «положительный» и «отрицательный» нуль. В дополнительном коде нуль имеет только одно, «положительное» представление. Запись отрицательного числа в дополнительном коде получится вычитанием каждой цифры числа из максимальной цифры системы счисления, при этом младшая цифра вычитается из основания системы счисления. Знак отрицательного числа кодируется максимальной цифрой системы счисления. Например,

$$\begin{array}{lll} (R)_2 = 0.110101 & R = 0 & 110101 \\ (R)_2 = -101.1101 & R = 1 & 0100011 \\ (R)_{10} = -25.43 & R = 9 & 7457 \\ (R)_5 = -0.4321 & R = 4 & 0124 \end{array}$$

Отметим, что при представлении с плавающей точкой отдельно кодируются мантисса и порядок числа. При этом возможно представление мантисс и порядков в одном и том

же или разных кодах. Например, порядок числа может быть представлен в прямом, а мантисса — в дополнительном кодах и т. п.

Таким образом, используя дополнительный код, операцию алгебраического сложения можно свести к арифметическому сложению кодов чисел, распространяющемуся и на разряды знаков, которые рассматриваются как старшие разряды числа. Алгебраическое сложение дополнительных кодов, включая разряды знаков (если единица переноса из разряда знака отбрасывается) дает следующий результат: если алгебраическая сумма положительна, то результат получается в прямом коде; если алгебраическая сумма отрицательна, то результат получается в дополнительном коде. Сказанное выше относится лишь к случаю, когда модуль полученной суммы меньше единицы.

Однако при сложении двух чисел, модуль каждого из которых меньше единицы, может получиться сумма, модуль которой больше или равен единице (так называемое переполнение разрядной сетки мантиссы). Для обнаружения такого переполнения с учетом, что модуль суммы двух таких чисел всегда меньше двух, используют дополнительный разряд. Код, в котором имеется такой дополнительный разряд, называют *модифицированным*. В модифицированных кодах этот разряд используется для дублирования знака числа, т.е. знак «+» обозначается 00, а «-» — двумя максимальными цифрами системы счисления (для двоичной системы счисления — 11). Правила переноса из разряда знаков остаются прежними и зависят от кода представления числа. На переполнение же разрядной сетки, т.е. на то, что модуль суммы  $1 \leq |R + Q| < 2$ , указывает несовпадение цифр в знаковых разрядах. Например, для двоичной системы счисления комбинации 01 в этом случае соответствует положительное, а 10 — отрицательное число.

Примеры алгебраического сложения чисел в дополнительном коде:

1.  $B = 10$  при 4-разрядной сетке

$$(+7\ 123) + (-6\ 344) \rightarrow \text{в дополнительном коде:}$$

$$\begin{array}{r} + 00\ 7123 \\ + 00\ 3656 \\ \hline 1\ (00)\ 0779 \end{array} \rightarrow + 0779$$

↑ ↑  
| код знака “плюс”  
отбрасывается

То же в модифицированном дополнительном коде:

$$\begin{array}{r} + 00\ 7123 \\ + 00\ 3656 \\ \hline 1\ (00)\ 0779 \end{array} \rightarrow + 0779$$

↑ ↑  
| переполнения разрядной сетки нет  
| код дублированного знака “плюс”  
отбрасывается

2.  $B = 10$  при 4-разрядной сетке

$$(+7123) + (+6344)$$

В модифицированном дополнительном коде

$$\begin{array}{r} + 00\ 7123 \\ + 00\ 6344 \\ \hline (01)\ 3467 \end{array} \rightarrow + 0779$$

↑  
переполнение разрядной сетки положительного числа

3.  $B = 10$  при 4-разрядной сетке  
 $(-7123) + (-6344)$   
 В модифицированном дополнительном коде

$$\begin{array}{r}
 + \quad 99 \quad 2877 \\
 \quad 99 \quad 3656 \\
 \hline
 1 \quad (98) \quad 6533
 \end{array}$$

↑      ↑  
 | переполнение разрядной сетки положительного числа  
 отбрасывается

4.  $B = 10$  при 4-разрядной сетке  
 $(-1234) + (-2566) \rightarrow$  в дополнительном коде:

$$\begin{array}{r}
 + \quad 9 \quad 8766 \\
 \quad 9 \quad 7434 \\
 \hline
 1 \quad (9) \quad 6200 = -3800
 \end{array}$$

↑      ↑  
 | код знака “минус”  
 отбрасывается

То же в модифицированном дополнительном коде:

$$\begin{array}{r}
 + \quad 99 \quad 8766 \\
 \quad 99 \quad 7434 \\
 \hline
 1 \quad (99) \quad 6200 \rightarrow -3800
 \end{array}$$

↑      ↑  
 | переполнения разрядной сетки нет  
 | код дублированного знака “минус”  
 отбрасывается

Итак, в общем случае сложение и вычитание в машинах с плавающей точкой состоит из следующих этапов:

- (1) сравнение порядков чисел;
- (2) выравнивание порядков чисел;
- (3) перевод мантисс слагаемых в дополнительный модифицированный код;
- (4) сложение мантисс;
- (5) исправление нормализации вправо или влево;
- (6) перевод мантиссы результата в прямой код;
- (7) выдача результата с учетом порядка.

Умножение и деление. Пусть опять имеются два нормализованных числа

$$X_1 = M_1 \cdot B^{P_1} \text{ и } X_2 = M_2 \cdot B^{P_2}.$$

Тогда

$$\begin{aligned}
 X_1 \cdot X_2 &= M_1 \cdot B^{P_1} \cdot M_2 \cdot B^{P_2} = (M_1 \cdot M_2) B^{P_1+P_2}; \\
 X_1/X_2 &= (M_1 \cdot B^{P_1}) / (M_2 \cdot B^{P_2}) = (M_1/M_2) B^{P_1-P_2}.
 \end{aligned}$$

В соответствии с этими соотношениями при умножении порядки чисел складываются, а мантиссы перемножаются. При делении порядок частного определяется разностью порядков делимого и делителя, а мантисса частного равна частному от деления мантисс делимого и делителя. Знак мантиссы произведения или частного определяется следующим

образом. Так как знак мантиссы «минус» кодируется в машине единицей, а «плюс» — нулем, то можно ввести функцию знакового разряда

$$s(X) = \begin{cases} 0, & \text{знак «плюс»,} \\ 1, & \text{знак «минус».} \end{cases}$$

Тогда имеем

$$\begin{aligned} s(X_1 \cdot X_2) &= s(M_1 \cdot M_2) = s(M_1) + s(M_2); \\ s(X_1/X_2) &= s(M_1/M_2) = s(M_1) + s(M_2), \end{aligned}$$

причем единица переноса, если она появляется, пропадает. Таким образом, правило вычисления знака арифметизировано.

При выполнении операций умножения и деления особую роль играет операция сдвига чисел. Сдвиг числа в системе счисления с основанием  $B$  на  $n$  позиций эквивалентен умножению этого числа на  $B^n$ , где  $n$  — целое. Если  $n$  — положительное, то сдвиг осуществляется на  $n$  разрядов влево; если  $n$  — отрицательное, то сдвиг осуществляется на  $n$  разрядов вправо. При сдвиге чисел вправо или влево надо сохранять содержимое знакового разряда. При сдвиге положительного числа влево пропадают цифры разрядов, расположенных перед знаковым разрядом, а в появляющихся справа разрядах располагаются нули. Если, например, мантисса со знаком записана в виде  $0a_1a_2a_3\dots a_x$ , то после сдвига на один разряд влево получится число  $0a_2a_3\dots a_x0$ . При сдвиге вправо пропадают разряды, начиная с самого правого, а в появляющиеся слева разряды заносятся нули; при сдвиге вправо на один разряд той же мантиссы получиться число  $00a_1a_2a_3\dots a_{x-1}$ .

Сдвиг отрицательных чисел, представленных в дополнительном коде, выполняется по особым правилам, определяющим модифицированный сдвиг. При таком сдвиге дополнительный код числа  $X$  преобразуется в дополнительный код числа  $X \cdot B^n$ . При сдвиге числа вправо на  $n$  разрядов ( $n < 0$ ) во все освобождающиеся слева разряды должна быть занесена старшая цифра основания системы счисления (т.е.  $B - 1$ ), т.е. цифра, совпадающая со знаком числа; при сдвиге числа влево освобождающиеся справа разряды заполняются нулями.

В зависимости от организации умножения результат может иметь код одинарной или двойной длины. Как правило, код двойной длины приходится округлять до одинарного.

Итак, при умножении чисел с плавающей точкой выполняются следующие действия:

- 1) по приведенной выше формуле определяется знак результата (произведения);
- 2) алгебраическим сложением порядков сомножителей вычисляется порядок произведения;
- 3) последовательным выполнением операций сдвига и арифметического сложения производится перемножение мантисс;
- 4) результат округляется;
- 5) если надо, полученный результат нормализуется.

Микрокоманда арифметического сдвига является частью микропрограммы вещественного умножения, выступающей на традиционном машинном уровне уже как единая машинная команда.

Деление является одной из самых трудоемких операций, и на ее выполнение в машинах затрачивается гораздо больше ресурсов, чем на выполнение других арифметических операций. Долгое время в вычислительных машинах операция деления реализовывалась путем последовательности операций вычитания и сдвига. Например, известен метод *деления с восстановлением остатка*. Он заключается в следующем:

- 1) в соответствии с приведенной формулой определяется знак частного;
- 2) вычитанием порядка делителя из порядка делимого находится порядок частного;
- 3) мантисса делимого делится на мантиссу делителя путем вычитания делителя из полученного положительного остатка предыдущей разности, сдвинутого на один разряд влево. Если остаток отрицателен, то восстанавливается прежний остаток и сдвигается на один разряд влево. При получении отрицательного остатка в частное заносится нуль, в противном случае — единица. При определении первой цифры частного за остаток принимается все делимое;
- 4) результат округляется;
- 5) если надо, полученный результат нормализуется.

В некоторых случаях этот порядок выполнения действий может быть нарушен. Если, например, делимое равно нулю, то деление как таковое не производится, а в качестве частного принимается нуль. То же самое может происходить при умножении, если один из сомножителей равен нулю. Порядок действий может быть изменен и в том случае, если в результате вычитания (деления) или сложения (умножения) порядков полученный порядок окажется меньше минимального отрицательного. В этом случае сразу же будет сформирован нулевой результат без выполнения каких либо дальнейших действий. Если делитель оказывается равным нулю, то деление не производится и происходит ОТКАЗ (автоматическая остановка машины — АВОСТ).

В современных процессорах реализованы малотактные операции деления с одновременной обработки всех разрядов числа.

Паскаль-программы интерпретации изображений вводимых и выводимых вещественных чисел приведены в п. 13 пособия Н.Вирта [8].

## Лекция 21

### 3.2.5 Тип литерный

Константы литературного типа — самоопределённые термы — заключаются в апострофы или кавычки: '3', 'a'. Множество значений определяется конкретной кодировкой: ASCII, КОИ-8 и т.д. Эта кодировка задаёт порядковый номер литеры, определяемой функцией преобразования типа `ord`, связывающей литературный тип с поддиапазоном целого (обычно  $[0..255]$ ), и предопределяет упорядоченность множества значений так, что имеет смысл понятия следующей и предыдущей литеры (`succ` и `pred` соответственно). Одновременно кодовая таблица задаёт и обратное соответствие литер и внутренних кодов (в Паскале это функция `chr`). Таким образом, для всякой литеры с имеет место соотношение `c = chr(ord(c))` и наоборот (для малых положительных целых, обычно в диапазоне  $[0..255]$ ) `i = ord(chr(i))`. Кроме того, `c = succ(pred(c)) = pred(succ(c))`. Для языка Си все условности отброшены и транслятор осуществляет неявное автоматическое преобразование `int ⇔ char`: `i = c` или `c = i`.

Тип литературный имеет минимальный набор операций и отношений, включающий в себя присваивание `:=` и стандартный набор отношений.

В языках Паскаль и Си имеется встроенный литературный тип.

```
var a, b, c : char;
i : integer;
```

```
char a = 'a', b = 'b', c;
int i;
```

<pre>a := 'a'; b := 'b'; <b>if</b> a &gt; b <b>then</b>     c := a; <b>else</b>     c := b; i := ord(c);</pre>	<pre>c = (a &gt; b) ? a : b; i = c;</pre>
--	---

Литерный тип используется для ввода-вывода и обработки текстовых данных (в том числе и для изображений значений других типов словами — цепочками знаков). По этой причине в языке Паскаль литерный тип является «более элементарным», чем другие типы. Основные входной и выходной файлы Паскаль-программы (`input` и `output`) — литерного типа. Имеется соответствующий предопределенный файловый тип `text`.

В настоящее время ввод-вывод утрачивает свою литерную ориентацию, приобретая графический и мультимедийный характер. Однако литерный тип весьма важен как простой и удобный стандарт.

Часто используются одномерные массивы литер, называемые **строками**. В расширениях Паскаля строковый тип снабжён богатым набором операций, отношений и функций, образуя своеобразную алгебру и даже геометрию. В языке Си строкового как такового типа нет. Вместо строк используется массив литер. Признаком конца строки является элемент с кодом 0 (*null-terminating string*).

Лента машины Тьюринга является строкой [84] и на любом языке программирования можно написать нечто подобное этому.

```
label 17, 43;
```

```
type ttape = array[1..MAXINT] of char;

var tape : ttape;
    wc : integer;

begin
    ...
    {(17, g, j, 17)}
17: if(tape[wc] = 'g') then begin
    tape[wc] := 'j';
    goto 17
end
    {(17, j, >, 43)}
    else if(tape[wc] = 'j') then begin
        wc := succ(wc);
        goto 43
    end;
    ...
end.
```

Это можно рассматривать как еще одно доказательство тезиса Тьюринга-Черча. Кроме того, это прекрасная иллюстрация операционной семантики состояний-переходов,

свойственной МТ.

Сравним строковые средства Паскаля и Си.

```
var str : packed array[1..80] of char;  
  
{ Копирование строки только с packed!  
}  
str := 'Hello, world!<67 пробелов>;  
  
{ Поиск и замена литеры }  
i := 1;  
found := false;  
while (i <= 80) and not found do  
  if str[i] = 'w' then begin  
    str[i] = 'W';  
    found := true;  
  end;  
  
{ Перевод в нижний регистр }  
for i := 1 to 80 do  
  if str[i] >= 'A' and str[i] <= 'Z'  
  then  
    str[i] := chr(ord(str[i]) + ord('A')  
      - ord('a'));
```

```
// Стока из 80 литер!  
char str[81];  
  
// Копирование строки в строку  
strcpy(str, "Hello, world");  
  
// Поиск и замена литеры  
str[strchr(str, 'w')] = 'W';  
  
// Вычисление длины строки  
strlen(str);  
  
// Конкатенация строк  
strcat(str, "!");  
  
// Перевод в нижний регистр  
strlwr(str);
```

Из примера видно, что в Паскале представление строк примитивное, не позволяющее работать со строками переменной длины, в которых должен быть либо терминатор (символ завершения строки), либо дескриптор строки с указанием ее длины и ссылкой на ее содержимое. Видимо, автор Паскаля, ограничиваясь строками фиксированной длины, заботился об эффективной статической компиляции Паскаль-программ, поскольку поддержка строк переменной длины добавляет элемент интерпретативности.

В языке Си строковый тип реализован интерпретативным образом как набор функций стандартной библиотеки языка.

### 3.2.6 Согласование типов

Подобно многотиповой машине фон Неймана, язык программирования фон Неймановского типа допускает согласованное использование различных типов данных в вычислениях. Выражения со смешанными типами явно (с помощью функций согласования типов) или неявно (функции преобразования осуществляются компилятором или языковой средой по умолчанию) приобретают вполне определённый однозначный смысл.

Помимо соответствия литературного и целого типов, заслуживает внимания согласование целого и вещественных типов, как близких с математической точки зрения (числовых), хотя весьма различных по внутримашинному представлению значений. Оно может осуществляться как округлением (round()), так и отбрасыванием дробной части (trunc()). Обратное преобразование (из целого в вещественный) производится по умолчанию

в случае необходимости приведения смешанного выражения к более *сложному* вещественному типу. Преобразование целый-вещественный часто реализуется интерпретативно, и, следовательно, достаточно трудоёмко. В современных процессорах это преобразование возлагается на аппаратуру (математический сопроцессор).

За трёхтиповым **примером** можно обратиться к программе печати графиков Н. Вирта ([9], с. 35–36).

В расширениях языка Паскаль вопрос согласования типов решён более систематически, так, что имя любого типа может использоваться как функция преобразования к нему: `i := integer(r)`. Кроме того, возможна трактовка значения одного типа в любом другом типе с соответствующей сменой интерпретации внутреннего представления (низкоуровневое средство). Это возможно даже в стандартном Паскале (записи с вариантной частью).

### 3.2.7 Небазовые типы данных

Поскольку предусмотреть в языке программирования встроенных базовых типов на все случаи жизни невозможно, обычно предоставляют возможность более или менее тривиального конструирования новых типов данных на основе базовых.

#### 3.2.7.1 Отрезок (диапазон)

Можно было бы объявить отрезок типа, например, следующим образом:

```
type diap = (0.0 .. 2.0; real; real);
```

Здесь в качестве множества значений берётся отрезок вещественного типа, операции и отношения также заимствуются из вещественного типа.

#### 3.2.7.2 Перечисление

В программе управления уличным движением удобно объявить

```
type StreetLight = ((Red, Yellow, Green) ; := ; integer);
```

## Лекция 22

### 3.2.8 Понятие о структурном типе данных

До сих пор значения типов, даже и изображаемые словами, нами считались **атомарными**, не имеющими структуры, хотя очевидно, что во многих задачах удобно иметь дело со значениями, имеющими ту или иную структуру (векторами, матрицами, последовательностями, деревьями, графиками).

Рассмотрим типы данных со структурными значениями. **Структурные значения** — это упорядоченные систематически организованные совокупности других (быть может, тоже структурных) значений, рассматриваемое как единое целое. Компоненты структурных значений также называется его **полями** или **элементами**.

Типичным примером структурного типа является комплексный тип, состоящий из двух вещественных полей: вещественной и мнимой частей.

Структурный тип характеризуется типом (или типами) своих компонент, и способом их организации (методом структурирования). Существует несколько методов структурирования, отличающихся способом доступа к компонентам и способом обозначения этих компонент: регулярный метод с индексированным или секвенциональным доступом к элементам, комбинированный метод с квалифицированным доступом к полям и др. **Регулярный** структурный тип содержит компоненты одного типа, называемого **базовым**. Напротив, **комбинированный** структурный тип содержит компоненты различных типов. Различают **индексированный** (компоненты структурного типа идентифицируются т. н. **индексом** (некоторым атомарным значением перечислимого типа прямо адресующим элемент), **квалифицированный** (компоненты идентифицируются именем) и **секвенциальный** (последовательный) методы доступа.

Переменные и константы структурного типа называются **структурными**. Заметим, что структурные константы отсутствуют в стандарте языка Паскаль, так что для этой цели используются структурные переменные, компоненты которых предварительно заполняются требуемыми значениями.

доступ\структура	регулярная	комбинированная
индексированный	массив	
секвенциональный	файл	
квалифицированный		запись

### 3.2.9 Тип массив

Тип массив — это регулярный структурный тип с индексированным методом доступа. Регулярность структуры означает, во-первых, одинаковый тип всех компонент и, во-вторых, использование в качестве индексной структуры декартова произведения (быть может, композированного, при определении массива через подмассивы) отрезков перечислимых типов (решётчатого многомерного прямоугольного параллелепипеда).

```
const MAX = 20;
type index = 1..MAX;
var data : array[index, index] of
    integer;
```

```
const int MAX = 20;
int data[MAX][MAX];
```

Для массивов *в точности одного и того же типа* определена операция присваивания и отношение равенства ( $A := B$  и  $A = B$  соответственно). Кроме того, могут быть тем или иным образом использованы операции над компонентами, например, целые и вещественные компоненты массивов можно умножать или располагать в порядке возрастания (убывания) с помощью атрибутов типов этих компонент.

Ввиду скалярности машины фон Неймана обработка массивов осуществляется покомпонентно, как правило с помощью циклов с параметром (**for**), с подстановкой параметра цикла в индексное выражение. Кроме того, что сложные индексные выражения позволяют осуществлять различные обходы или переборы компонент, параметр цикла как своеобразное время цикла в индексном выражении становится указателем пространства, т. е. цикл по времени становится циклом по пространству. Также покомпонентно требуется осуществлять ввод-вывод массивов из текстовых файлов, содержащих слова — изображения компонент в порядке, определяемом индексными наборами, обычно генерируемыми

циклами с параметром (**for**).

```
program p(input, output);

const N = 10;
var data : array[1..N,1..N] of integer;
    i, j : integer;

begin
{ Ввод }
  for i := 1 to N do
    for j := 1 to N do
      read(data[i, j]);
{ Выход }
  for i := 1 to N do begin
    for j := 1 to N do
      write(data[i, j] : 0, ',');
    writeln
  end
end.
```

```
#include <iostream>

const int N = 10;
int data[N][N];

int main(void)
{
// Ввод
  for(int i = 0; i < N; i++)
    for(int j = 0; j < N; j++)
      std::cin >> data[i][j];
// Выход
  for(i = 0; i < N; i++)
  {
    for(j = 0; j < N; j++)
      std::cout << data[i][j] << ",";
    std::cout << "\n";
  }
  return 0;
}
```

### 3.2.10 Понятие о записях

Тип записи — это комбинированный структурный тип с квалифицированным методом доступа. Комбинированность означает, что поля записи, вообще говоря, имеют различные типы. Квалифицированный доступ, в отличие от индексированного, негибок и невычислим, так как требует явного указания статически определённых до компиляции имён полей. Однако, это компенсируется большой семантической нагрузкой мнемоничных имён полей. С учётом возможной вложенности определений комбинированных типов, легко реализуются иерархические нереляционные структуры, лучше отражающие реальные соотношения объектов в обществе и в технике.

С помощью оператора-квалификатора **with** поля некоторой записи в ограниченном контексте приобретают форму простых переменных.

```
type date = record
  day : integer;
  month : integer;
  year : integer;
end;

type person = record
  name : packed array[1..20] of char;
  birthday : date;
```

```
struct date
{
  int day;
  int month;
  int year;
};

struct person
{
```

```

end;

var JV : person;
begin
  with JV do begin
    name := 'Ja}ues_Villeneuve';
    with birthday do begin
      day := 9;
      month := 4;
      year := 1971;
    end;
  end;
end.

```

```

char name[21];
date birthday;
};

person JV = {"Ja}ues_Villeneuve", {9, 4,
1971}};


```

Заметим, что в Паскале нет структурных констант, за исключением строковых. Однако строки-константы должны быть в точности той же длины, что и инициализируемые строки-переменные. Поэтому в данном примере полезная часть строки дополнена пробелами справа.

Комбинированные структуры Паскаля и Си (записи и объединения) допускают *вариантную часть*, позволяющая задавать несколько *совершенно различных* вариантов одного и того же поля.

```
type t = (bb, cc, ii , rr);
```

```

type variant = record
  case t of
    bb : (b : boolean);
    cc : (c : char);
    ii : (i : integer);
    rr : (r : real);
end;
```

```

var v : variant;
begin
  v.i := ord('9');
  writeln(v.b);
  writeln(v.c);
  writeln(v.i);
  writeln(v.r);
end.
```

Все варианты такого поля записи размещаются в одной и той же области памяти, длина которой равна максимальной длине варианта. Типовая трактовка этой области памяти зависит от значения селектора. Если занести в вариантное поле данного примера литеру «9» (порядковый номер в ASCII 57) в системе Compaq Pascal будет напечатано следующее:

TRUE

0.00000e+00

То есть литерная девятка не является ни целой, ни вещественной девятками. Она трактуется как «Истина», поскольку младший бит соответствующего машинного слова равен 1.

### 3.2.11 Понятие о файлах

Самый маленький файл всегда больше самого большого массива!

Зайцев В. Е.

Структура файла является обобщением понятия последовательности. Поэтому файлы прямого доступа следует считать «массивами на диске». Компоненты файла должны быть одного типа, и они доступны только путём последовательного прочтения. Поскольку файлы ввиду их потенциально большого размера размещаются на устройствах внешней памяти, в каждый момент времени доступна лишь текущая компонента файла, а другие компоненты могут быть получены лишь последовательным прочтением компонент файла вперёд, или, после перемотки, с начала. Движения «назад» нет ввиду инерционности электромеханических устройств.

Характерным аппаратурным аналогом последовательного файла является магнитная лента. В настоящее время на компьютерах устройства памяти на магнитных лентах существуют, как правило, в кассетном исполнении и практически вытеснены из массового употребления CD/DVD-устройствами. Однако в быту аудио- и видеокассеты все еще распространены и поэтому далеко ходить за соответствующими примерами нам не придется. Так, для того чтобы просмотреть второй тайм футбольного матча или середину видеофильма, необходимо просмотреть (или перемотать) первую половину кассеты.

Ввиду потенциальной бесконечности размера файлов, при их обработке следует использовать специфические последовательные методы, например, применять вспомогательные файлы. Это неизбежное зло не очень велико. Не следует пытаться заносить весь файл в память, помня приведенное в эпиграфе изречение. Все такие программы не будут работать на больших файлах. Таким образом, разрешается держать в памяти одну или несколько (небольшое «конечное» число) компонент файла. Запрещается помещать в память половину, четверть или любую другую долю от всегда «бесконечно» большого файла. Такой же запрет можно наложить на и хранение в памяти строки текстового файла т. к. её длина также не ограничена.

Файлы принято подразделять на внешние и внутренние, текстовые и нетекстовые, входные и выходные.

**Внешние** файлы обычно перечисляются в заголовке программы. Они существуют до начала работы программы и/или сохраняются после окончания её работы. То есть отрезок времени их жизни шире промежутка времени работы программы. Операционная система обычно предоставляет возможность сопоставить идеальным файлам программы на языке высокого уровня реальные файлы операционной системы. Стандарт Паскаля фактически предполагает, что файл имеет простое имя и находится в текущей директории, из которой запущена программа. Поэтому процедуры инициализации и установки на начало файла **rewrite** и **reset** в стандарте Паскаля не содержат средств связи файлов Паскаль-программы с конкретными файлами используемой ОС. В различных расширениях Паскаля существуют процедуры сопоставления файлам Паскаля произвольных файлов

ОС: **reset** со вторым параметром, **open**, **assign**, — полные имена которых задаются как строковые константы или даже переменные, и это соответствие является динамическим.

**Внутренние файлы**, также как и внешние, описываются в программе как файловые переменные. Их время жизни совпадает со временем работы программы или, в случае динамического сопоставления физических файлов, короче его. Уникальные имена физических файлов операционной системы, реализующих внутренние файлы, генерируются операционной системой по требованию среды языка. Внутренние файлы используются в качестве рабочих файлов — временной памяти практически неограниченного размера.

**Текстовые файлы** представляют собой распространённый вид файлов, используемых для ввода-вывода или для хранения данных в виде, непосредственно пригодном для ввода-вывода (т. е. в форме внешних изображений, а не как внутримашинные значения). Характерным для текстовых файлов является также интерпретация символа конца строки (EOL), разбивающая файл на строки. В стандарте Паскаля доступен только сам факт обнаружения конца строки вне зависимости от его физической реализации, а не соответствующая литера используемого кода. Часто текстовые файлы бывают только **входными и выходными**, как, например, INPUT и OUTPUT в Паскале. В языке Паскаль для машино-системонезависимого представления текстовых файлов и их строк существуют предикаты EOF и EOLN. Они позволяют Паскаль-программисту не зависеть от конкретных представлений этих служебных кодов. Таким образом, считывание литер до конца строки текстового файла должно выглядеть так:

```
while not eoln do  
  read(ch);
```

Если вместо eoln написать одно из конкретных представлений ch = chr(13), то получится зависимая от кодировки Паскаль-программа, что противоречит идеологии языка высокого уровня.

**Нетекстовые файлы** не предназначены для ввода-вывода и хранят данные непосредственно во внутримашинном представлении, которое компактно, экономит не только место на устройствах внешней памяти, но и время их передачи в оперативную память, причём экономия достигается также и потому, что не требуется перевода значений в текстовые изображения и обратно, как это делается, например, при вводе/выводе **real** и **integer** из/в файлы(ов) INPUT/OUTPUT. Эта экономия весьма значительная, поскольку для преобразования изображения в значение или значения в изображение требуется его полиномиальная интерпретация (деинтерпретация), которая даже при экономной схеме Горнера имеет линейную сложность: каждое число в позиционной системе счисления есть значение многочлена с коэффициентами, равными его цифрам, вычисленного в точке, равной основанию системы счисления. Поэтому во многих компьютерах недавнего прошлого были специальные команды десятичной арифметики для выполнения операций над изображениями десятичных чисел (**BCD** — binary-coded decimals — *двоично-кодированные десятичные*).

Теперь, после того как мы ознакомились с описанием и употреблением массивов, мы можем написать соответствующую Паскаль-программу вычисления значения числа по его цифрам в произвольной системе счисления.

```
program Horner(input,output);  
{ Вычисление значения многочлена по схеме Горнера, представленного в ЭВМ  
  вектором коэффициентов. }
```

$p$  – коэффициенты многочлена,  $n$  – его степень }

```
var p : array [0..10] of real;
n, i : integer;
x, s : real;
cont : boolean;
begin
repeat
  readln(n);
  if (n < 0) or (n > 10) then
    writeln('Степень многочлена должна быть от 0 до 10');
until(n >= 0) and (n <= 10);
for i := 0 to n do
  read(p[i]);
readln;
writeln('Ведите x');
readln(x);
write('P(x)=');
cont := false;
s := 0.0;
{ Схема Горнера:  $p_n = ((\dots(a_n x + a_{n-1})x + \dots)x + a_1)x + a_0$  }
for i := n downto 0 do begin
  s := s * x + p[i];
  if p[i] <> 0.0 then begin { печать }
    if cont then
      write('+');
    write(p[i] : 1 : 3);
    if(i <> 0) then
      write('*x^', i : 1);
    cont := true;
  end;
end;
writeln;
writeln('P(', x : 1 : 3, ')=', s : 1 : 3);
end.
```

На самом деле при использовании схемы Горнера для полиномиальной интерпретации изображения числа, цифробуквы которого поступают из текстового файла, массив для хранения значения той части числа, которая уже считана, не требуется: достаточно одной переменной типа **integer** для накопления частичных постепенно вычисляемых значений вводимого числа. ... `read(c); i := i * 10 + (ord(c) - ord('0'));` ... Полученное таким способом числовое значение во внутреннем представлении можно выводить в десятичной системе счисления. Необходимая обратная интерпретация аккумулированного числа может быть выполнена автоматически при выводе полиморфной процедурой **writeln**. Заметим, что для больших чисел обычной целой переменной может оказаться недостаточно.

## Лекция 23

### 3.3 Блочная структура программ

Ранее нами были введены основные конструкции языка программирования (композиции, ветвления и цикла), предназначенные для описания внутренней логики программы. Этих конструкций достаточно для описания любых программ (теорема Бойма-Джакопини-Миллса), причём программы при исходящей разработке получаются с чётко выраженной структурой последовательного выполнения инструкций, легко читаются, тестируются и отлаживаются. Таким образом, использование минимального набора инструкций позволяет решить проблему эффективной организации программ **по управлению**.

Не менее важной проблемой является проблема организации программ **по данным**. Для организации программ по данным ещё в ранних языках программирования (АЛГОЛ-60) была введена специальная конструкция — блок. Блок является простейшей формой программной единицы, представляющей собой обособленный фрагмент программы со своим локальным контекстом. Типичный пример объектов локального контекста — параметр цикла **for** или временная переменная для обмена значениями двух переменных. Иерархическая вложенная система блоков позволяет организовать проектирование программы по данным как иерархический процесс детализации её функций. Каждая функция детализируется в виде блока с указанием некоторых характерных локальных данных. Если блок оказывается длинным и сложным, то в нём организуются дополнительные блоки.

Определим блок как разновидность *составной* инструкции языка программирования, обозначаемую открывающей скобкой (**begin**, {}) и закрывающей скобкой (**end**, { }) имеющую следующую структуру:

**begin** // Начало блока языка АЛГОЛ–60

<последовательность описаний>

<последовательность инструкций>

**end** // Конец блока

Известный нам составной оператор Паскаля блоком не является, так как не содержит описаний и не имеет локальных данных. Как и составной оператор, блок может быть вложенным и входить в другой, охватывающий блок. Во вложенном блоке доступны объекты охватывающего блока, если только они не экранированы омонимичными (одноименными) локальными переменными. Экранирование означает приоритет локальных объектов над глобальными и является средством разрешения конфликта имен. В то же время локальные переменные и константы блока недоступны в охватывающих блоках. В языках с развитой блочной и модульной структурой существуют средства явного экспорта/импорта глобальных и локальных объектов. Обычно программы, процедуры и функции представляют собой блоки, т. к. имеют описания локальных объектов.

Локальные объекты позволяют не заботиться об уникальности имен при написании больших программ. Более того, они защищены от несанкционированного использования или от непреднамеренной порчи в большей части программы тем, что просто не видны оттуда. Локализация области действия переменных и констант позволяет не просто разгрузить глобальный контекст от второстепенных деталей, но и оптимизируют использование памяти.

При использовании блочной структуры управление памятью локальных данных блоков

происходит следующим образом: для каждого из блоков на время его выполнения выделяется область памяти для размещения локальных переменных и констант. Глобальные переменные и константы при этом уже размещены в области памяти охватывающего блока. После завершения выполнения блока память, отведенная под локальные переменные и константы, освобождается и может быть повторно использована, что сокращает общую потребность программы в памяти, так что она значительно меньше суммарной длины всех переменных и констант. В каждый момент времени занимается память только под данные, необходимые текущему блоку и всем его прародителям. Это можно проиллюстрировать использованием памяти при выполнении программы со следующей блочной структурой:

```
// Глобальный контекст описаний

const int MAX = 100; // Глобальная константа, доступна во всех блоках

int main(void) // Начало блока 0
{
    int B[MAX];
    // ...
    for(int i = 0; i < MAX; i++) // Блок 1 – тело цикла for, вложенный в блок 0, i
        – локальный параметр цикла for для блока 1
    for(int j = 0; j < MAX; j++) // Блок 2 – цикл for, вложенный в цикл for; j –
        его локальный параметр
    if(B[i] > B[j]) // Блок 3 – проверка
    {
        int t = array[i]; // Локальная переменная для обмена значениями
        array[i] = array[j];
        array[j] = t;
    } // Конец блоков 3, 2 и 1
    return 0;
} // Конец блока 0
```

Обратите внимание, что вложенные блоки заканчиваются в порядке, текстуально обратном их описанию.

При выходе из блока память, занятая локальными переменными, освобождается и все хранящиеся в них значения теряются. В некоторых языках программирования возможно сохранение локальных переменных. Так в языке Алгол-60 описателем **own** помечаются локальные переменные, значения которых надо сохранять между обращениями к блоку. В языке Фортран локальные переменные всегда статичны и их значения сохраняются между вызовами. В языке Си существуют два описателя для этих классов памяти: **static** и **auto**. Всякая переменная в языке Си по умолчанию имеет статус автоматической, поэтому слово **auto** используется крайне редко.

Иерархическая структура блоков программы такова изображена на рис. 3.3

Вложенные области данных удобно размещать в стековой памяти. Это еще одна лента машины Тьюринга.

В случае, если блок содержит обращения к себе (рекурсия), то возможны такие ситуации:

БЛОК 0 БЛОК 1 БЛОК 1 . . . БЛОК 1

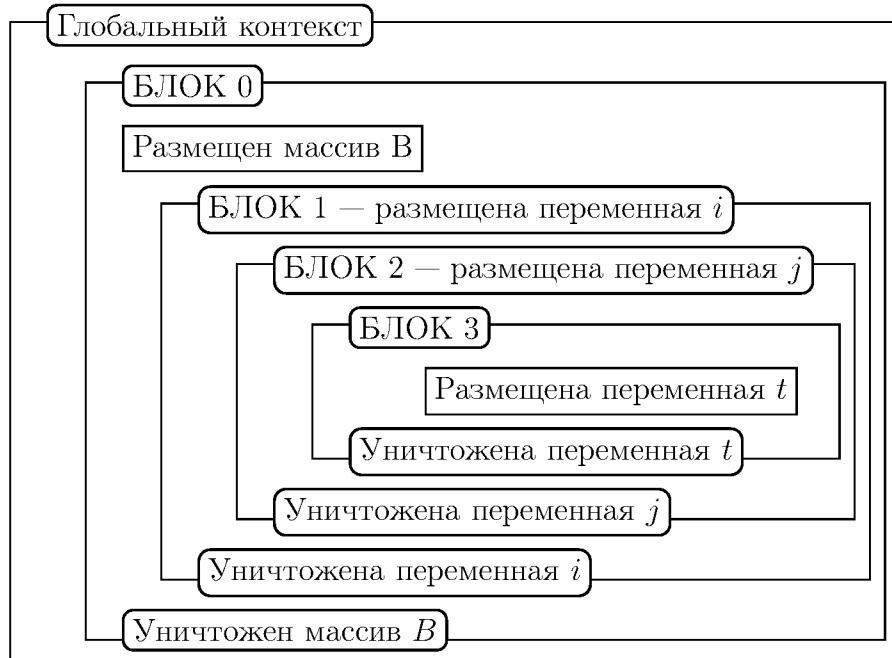


Рис. 3.3: Блочная структура программы

Без выделения памяти для локальных объектов в блоках и процедурах их рекурсивное использование невозможно: каждый вызов процедуры требует нового набора омонимичных локальных переменных.

Для того, чтобы блок стал элементом построения программы, он может быть помечен и выполняется путем перехода на эту метку.

Поскольку в Паскале нет полноценной блочной структуры, приведем пример на Си:

```

int x = 0; // Глобальная переменная

int main(void) // Блок 0
{
    x = 1; // Изменение глобальной x
    { // Блок 1
        int x = 1; // Локальная x
        x = ::x; /* Присвоение локальной переменной значения омонимичной глобальной.
                     Обращение к глобальной x квалифицируется символом :: */
    }
    { // Блок 2
        int y;
        { // Блок 3
            int y = 1; // Локальная переменная
            ::y = y; // Копирование значения переменной y блока 3 в переменную внешнего
                      // блока 2
        }
    }
}
  
```

```
// Отсюда у не виден!
}
```

Где начало того конца, которым кончается  
начало?

Козьма Прутков

Структура вызовов этой программы имеет вид

БЛОК 0	начало выполнения блока 0
БЛОК 0 БЛОК 1	начало выполнения блока 1
БЛОК 0	конец выполнения блока 1
БЛОК 0 БЛОК 2	начало выполнения блока 2
БЛОК 0 БЛОК 2 БЛОК 3	начало выполнения блока 3
БЛОК 0 БЛОК 2	конец выполнения блока 3
БЛОК 0	конец выполнения блока 2

Соответствующий пример для языка Паскаль возможен только для вложенных процедур и будет приведен позже.

Интересно, что структура текста в издательской системе ТЕХ также блочная. В качестве скобок используются { и }.

## Лекция 24

### 3.4 Процедуры и функции

Рассмотрим более совершенные средства организации программ: функции и процедуры. Их значение в том, что они, как и блоки, являются средством представления программ в виде логически связанных компонент (подпрограмм), выделение которых отражает связь программы по данным и придает ей четко выраженную структуру, легко контролируемую, в том числе и автоматически. Кроме того, организация в процедуры часто встречающихся фрагментов программ сокращает размер программы и снижает вероятность ошибок, и, что самое главное, фактически вводит новые операторы в язык программирования, не требуя никаких изменений входного языка и системы программирования. Каждый осмысленный набор процедур и функций фактически представляет собой *интерпретативное расширение языка программирования*. Во случаях, когда процедуры и функции набора реализуют операции и отношения некоторого типа данных их можно считать программной реализацией этого типа. Такие типы данных, не имеющие прямой аппаратной или программной поддержки, принято называть *абстрактными*. Процедуры являются одной из первых форм «товарного программирования» и представляют собой простейшую форму программного продукта, пригодную для повторного использования, продажи или распространения. Они позволяют избежать дублирования кода в тексте программ. Процедуры, как правило, параметризуются с целью придания им универсальности, причем эта параметризация динамическая, т. е. происходящая в период выполнения программы, а не этапе компиляции или препроцессирования, когда осуществляются подстановки или манипуляции с исходным текстом программы.

**Определение 3.4.1. Подпрограммой** будем называть именованный фрагмент программы, предназначенный для выполнения специальной инструкцией обращения.

### 3.4.1 Описание и использование

Принадлежностью инструкции вызова подпрограммы является ее имя и набор значений параметров, которые должны быть использованы для вычислений (набор фактических параметров). Вызов подпрограммы производится в тех точках программы, в которых это необходимо для выполнения соответствующих вычислений, причем наборы значений фактических параметров фиксируются на момент обращения и, следовательно, могут быть различными. Перед использованием подпрограммы необходимо дать ее описание, которое состоит из двух обязательных частей — заголовка и тела. В заголовке специфицируется вид подпрограммы (функция, процедура), ее имя, количество и виды параметров, способ передачи значений параметров, тип результата, а также некоторые атрибуты подпрограммы (возможность рекурсивного вызова, ссылка на тело, находящееся вне текста программы или ссылка на библиотеку, содержащую это тело в оттранслированном виде, в том числе и на другом языке программирования).

Заголовок определяет количество и порядок следование параметров, которые, как правило, фиксированы. В некоторых языках (C/C++, Perl) допускается описание функций с переменным числом параметров.

Тело подпрограммы представляет собой блок и содержит описания локальных и глобальных объектов вместе с совокупностью инструкций, описывающих действия алгоритма.

### 3.4.2 Вызов функций и процедур. Формальные и фактические параметры

Особенностью описания подпрограмм является наличие в заголовке так называемых **формальных параметров**, *параметризующих* тело программы. Этим достигается универсальность подпрограммы по сравнению с блоком. Формальные параметры являются локальными объектами подпрограммы и поэтому они по существу лишь обозначают объекты подпрограммы и им не отводится памяти при трансляции. Подобно всяким локальным объектам, память для формальных параметров выделяется только при входе в блок подпрограммы.

При вызове подпрограммы на место формальных параметров подставляются фактические параметры, так что их значения или ссылки на них замещают формальные параметры в период данного вызова подпрограммы. При другом вызове на место этих же формальных параметров могут подставляться другие фактические параметры, что и обеспечивает динамизм и универсальность подпрограмм.

В отличие от процедур подпрограммы-функции имеют один явный результат, как правило скалярный, явно присваиваемый имени функции. Обычно этот результат оставляется на определенном регистре соответствующего регистрового файла. Нескалярный параметр заменяется ссылкой на него, которая всегда скалярна; в Си и расширениях Паскаля тип результата функции может быть структурным. Кроме того, признаком хорошего тона является использование всех параметров функции как констант, так чтобы результат функции был ее единственным эффектом. Обращение к функции происходит, как правило, без специальной инструкции вызова в выражении соответствующего типа. В соответствии с парадигмой императивных языков фон Неймановского типа вычисление выражения понуждает к выполнению все указуемые им функции. Указатель функции, т. е. имя

функции с конкретным набором параметров, является таким же элементом выражения, как константа, переменная, элемент массива.

### 3.4.3 Передача параметров

Передача параметров является тем механизмом, с помощью которого устанавливается связь по данным между подпрограммой и внешней средой, из которой происходит обращение к подпрограмме.

В современных языках программирования существует несколько способов передачи параметров, различающихся степенью и характером взаимосвязи по данным вызывающей и вызываемой подпрограмм. Эти способы разбиваются на четыре группы по характерным особенностям передачи, хотя в конкретных языках эти качества могут комбинироваться или отсутствовать в чистом виде.

#### 3.4.3.1 Передача по значению

При передаче по значению каждому формальному параметру в соответствии с его типом при вызове подпрограммы выделяется память в области данных программы. Значение каждого фактического параметра (быть может, выражение) вычисляется в точке вызова подпрограммы и пересыпается (ay, матрицы!) в область памяти соответствующего формального параметра, после чего связь по этому параметру подпрограммы с вызывающей программой прекращается и, в частности, вызванная подпрограмма лишается возможности несанкционированного доступа к оригинальному значению. Передача по значению может быть охарактеризована как `read-only`, или, в языках АДА и PL/SQL, квалификатором `in` у соответствующего параметра. В языках Алгол-60 и Алгол W (предшественниках Паскаля) передача параметра по значению характеризуется описателем `value`, в Visual Basic `ByVal`. В Паскале передача по значению производится *неуказанием* спецификатора `var` у соответствующего формального параметра. Для этого способа конечно же, очень подходят фактические параметры-константы.

Рассмотрим пример аппроксимации синуса по формуле Тейлора. Всемирно-историческое значение этой формулы в том, что она сводит вычисление трансцендентных аналитически заданных функций к машиннореализуемым алгебраическим, давая экономное полиномиальное приближение (за линейное время, если применить схему Горнера). Но эта простая всем известная формула не применяется в вычислениях из-за медленной сходимости ряда и, как следствие, низкой точности. Вместо нее используются цепные дроби Вейерштрасса и экономизированные чебышевские приближения.

```
program ScalarByValue(input, output);

var pi, y, z : real;

function sint({ var } x : real) : real;
begin
  sint := x - x * x * x / 6.0 + x * x * x * x * x / 120.0;
  { x := 2.718; }
  writeln('x= ', x);
end; { sint }
```

```

begin
  pi := 3.14;
  y := sint(pi);
  z := sin(pi)
  writeln(y, ',', z);
end.

```

В приведенном примере *x* передается по значению неуказанием **var**. Поэтому, если раскомментировать строку *x* := 2.718;, то значение формального параметра *x* будет испорчено (в данном случае с  $\pi$  на  $e!$ ), но испорченное значение не вернется в вызывающую программу и там  $\pi$  останется равным  $\pi$ , но если раскомментировать **var** в заголовке программы, параметр *x* станет передаваться по ссылке, что приведет к немедленному изменению фактического параметра во время работы процедуры одновременно с формальным параметром.

### 3.4.3.2 Передача по результату

Так же как и при передаче по значению, при **передаче по результату** формальный параметр получает память в области данных вызываемой подпрограммы в момент обращения и соответствующий параметр используется во всех вычислениях как локальная переменная, ему, в частности, надо обязательно присвоить начальное значение (присваиванием, вводом или как результат работы внутренней процедуры или функции). В момент вызова подпрограмма, как и в предыдущем случае, получает адрес фактического параметра. Но в отличие от передачи по значению, по этому адресу *при нормальном завершении* подпрограммы возвращается вычисленное значение параметра-результата. Передача по результату обычно специфицируется указанием **out** или **result**. В Паскале (Бейсике) такого описателя нет, но спецификатор **var (ByRef)** позволяет достичь эффекта возврата значения, правда нижеописываемым ссылочным путем. В чистом виде параметры-результаты присутствуют в PL/SQL. В Фортране переменные по умолчанию передаются комбинированным по значению-результату.

Передача выражений и констант по результату бессмысленна и может быть синтаксически некорректной в строго типизированном языке.

Передача массивов по значению или результату помимо удвоенного расхода памяти на формальные параметры требует копирования содержимого массива. Ввиду большого размера массива эта форма передачи параметров является неэффективной. Поэтому ее применение должно быть обосновано.

### 3.4.3.3 Передача по ссылке

Этот способ называют так же передачей по адресу, передачей по простому имени, подстановкой параметров. При передаче параметров по ссылке подпрограмме передается адрес объекта в вызывающей программе и все обращения к параметру в подпрограмме происходят по этому адресу, так что объект в вызывающей программе заменяет соответствующий формальный параметр, то есть не производится передача самого значения и возврат результата. При этом *всякое* присваивание значения внутри подпрограммы приводит к *одновременному* изменению значения объекта в вызывающей программе. При

многозадачном или параллельном исполнении это может привести к совершенно другим результатам, нежели при передаче вышеуказанными способами.

При передаче по ссылке получается выигрыш во времени (значения не копируются) и в памяти (работа идет с оригинальным значением, которое не дублируется в подпрограмме), за исключением случаев, когда четырехбайтовая ссылка (`sizeof(void*) == 4!`) короче двухбайтового целого или однобайтной литеры. В Фортране массивы всегда передаются по ссылке, а переменные — только если соответствующие формальные параметры заключены в слэши.

При передаче по ссылке также не следует задавать фактические параметры-выражения или константы, даже если они синтаксически допустимы. Кроме того, в языке Фортран таким образом может быть испорчена константа, переданная по ссылке в подпрограмму.

```
program ArrayByReference(input, output);
type V = array[1..65536] of integer;

var a, b, c : V;

procedure m({var} x, y, z : V);
begin
  z[1] := x[1] + y[1];
  z[2] := x[2] + y[2];
  { x[1] := 4;
    y[1] := 4; }
end;

begin
  a[1] := 1;
  a[2] := 1;
  b[1] := 2;
  b[2] := 2;
  m(a, b, c);
  writeln(a[1], ', ', b[1]);
  writeln(c[1], ', ', c[2]);
end.
```

Если в процедуре раскомментировать два последних присваивания, то частично испортятся значения формальных параметров `x` и `y`. Но так же, как и в случае скалярных переменных, модифицированные значения векторов `x` и `y` не вернутся в вызывающую программу. Если дополнительно раскомментировать `var` в заголовке процедуры, то формальные параметры-массивы будут передаваться по ссылке и значения фактических параметров-массивов таким образом будут изменены.

#### 3.4.3.4 Передача по имени

Этот способ откладывает обработку фактических параметров до того момента, когда они действительно потребуются, то есть относится к идеям динамического связывания или динамического знакомства. Этот метод является самой мощной формой передачи параметров, одновременно наиболее опасной и неэффективной. Этот способ был введен

уже в Алголе-60, развит в Алголе-68 и в некотором виде присутствует в абсолютно объектных системах программирования.

При передаче параметра **по имени** фактический параметр перед выполнением подпрограммы буквально (текстуально) без каких-либо предварительных вычислений его значения подставляется вместо формального параметра (реализовать такую подстановку можно на уровне внутреннего представления, передавая ссылку на внутреннее представление — дерево соответствующего выражения). Таким образом, вместо передачи значения или ссылки, в тело подпрограммы передается правило, по которому должен быть вычислен параметр. При передаче по имени значение фактического параметра вычисляется всякий раз заново *в месте и в момент использования* в вызываемой подпрограмме, то есть всякий раз, когда в теле подпрограммы есть обращение к соответствующему формальному параметру. При этом значения фактических параметров вычисляются в контексте вызова, а не в контексте буквальной подстановки в теле подпрограммы.

Если параметр — простая скалярная переменная, то вызов по имени в точности совпадает с вызовом по ссылке. Если параметр — выражение из скалярных переменных, то результат аналогичен вызову по значению, за исключением того, что выражение вычисляется столько раз, сколько раз используется параметр, а не один раз при входе в подпрограмму. Если параметр содержит элемент массива, то при вызове по имени индексы (индексные выражения!) вычисляются при каждом использовании параметра в то время как при вызове по значению или ссылке значения индексов замораживаются. При передаче параметров по имени может случиться конфликт имен формальных и фактических параметров, который разрешается систематической заменой имен, проводимой интерпретативно средой языка при выполнении программы. Все это также усложняет транслятор, поскольку соответствующие места он должен оставлять в недотранслированном виде, подлежащем интерпретации в период выполнения программы. Поддержка процесса передачи по имени возлагается на языковую среду, роль которой существенно возрастает, увеличивая интерпретативную компоненту компилируемого языка.

При передаче по имени возможны неожиданные с примитивной точки зрения результаты. Так, вызвав процедуру

```
procedure swap(var a, b : integer {by name});  
var t: integer;  
begin  
    t := a;  
    a := b;  
    b := t;  
end;
```

инструкцией `swap(i, x[i])` получим

```
t := i;  
i := x[i]; { x[i] берется от прежнего i, после чего i изменяется }  
x[i] := t; { x[i] берется от измененного i! }
```

т. е. фактический обмен в связи с изменением `i` произойдет с другим элементом вектора `x`. Желаемый эффект достигается вызовом по ссылке, при котором значения-ссылки будут заморожены.

В заключение остановимся на передаче параметров-процедур. Поскольку значением программы является её двоичное выполнимое тело, субъект, а не объект обработки данных,

то передача процедур по значению-результату не имеет особого смысла. Передача по ссылке вполне отражает парадигму передачи управления в программе по заданному адресу в машине фон Неймана. Хорошие примеры передачи процедур приведены в книгах Н. Вирта и П. Грогоно. Это процедура интегрирования с подынтегральной функцией и процедура итерационного решения нелинейного уравнения, параметрами которой являются функции уравнения, разрешённого относительно  $x$  и её производной. Но самое интересное – это передача процедур по имени.

Все рассмотренные способы передачи параметров нашли применение в языках программирования. Каждый из них имеет свои достоинства и недостатки. Критериями выбора того или иного способа передачи параметров являются:

1. Защита области данных вызывающей программы от влияния подпрограммы.
2. Предохранение данных вызывающей программы от изменений до нормального завершения подпрограммы.
3. Экономия памяти, выделяемой всей программе.
4. Экономия времени вычисления.
5. Простота реализации и использования.

#### 3.4.4 Побочные эффекты

В теле подпрограммы могут быть использованы глобальные объекты, которые могут получить новые значения после её выполнения. Если подпрограмма вычисляет функцию, то целью ее выполнения является получение нового значения для идентификатора подпрограммы. Изменение же глобальных переменных в результате вычисления функции не является целью выполнения тела подпрограммы и в соответствии с этим называется **побочным эффектом**. Кроме того, побочный эффект может выражаться еще и в изменении значений фактических параметров функции, вызванном нежелательным для функций способом передачи параметров.

```
function sin({const} x : real) : real;
begin
    sin := x - x * x * x / 6.0 + x * x * x * x * x / 120.0;
    {writeln('sin(', x, ')=', sin);}
end; { sin }
```

Во избежание побочных эффектов формальные параметры функции не должны иметь спецификатора **var**. Более того, для надежности их следует атрибутировать **const**.

В процедуре побочным эффектом является изменение значений глобальных объектов и тех фактических параметров, которое не является целью выполнения тела процедуры. Рекомендуется защищать входные фактические параметры процедуры от модификации неуказанием **var** и даже указанием **const** при описании соответствующих формальных параметров.

Кроме того, ввод-вывод в процедурах и функциях также является побочным эффектом, если он модифицирует внешние глобальные файлы, не указанные в заголовке процедуры.

Побочный эффект приводит к ряду трудностей при программировании и отладке. Это также, как и использование меток и безусловных переходов **goto**, затрудняет автоматическую оптимизацию, верификацию, фильтрацию и обfuscацию программ.

### 3.4.5 Критика алгоритмической модели фон Неймана

Модель фон Неймана получена нами как развитие простой операционной модели Тьюринга, к которой также могут быть отнесены и различные автоматы. Математические основания модели Тьюринга точные и полезные [26]. Модель Тьюринга наделена памятью и чувствительностью к предыстории. Простая операционная семантика [103] машин Тьюринга состоит в частой смене состояний (state-transition), а сами состояния очень просты. Программы для МТ неясные и концептуально бесполезные. Строя диаграммное и схемное исчисление и предлагая нотацию для линейной записи (язык ОСТ), мы улучшили эту модель. Однако, избавиться от ее существенных недостатков не удалось. Только модель фон Неймана позволила предложить практически полезную и физически реализуемую концепцию.

К модели фон Неймана относят как аппаратные компьютеры, так и традиционные языки программирования. Их математические основания считаются сложными, громоздкими и концептуально бесполезными [26]. Так же, как и машина Тьюринга, модель фон Неймана существенно использует память и чувствительна к предыстории. Семантика модели фон Неймана также заключается в переходах из состояния в состояние, только состояния более сложные. Ясность программ и концептуальная полезность по сравнению с Тьюрингом несколько выше.

Критика традиционных языков программирования всегда основывается на исследовании их интеллектуального предка — компьютера фон Неймана. Когда фон Нейман и его гениальные коллеги предложили концепцию компьютера, они изящно и практически решили ряд математических, инженерных и программистских проблем. Несмотря на прошедшие с тех пор 60 лет, суть фон неймановского компьютера не изменилась: по-прежнему он состоит из процессора, памяти и соединяющей их шины, которая за один такт может передавать только одно слово между ними. Это слово либо данное, либо команда, либо адрес. Джон Бэкус предлагает называть эту шину «бутылочным горльшком» (узким местом) фон неймановской архитектуры. Задача программы состоит в некотором существенном изменении содержимого памяти процессором исключительно посредством перекачки данных через шину из памяти и обратно. Ирония ситуации состоит в том, что большую часть потока через «этую узость фон Неймана» [26] составляют не полезные данные, а всего лишь имена (адреса) данных, а также команды и данные, служащие лишь для вычисления таких адресов. Прежде чем слово можно будет послать через шину, его адрес должен оказаться в процессоре; поэтому он тоже должен быть послан через шину из памяти либо сгенерирован самим процессором. Если адрес посыпается из памяти, то *адрес этого адреса тоже должен быть послан из памяти либо выработан процессором, и т. д.* Здесь не просто снуют, в основном, курьеры (35 000 по Хлестакову), но еще и одни курьеры посыпаются за другими! С другой стороны, если адрес генерируется процессором, онрабатывается по фиксированному правилу (типа «добавить 4 к регистру адреса команд») либо по команде, которая должна быть затребована также через шину; в последнем случае оттуда же надо получить и её адрес... и т. д.

Разумеется, должен существовать менее примитивный способ внесения в память

больших изменений, чем мельтешение множества слов туда и обратно через бутылочное горлышко. Эта шина является не только узким местом для потока команд и данных задачи, но, что более важно, и интеллектуальным сужением, привязывающим нас к мышлению «слово за словом» вместо того, чтобы вдохновлять работу с более крупными концептуальными понятиями решаемой задачи. Программирование по фон Нейману большей частью заключается в планировании и спецификации огромного потока слов через это сужение, причем большая часть этого потока состоит не из самих значащих данных, а из сведений о том, где их искать.

Вопреки расхожему представлению обычные языки программирования в основном являются более высокоуровневыми и сложными программно реализованными версиями компьютера фон Неймана. При всей существенности различия между Паскалем и Си оно имеет меньше значения, чем тот факт, что оба этих языка основываются на программистском стиле компьютера фон Неймана. В языках программирования фон Неймана переменные используются для имитации ячеек памяти компьютера; операторы управления выражают его команды передачи управления и проверки, а операторы присваивания имитируют загрузку содержимого ячейки и некоторую арифметику с последующим запоминанием результата. Бутылочным горлышком фон Неймана для языков программирования является оператор присваивания. Именно он вынуждает нас программировать на уровне «слово за словом», как это имело место на аппаратном компьютере фон Неймана с его шиной обмена данными между памятью и процессором.

Рассмотрим типичную программу. Ее основу составляет последовательность операторов присваивания, содержащих некоторые переменные с индексами. Каждый оператор присваивания порождает результат, состоящий из одного слова. Программа должна организовать многократное выполнение этих операторов со сменой значений индексов, чтобы произвести желаемое итоговое изменение в памяти, поскольку она связана необходимостью изменять каждый раз только одно слово. Таким образом, программист имеет дело с потоком слов через бутылочное горлышко присваиваний в соответствии с тем, как он проектирует вложенность управляющих операторов для обеспечения необходимых повторений.

Кроме того, оператор присваивания расщепляет программирование на два мира. Первый мир включает в себя правые части операторов присваивания. Это упорядоченный мир выражений, мир с полезными алгебраическими свойствами (если не учитывать того, что эти свойства часто нарушаются побочными эффектами). Это тот мир, в котором происходит большинство полезных вычислений. Второй мир традиционных языков программирования — это мир операторов. Первичным оператором в этом мире является сам оператор присваивания. Все остальные операторы языка существуют для того, чтобы создать возможность выполнения вычисления, которое должно основываться на этой примитивной бутылкогорлочной инструкции присваивания.

Мир операторов не упорядочен и у него мало полезных математических свойств. Структурное программирование можно считать скромной попыткой внести некий порядок в этот хаотический мир, спагеттизованный goto, но и оно в малой степени способствует разрешению тех фундаментальных проблем, которые вносятся пословным стилем программирования фон Неймана с его примитивным использованием циклов, индексов и разветвления потока управления. Всеобщая запоренность языками фон Неймана предопределила сохранение фон неймановских компьютеров, сделала другие концепции неэкономичными и ограничила их развитие. Отсутствие законченных, эффективных стилей программиро-

вания, опирающихся на иные принципы, обезоружило проектировщиков и разработчиков новых компьютерных архитектур. В своей тьюринговской лекции Бэкус предлагает функциональный стиль программирования. Языки функционального программирования были задуманы как средство для рекурсивных построений. Процедуры в них могут служить данными, подставляемыми на место других аргументов. Каждое действие порождает некое значение, которое, в свою очередь, становится аргументом следующего действия и т. д. Основными средствами функционального программирования являются композиция и рекурсия. Функциональные языки хорошо подходят для смешанных вычислений, когда вычисления над данными приводят к получению программного кода, который может быть выполнен.

### 3.5 Критика языка Паскаль

Автор Паскаля, профессор Вирт, был удостоен Тьюринговской премии, высшей награды ACM. С этим простым и понятным языком программирования, хорошо подходящем для первоначального обучения, связан ряд достижений систем программирования. Это и раскрутка (boot-strapping) — постепенная реализация компилятора на том языке, с которого он компилирует, — и компилитивно-интерпретативная реализация переносимой системы программирования на Паскале, основанной на р-коде (ay, Java!). Именно интерпретатор р-кода, реализуемый за неделю на любом языке программирования на любом новом компьютере предопределил второе дыхание Паскаля, совершившего 25 лет назад победное шествие по всем появившимся в то время микропроцессорным платформам.

Недостатки этого замечательного языка являются продолжением его достоинств. Более того, эти достоинства вызвали поток критических статей в 70-х годах прошлого века [41]. Например, статья автора языка Си, лауреата Тьюринговской премии Брайана Кернигана, называется «Почему язык программирования Паскаль не является моим самым любимым языком программирования».

Язык Паскаль представляет собой программно-компилируемую реализацию машины фон Неймана и к нему можно отнести всю критику этой алгоритмической модели. Скалярный оператор присваивания, через бутылочное горлышко которого надо прокачивать сложные математические объекты, заставляет программиста постоянно заботиться о рациональном использовании этой шины (`:=`) и отвлекает от решения самой задачи. Справедливости ради отметим, что эта порочная система в прошлом, настоящем и ближайшем будущем остается единственным технически возможным средством автоматизации обработки информации.

Язык Паскаль является строго типизированным языком. Все объекты программ на Паскале должны быть описаны и употребляются в строгом соответствии с описаниями. Это предполагает строгую дисциплину программирования, неудобную в задачах системного программирования. Однако в своей строгости Паскаль непоследователен. Например, записи с вариантными частями образуют брешь в системе типизации, не уступающую тому самому метру государственной границы, о котором мечтал герой сатирического романа Остап Бендер. Тем не менее, особенности Паскаля таковы, что возможна реализация быстрого однопроходного эффективного компилятора. Языковая среда Паскаля также невелика, проста и эффективна. Паскаль содержит только такие языковые средства, которые эффективно компилируется на аппаратуру. Но это превращается в недостаток, потому что многие нужные любому программисту типы данных (такие как строки

и массивы переменной длины) либо не реализованы вообще, либо их (эффективная) реализация наталкивается на ряд проблем.

Модель идеального компьютера, представляемая Паскалем, слишком далека от современной программно-аппаратной среды, представляющей современными ОС. Товарное программирование на Паскале невозможно ввиду отсутствия модульности и внешних процедур. Паскаль проигрывает Си не только в выразительной силе и лаконичности. Библиотека языка Си представляет собой весьма мощную интерпретируемую компоненту.

"Опечатки весьма оживляют чтение скучных математических текстов"

Л. Эйлер

# Глава 4

## Распределение памяти

### 4.1 Статические и динамические объекты программ

Некоторые свойства объекта и связи с другими объектами остаются неизменными при любом исполнении его области действия (участка программы, где этот объект считается существующим). Такие свойства и связи называются *статическими*. Их можно определить по тексту программы, без ее исполнения [21].

Например, в Паскале тип объекта — одно из статических свойств. Сама область действия объекта — по определению статическое его свойство. Связь двух объектов по свойству принадлежности одной области действия — статическая связь. Свойство объекта при любом исполнении области действия принимать значения только из фиксированной совокупности значений — статическое свойство. Исчерпывающий перечень применимых к объекту операций, как правило, статическое свойство. Хотя в таком уже вошедшем в практику программирования динамическом языке как Python имеется возможность добавлять и удалять применяемые к объекту операции в ходе работы программы.

Другие свойства и связи изменяются в процессе исполнения области действия. Их называют *динамическими*. Например, конкретное значение переменной — динамическое свойство. Связь формального параметра с конкретным фактическим в результате вызова процедуры — динамическая связь. Размер конкретного массива с переменными границами — динамическое свойство.

Часто статические и динамические характеристики называют соответственно характеристиками *периода компиляции* (трансляции) и *периода выполнения* (runtime), подчеркивая то обстоятельство, что в период компиляции исходные данные программы неизвестны и, следовательно, динамические характеристики недоступны. Известна лишь информация, извлекаемая непосредственно из текста программы и тем самым относящаяся к любому ее исполнению (т. е. статическая информация).

Это деление характеристик на статические и динамические иногда оказывается слишком черно-белым. Например, длина массива с регулируемым размером — формального параметра функции языка Си — также формального параметра — может меняться от вызова к вызову в зависимости от значения фактического параметра-размера. Так что это и не чисто статическая характеристика, и не вполне динамическая, как значение переменной, которое можно изменить любым оператором присваивания. Возможна и более тонкая классификация характеристик по фактору изменчивости. Например, связывают изменчивость не с областью действия, а с периодом постоянства других его избранных

характеристик (выделяемой объекту памяти, связи с другими объектами и т. п.).

Уровень изменчивости характеристик объектов языка — одно из важнейших свойств языка. Одна крайняя позиция представлена концепцией неограниченного динамизма, когда по существу любая характеристика обрабатываемого объекта может быть изменена при выполнении программы. Такая концепция не исключает прогнозирования и контроля, но и не связывает их жестко со структурой текста программы.

Неограниченный динамизм присущ не только всем машинным языкам, но и многим языкам высокого уровня. Эта концепция в разной степени воплощена в таких интерпретативных динамических языках, как Бейсик, APL, Лисп, CLU, Smalltalk, CLOS, Python и т.п.

Другая крайняя позиция выражена в стремлении затруднить программисту всякое изменение характеристик объектов. Вводя объект, надо объявить характеристики, которым должно соответствовать всякое его использование. Конечно, неограниченной статики в программировании добиться невозможно (почему?). Так что всегда разрешается менять, например, значения объявленных переменных.

Зато остальные характеристики в таких статических языках изменить трудно. Обычно стремятся к статике ради надежности программ (за счет их дополнительной избыточности при обязательном объявлении характеристик возникает возможность дополнительного контроля) и скорости объектных программ (больше связей можно выполнить при трансляции и сэкономить время в период исполнения).

Вместе с тем, сама по себе идея объявления характеристик (прогнозирования поведения) и контроля за их инвариантностью требует создания, истолкования и реализации соответствующего языкового аппарата. Поэтому статические языки, как правило, сложнее динамических, их описания объемнее, реализации тяжеловеснее.

Если память выделяется (распределяется) в процессе трансляции и ее объем не меняется от начала до конца выполнения программы, то такой объект является *статическим*. Если же память выделяется во время выполнения программы и ее объем может меняться, то такой объект является *динамическим*.

Поскольку транслятор распределяет память на основе информации, содержащейся в описаниях объектов программы, то все объекты, описанные в основной программе являются статическими, в отличие от локальных переменных процедур.

## 4.2 Сылочный тип

Кроме известных нам статических объектов, многие языки программирования допускают также и динамические объекты. При этом в языках со строгой дисциплиной описаний, например, в Паскале динамический объект не может иметь собственного имени, так как все идентификаторы языка, кроме идентификаторов стандартных (предопределенных) процедур и функций, должны быть описаны в соответствующих разделах программы. Поэтому принято не *именовать*, а *обозначать* динамический объект посредством ссылки на него. В Паскале это достигается присоединением символа  $\uparrow$  к имени связываемой с каждым таким объектом переменной — ссылки на этот динамический объект. Переменная-ссылка должна быть описана в разделе объявлений программы как переменная ссылочного типа, в Паскале это раздел переменных **var**. При этом сама ссылка является статическим объектом. Ссылка занимает всего лишь одно машинное слово, что совсем немного ( $O(1)$ ) по сравнению с возможным размером растущего динамического объекта. Сылочный

тип — это такой же простой скалярный тип, как целый, вещественный и логический, также имеющий прямую аппаратную поддержку. Элементами множества значений этого типа являются конкретные ссылки на объекты указанного типа, созданные в основной памяти в процессе выполнения программы. Ссылки в Паскале строго типизированы, и не следует их считать универсальными адресами областей памяти.

Для названия ссылочного типа в языке Паскаль не зарезервировано никакого специального идентификатора:

```
<ссылочный тип> ::=  $\uparrow$ <имя указанного типа>
```

В языке Си указатель обозначается звездочкой, а в языке Модула 2 — словами **pointer** **to**.

Таким образом, ссылочные типы — это множества значений, указывающих на объекты некоторых целевых (указемых) типов. Разные указуемые типы порождают разные ссылочные типы, множества значений которых не пересекаются. Однако пустое ссылочное значение **nil** принадлежит любому из ссылочных типов. Оно указывает на отсутствие связи с объектом (аналогично нулю, который указывает на отсутствие количества). Следует заметить, что **nil** не является неопределенным значением ссылочной переменной  $\perp$ . Кстати, слово **nil** — аббревиатура Лисповского происхождения, означает **not in list**.

Ссылочный тип может быть именованным и неименованным. В первом случае в разделе **type** Паскаль-программы ему надо дать имя. Например:

```
type T = ...; { тип динамического объекта }
pointer =  $\uparrow$ T; { имя ссылочного типа — pointer }
```

Теперь элементами множества значений этого типа являются «адреса» областей памяти для размещения объектов типа **T**. Имя **pointer** можно употребить при описании переменной-ссылки:

```
var p : pointer;
```

При использовании неименованного ссылочного типа переменная-ссылка должна быть описана следующим образом:

```
var p :  $\uparrow$ T;
```

На Си это же выглядит так:

```
typedef T* pointer;
```

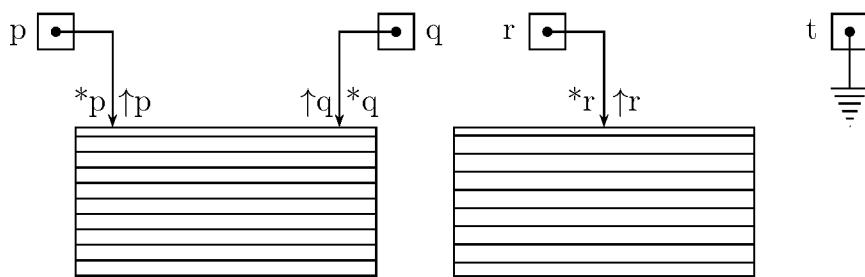
```
pointer p;
T* p1; // В отличие от Паскаля p1 того же типа, что и p!
```

В обоих случаях, обрабатывая описание переменной **p**, транслятор выделит для хранения значений ссылки место в основной памяти (обычно одно машинное слово, если ссылка реализована адресом значения, а не составным дескриптором динамического объекта). При этом следует учитывать, что память для ссылки находится в том состоянии, в котором она оставалась после предыдущего ее использования, т. е. там располагается

некоторая случайная последовательность 0 и 1, не определенная данной программой, или неопределенное значение  $\perp$ , см. п. 3.1.1. В отличие от константы `nil`, имеющей и имя, и постоянное значение (известное, правда, только разработчикам транслятора), используемое для указания на отсутствие ссылки на какой-либо объект, за неопределенностью ( $\perp$ ) не закреплено ни имени, ни значения (от случая к случаю в отведенных областях памяти оказываются, как правило, различные последовательности нулей и единиц). Поэтому, если транслятор не контролирует использование неопределенных значений, такая последовательность 0 и 1 может быть интерпретирована как ссылка на объект; это дает возможность доступа к случайной области памяти и может привести к непредсказуемому эффекту [43]. В некоторых реализациях Паскаля значения переменных инициализируются нулями, пробелами или `nil`'ами. Поскольку инициализация значений стандартом Паскаля не предполагается, это приводит к тому, что одна и та же программа на различных платформах работает по-разному.

Для переменных в точности одного и того же ссылочного типа определены операции *присваивания* и *разыменования* и отношение *равенства*. Если, например, результатом сравнения  $p = \text{nil}$  является значение `true`, то ссылочная переменная  $p$  не указывает ни на какой объект. Операция *разыменования* обеспечивает доступ к значению обозначаемого ссылкой объекта. В Паскале разыменование обозначается стрелкой, приписываемой справа от указателя ( $p^$ ), а в Си — звездочкой слева от него ( $*p$ ). Операция разыменования имеет аппаратную поддержку в любом современном процессоре (косвенная адресация). «Важно делать различие между переменной-указателем и указуемым объектом и быть очень осторожным при присваивании и сравнении указателей» [84]. В этом вопросе очень полезны графические иллюстрации.

При графической иллюстрации динамических структур данных принято изображать ссылочные переменные малыми прямоугольниками, а указуемые ими значения — большими. Стрелкой показывается связь между ссылкой и объектом. Пустое ссылочное значение `nil` (в Си это без лишней дипломатии просто 0) очень удачно визуализируется знаком заземления. Для иллюстрации процесса манипуляции со ссылками полезно одновременно с написанием программного кода рисовать серию таких картинок. Эти комиксы позволяют обнаруживать ошибки, неочевидные из текста программы.



В свою очередь эта графическая иллюстрация может быть выражена следующими соотношениями на языках программирования:

$p == q;$	$p = q;$
$*p == *q;$	$p^ = q^;$
$p != r;$	$p <> r;$
$*p != *r;$	$p^ <> r^;$
$t == 0;$	$t = \text{nil};$

$p == q;$	$p = q;$
$*p == *q;$	$p^ = q^;$
$p != r;$	$p <> r;$
$*p != *r;$	$p^ <> r^;$
$t == 0;$	$t = \text{nil};$

Динамические объекты определенного типа порождаются при выполнении встроенных процедур Паскаля и Си (**new** и **malloc**), фактическим параметром которой является ссылка на элемент этого типа. Поскольку в Си все процедуры являются функциями, то **malloc** возвращает указатель на выделенную область памяти как значение самой функции. В С++ порождение динамических объектов выполняется оператором языка **new**, который помимо выделения памяти и собственно порождения объекта вызывает конструктор объекта, выполняющий его инициализацию. Инициализация — это не только установление начальных значения полей объекта, но и запуск процессов, получение ресурсов, если они входят в состав активного объекта. В простейшем же случае сначала выделяется память для хранения значения, а затем в ссылочную переменную *p* помещается указатель на созданный объект типа *T*. Состояние памяти, выделенной под этот объект, вообще говоря, остается неопределенным ( $\perp$ ).

В С++ одна из форм оператора **new** позволяет сконструировать объект в ранее выделенной памяти, тем самым функции распределения памяти и создания объекта разделяются.

При работе процедуры **new** память для динамических объектов выделяется из так называемой «кучи» — области основной памяти машины, зарезервированной для этой цели. Процедура **new** при помощи операционной системы и посредством языковой среды Паскаля находит свободную в данный момент область памяти подходящего размера и соответственно присваивает переменной-ссылке значение начального адреса этой области. Размер кучи в системе обычно ограничивается некоторым значением, которое может быть задано при компиляции программы. Поэтому, если процедура **new** вызывается многократно (например, в случае зацикливания программы), то может наступить такой момент, когда вся память, предназначенная системой для динамических переменных, будет исчерпана, и программа не сможет продолжить работу.

В языке Паскаль определена также специальная процедура **dispose**, уничтожающая динамический объект, ссылка на который передана ей в качестве фактического параметра. Поэтому обработка динамических объектов обычно происходит по следующей схеме:

```
new(p); { порождение динамического объекта p↑ }  
p^ := ...; { обработка динамического объекта p↑ }  
write(p^);  
dispose(p); { уничтожение динамического объекта }
```

Однако точный результат работы (основной эффект) процедуры **dispose** в стандарте языка Паскаль не определен, поэтому реализация этой процедуры целиком зависит от решений, принятых разработчиком системы программирования. Иногда процедура **dispose** вообще не выполняет никаких действий. Развитые реализации Паскаля дают возможность повторного использования памяти, занимаемой динамическим объектом, возвращая ее в кучу.

Если значение ссылочной переменной утеряно, то связь с соответствующим объектом безвозвратно теряется.

```
var p, q : ^T;  
begin  
  new(p);  
  q := p;
```

```

dispose(p); { p указывает «в
никуда», программист помнит об этом; q тоже указывает «в
никуда», но об этом легко забыть }
end.

```

После уничтожения объекта *p* обе ссылки на него *p* и *q* становятся некорректными. Деструктор **dispose** мог бы обнулить передаваемую ему ссылку, но это не предусмотрено стандартом языка. Это проблема решается в современных языках сборщиком мусора. Поэтому необходимо присваивать ссылке значение **nil** непосредственно после вызова **dispose**. Альтернативой является использование «умного» указателя (smart pointer), все копии которого при уничтожении объекта автоматически обнуляются.

Таким образом, единственным источником начальных значений ссылочных переменных является процедура типа **new** — своеобразное присваивание, совмещенное с порождением динамического объекта. Поскольку ссылочный тип является внутримашинным, непечатным, его значения не имеют текстовых изображений и не могут быть введены или выведены процедурами **read** или **write**. В Си ссылочный тип хорошо согласован с целым, а целый — со ссылочным, поэтому ввод-вывод ссылочных переменных осуществляется обычными функциями **scanf** и **printf** (спецификация формата **%p**).

Конечно же, используя вариантные записи или объединения, можно наложить в памяти ссылочную и целую переменные и распечатать целочисленное значение ссылки.

```

type ptr2int = record
  case boolean of
    false: (val : integer);
    true: (ptr : ^integer);
  end;

var pi : ^integer;
  p2i : ptr2int;
begin
  new(pi);
  pi^ := 314;
  p2i.ptr := pi;
  writeln('Значение', pi^, 'находится
           по адресу',
           p2i.val : trunc(ln(p2i.val) / ln
                           (10.0)) + 1);
  dispose(pi);
end.

```

```

#include <stdlib.h>

typedef union
{
  int val;
  int* ptr;
} ptr2int;

int main(void)
{
  int* pi = malloc(sizeof(int));
  *pi = 314;
  ptr2int p2i;
  p2i.ptr = pi;
  printf ("Значение %d находится по_
           адресу %d", *pi, p2i.val);
  free(pi);
}

```

Поскольку в языке Си ссылочный тип идентичен адресному, а тот, в отличие от Паскаля, согласован с целым, ссылки могут быть не только напечатаны, но и введены, что может быть применено для задания абсолютных адресов. Поэтому пример на Си примет лаконичный вид:

```
printf ("Значение %d находится по адресу %p", *pi, pi);
```

В качестве примера статико-динамического дуализма рассмотрим константы. Казалось бы, значение константы — чисто статическое свойство, но в развитых языках программирования (C++) это свойство объекта может быть изменено в процессе работы программы. С другой стороны константы можно порождать и уничтожать динамически, т. е. постоянство значения не означает фиксированного времени существования.

```
const int* pci = new int(100); // Указатель на константу, инициализированную  
значением 100
```

```
*pci = 150; // Ошибка! Попытка изменить константное значение
```

```
pci = new int(200); // Указатель на новую константу со значением 200
```

```
int const* cpi = new int(100); // Неизменяемый указатель
```

```
*cpi = 150; // Теперь по адресу cpi находится число 150
```

```
cpi = new int(200); // Ошибка! Попытка изменить константный адрес
```

```
const int const* cpci = new int(100); // Неизменяемый указатель на константу
```

```
*cpci = 150; // Ошибка!
```

```
cpci = new int(200); // Ошибка!
```

Динамическими объектами в основной памяти можно также считать различные процессы, задачи, которые запускаются, останавливаются, синхронизируются, обмениваются данными и взаимодействуют в процессе выполнения программы.

Мы рассмотрели явное манипулирование динамическими объектами. Ранее изученная блочная структура с локальными переменными представляла собой неявное динамическое распределение и освобождение памяти, выполняемое языковой средой при входе в блок и выходе из него.

Локальные переменные динамически размещаются в стеке, что естественным образом обеспечивает их омонимию. Своеборзным случаем блочной структуры можно считать рекурсивный вызов процедур, при котором неявным образом динамически формируется семейство вложенных блоков со своими наборами независимых локальных переменных.

Стек, в отличие от кучи, предназначен для систематического последовательного выделения и освобождения памяти, в то время как куча рассчитана на беспорядочное размещение/освобождение объектов.

Естественно, что весь этот неявный процесс автоматического управления памятью недоступен и неподконтролен программисту.

Методы динамического распределения памяти подробно рассматриваются в первом томе монографии Дональда Кнута [63]. Динамические структуры данных, полезные для распределения памяти, будут изучаться в нашем курсе позднее.

# Глава 5

## Структуры данных

### 5.1 Уровни описания структур данных

При описании какого-либо объекта (простого или структурного) необходимо четко различать его абстрактные свойства и их конкретную реализацию на данном языке программирования и в данной машине [36, 59, 63, 68, 72].

Для описания абстрактных свойств рассматриваемого объекта удобно иметь определение класса всех объектов, обладавших этими свойствами, т. е. определение типа данных [43].

**Функциональной спецификацией** какого-либо типа данных называют внешнее формальное определение этого типа данных, не зависящее ни от языка программирования, ни от конкретной вычислительной машины. Дать формальное определение типа данных — это значит задать множество значений этого типа и множество изображений этих значений вместе с правилом их интерпретации (только через изображения можно работать со значениями объектов типа), а также базовое множество атрибутов этого типа, включающее изображения некоторых выделенных значений, операции и их свойства, отношения и их свойства, функции создания, доступа и модификации объектов этого типа [6]. Функциональная спецификация обеспечивает полное описание данного типа, однако в нее не включаются никакие требования к внутренней организации объектов этого типа, диктуемые особенностями используемых языка программирования и машины.

Например, функциональная спецификация *целого типа* вообще определяет множество изображений значений этого типа как последовательность цифр со знаком или без знака, полиномиальную интерпретацию любого такого изображения как числа в позиционной системе счисления с основанием 10, операции (+, -, \*, деление (`div` и `mod`)) с их свойствами (законы ассоциативности, дистрибутивности и т. д.), отношения (=, <>, <, <=, >, >=) и их свойства, а также ряд функций, аргументами и/или результатами которых являются значения этого типа [6].

Другой пример: функциональная спецификация структурного типа «одномерный массив из пяти элементов вещественного типа, пронумерованных от 0 до 4», включает в себя возможные изображения значений этого структурного типа (пятерки слов), возможные изображения значений его элементов (последовательность двух групп цифр, разделенных точкой, со знаком или без знака), интерпретацию любого такого изображения как числа с фиксированной точкой в позиционной системе счисления с основанием 10, поэлементные операции и отношения (атрибуты вещественного типа), функцию доступа к значению

массива (поэлементный доступ по имени массива и индексу его элемента, причем индекс может принимать только значения 0, 1, 2, 3 или 4), функцию модификации массива (модификация любого элемента массива как переменной вещественного типа приводит к модификации массива).

**Логическое описание** — это отображение функциональной спецификации на средства выбранного языка программирования. При выполнении этого отображения могут быть две ситуации: 1) в выбранном языке программирования есть подходящий тип данных; 2) подходящий тип данных в языке не определен.

В первом случае в программу включают описание объектов имеющегося типа в соответствии с синтаксическими правилами этого языка. Например, при необходимости описать объект X как «переменную целого типа» в раздел переменных Паскаль-программы следует включить предложение

**X : integer;**

а в модуль программы на Си — описатель

**int X;**

так как и в том, и в другом языке программирования определен тип целый.

Во втором случае необходима декомпозиция объекта на такие составные части, которые могут быть описаны как отдельные объекты программы средствами выбранного языка программирования, или программное моделирование требуемого для решения задачи типа данных. При таком моделировании следует использовать типы данных, определенные в языке как базовые, и реализовать необходимые операции и отношения процедурами и функциями.

Машина Тьюринга манипулирует только литерами, поэтому целый тип должен быть промоделирован работой со словами-изображениями, как это было сделано в первой части курса. В настоящее время трудно привести пример языка, где совсем бы не было чисел, но в первых версиях Лиспа, Снобола и Пролога это имело место, и приходилось соответствующие функции реализовывать вручную.

Например, если в языке программирования нет записей (Фортран), то их поля кодируются как отдельные переменные, присваивания и сравнения реализуются группами соответствующих скалярных операций, а массивы записей — таблицы — согласованными наборами неаггрегированных векторов.

Пусть требуется написать программу обработки таблицы из 100 строк, каждая из которых состоит из трех компонент. И пусть первая компонента принимает целочисленные значения (табельный номер), вторая — текстовые данные длиной не более 20 знаков (фамилия), а третья — значения действительных чисел (доход). Если для реализации (например, в соответствии с приказом министерства авиационной промышленности СССР!) надо использовать язык Фортран, то приходится учитывать, что в этом языке нет записей. Поэтому одним из вариантов реализации может быть декомпозиция обрабатываемого массива и описание его частей одномерными массивами Фортрана, согласованными между собой по порядку расположения элементов:

**INTEGER ID (100)  
LOGICAL NAME (2000)**

## **INTEGER LEN (100) REAL SALARY (100)**

Здесь целый массив  $id$  является вектором из 100 компонент, причем  $i$ -тый элемент содержит табельный номер  $i$ -го сотрудника. Доступ к текстовым данным в массиве name (типа logical, т. к. литерного типа в Фортране нет!) осуществляется с помощью простого вычисления индекса  $(n - 1) \cdot 20 + 1$ . В массиве len (тоже из 100 компонент) на  $i$ -тое место помещена длина фамилии сотрудника, поскольку в Фортране нет и строкового типа и приходится отмеривать строки-отрезки массива name вручную. В массиве salary содержатся вещественные числа — значения третьих компонент элементов исходных данных. Их порядок должен строго соответствовать расположению целых чисел в векторе  $id$ .

Другой вариант реализации обработки исходного массива с помощью языка Фортран может быть основан на использовании описателя equivalence, аналогичного вариантной записи.

При выборе языка Паскаль для реализации решения той же задачи не требуется декомпозиции исходной таблицы на векторы-столбцы, так как этот язык позволяет использовать массив записей, каждая запись которого состоит из именованных полей, а каждое поле записи может иметь свой тип и длину:

```
var
  collaborators : array[1..100] of record
    id : integer;
    name : array[1..20] of char;
    salary : real
  end;
```

**Физическое представление** — это конкретное отображение на память машины объектов программы в соответствии с логическим описанием. Такое отображение всегда связано с линеаризацией структуры [5], так как память машины обычно состоит из некоторых единиц (слов или байтов), пронумерованных последовательными целыми числами, начиная с нуля. Используя многомерный массив на языке программирования, иногда необходимо знать принцип его линеаризации, и учитывать его при ручной навигации по массиву. Например, двумерный массив (матрица) в Фортране линеаризуется «по столбцам» (наследие архитектуры IBM-709, на которой впервые был реализован Фортран), а в языке Паскаль —«по строкам» (естественный порядок чтения слева направо и сверху вниз; в некоторых культурах читают по столбцам (иероглифы) или справа налево (семитские письменности)). Поэтому при передаче матрицы из Паскаль-программы в Фортрановскую библиотечную подпрограмму ее необходимо транспонировать. Если передается массив регулируемого размера, то необходимо помнить, что его линеаризация не является главнодиагональным кусочком матрицы, а его элементы просто будут размещены в строчном или столбцовом лексикографическом порядке.

Конструктивные особенности памяти как последовательности слов с произвольным доступом обуславливают два вида физического представления объектов в памяти машины: *слошное* и *цепное*.

*Слошное представление* — это представление, при котором объект размещается в памяти машины в непрерывной последовательности единиц хранения. Например, переменная целого типа представляется на физическом уровне одним машинным словом,

состоящим из двух, четырех или восьми байтов с последовательными адресами. При этом адрес первого байта должен быть кратным двум, четырем или восьми соответственно, и он считается адресом всего слова. Этого требует конструкция устройства памяти, выдающего не отдельные байты, а целые слова. В противном случае придется тратить два такта устройства на выборку одного слова. Для всех простых объектов используется сплошное представление. Оно часто применяется для таких структур данных, как массив, очередь, стек, дек (вспомните, что при доказательстве тезиса Тьюринга-Черча массив был первоначально построен как отрезок ленты машины Тьюринга!). Сплошное представление имеет эффективную аппаратную поддержку, поскольку для доступа к элементам требуется простое индексное вычисление (**номер слова** – 1) × **длина слова** с фиксированной малой ценой доступа  $O(1)$ . Для внешних электромеханических устройств, основанных на последовательном движении носителя, сплошное представление является единственным возможным, поскольку скачкообразное движение к произвольному элементу не предусмотрено конструкцией.

*Цепное представление* — это такое представление, при котором значение объекта разбивается на отдельные части, которые могут быть расположены в разных участках памяти машины (необязательно подряд), причем эти участки тем или иным способом связаны «в цепочку» с помощью указателей, т. е. они содержат ссылки на следующие части объекта. Цепное представление используется, как правило, для динамических структурных объектов (списки, деревья, очереди, стеки и деки).

Если объект имеет статически заданный конкретный размер или такой размер можно задать, исходя из каких-либо разумных соображений, сознательно ограничивая тем самым максимальный размер такого объекта (например, задать наибольший допустимый размер очереди), то предпочтительнее сплошное представление; при этом упрощается доступ к значению объекта и к его компонентам. В противном случае используется цепное представление.

В заключение этого раздела отметим, что логическая структура, позволяющая реализовать данное функциональное описание, редко бывает единственной, точно так же, как и соответствующее физическое представление. Выбор может осуществляться по нескольким критериям, в частности по критерию общности: делают так, чтобы одно и то же логическое описание могло обслуживать сразу несколько функциональных спецификаций [72].

Рассмотрим еще раз иерархию описаний на примере т. н. длинной арифметики. На абстрактном математическом уровне вся арифметика сколь угодно длинная, никаких количественных ограничений не накладывается. При отображении на средства языка программирования может не найтись соответствующего типа данных, например, вещественного. В последнем случае его придется моделировать программным путем. Или может быть, что подходящий тип есть, но его точность недостаточна. В таком случае подключается длинная арифметика, которая может быть встроенной (Python, PL/SQL), или библиотечной (C++ с библиотекой boost, Java), или созданной программистом вручную. На уровне физического представления некоторые длинные типы могут иметь аппаратную поддержку, либо эмулироваться микропрограммно и даже, как это было на первых процессорах Intel, иметь чисто программную интерпретацию на традиционном машинном уровне.

## 5.2 Файл

Динамические объекты могут размещаться не только в основной, но и во внешней памяти ЭВМ [43]. Динамические объекты, размещаемые на устройствах внешней памяти (магнитных, оптических, флэш и др.), обычно имеют структуру классического последовательного файла-карточки (линейное медленное последовательное движение ленточного носителя) или массива (в случае дисковых устройств с высокой скоростью вращения позиционирование головки и сектора выполняется хоть и тоже последовательно, но очень быстро). Файл, как и любой другой динамический объект, может быть пустым; количество составляющих его компонент может меняться в процессе выполнения программы. Однако, в отличие от динамических объектов в основной памяти, файл должен иметь собственное имя, по которому его можно обнаружить в определенном месте конкретного внешнего запоминающего устройства. Это является следствием большей долговременности файлов как динамических объектов на внешней памяти: время жизни файла, как правило, превышает время жизни программы, в пределах которого существуют все ее резидентные динамические объекты. Следовательно, имя файла и его характеристики должны быть известны операционной системе до начала выполнения программы обработки этого файла, хотя с общей точки зрения безымянные файлы как указатели на файловые переменные вполне правомерны. Отметим некоторую несистематичность динамических объектов Паскаля: создать или уничтожить файл средствами стандартного Паскаля (**new**, **dispose**) нельзя.

Файлы играют важную роль в любой вычислительной системе [72]. При этом операции, которые вычислительная система может выполнять над ними, определяются *принципами*, заложенными в конструкции тех устройств памяти, на которых эти файлы хранятся [63]. Например, для файлов, хранящихся на МЛ и МД, определены чтение, запись и стирание; для принтера — только запись, а для CD-ROM и сканера — только чтение. С другой стороны, флэш-память не основана на электромеханическом (линейном или вращательном) движении носителя.

Заметим, что понятие файла используется и как абстракция данных, хранящихся на любом из запоминающих устройств, что позволяет единообразно формулировать их общие характеристики, а также определять операции над ними [20].

Слово «файл» в языке Паскаль употребляется для динамических объектов, состоящих из *последовательности* компонент одного типа. Сама последовательность устанавливает естественный порядок компонент, и в любой момент непосредственно доступна (для чтения или для записи) только одна текущая компонента. Другие компоненты становятся доступными по мере продвижения по файлу. Длина файла (число компонент) не фиксируется: в процессе выполнения программы файл может стать пустым, а может быть дополнен новыми компонентами. Мы уже знакомы с лентой машины Тьюринга — последовательностью ячеек со строго последовательным доступом.

По отношению к программе файлы могут быть *внутренними* и *внешними*. Внутренними (локальными) являются файлы, которые описаны в разделе **var** Паскаль-программы и имена которых не указаны в заголовке этой программы. Данные, находящиеся в таком файле перед завершением работы программы, недоступны для дальнейшего использования, так как память для файла была выделена с целью *временного хранения* данных в период работы программы и должна быть освобождена для повторного использования другими программами. Файлы, созданные (сгенерированные) до начала выполнения

программы, а также сгенерированные в процессе ее выполнения и оставленные для долговременного хранения, называют внешними. Имена таких файлов необходимо не только описывать в разделе **var** Паскаль-программы, но и обязательно указывать в заголовке программы.

Концепция файлов Си более общая, чем в Паскале, она испытала большое влияние ОС UNIX. Наряду с традиционными файлами в Си файлами являются стандартные потоки (`stdin`, `stdout`, `stderr`), конвейеры, физические устройства вроде принтеров или датчиков температуры процессора, виртуальные устройства типа «черной дыры» `/dev/null`. В отличие от Паскаля в Си есть функция быстрого подвода к нужному месту файла, которая при соответствующей аппаратной поддержке может обеспечивать прямой доступ, моделируя массивы. Работа с файлами в Си, также как и в Паскале, организована на процедурном уровне. Но соответствующий набор функций библиотеки языка Си существенно более динамичен и прагматичен. Возможно создание, переименование, удаление постоянных и временных файлов, имена которых задаются строковыми константами или переменными. В отличие от Паскаля каждый файл в Си может быть открыт и использован как в текстовом, так и в двоичном режимах. Такая дуалистическая интерпретация файлов естественна для языка системного программирования.

### 5.2.1 Функциональная спецификация.

Обозначим через  $F_T$  файловый тип с компонентами типа  $T$ . Для задания функциональной спецификации файлового типа необходимо, во-первых, указать множество значений, множество изображений значений этого типа, правила интерпретации изображений и, во-вторых, определить базовое множество атрибутов этого типа (множество операций над значениями этого типа и их свойства, множество отношений и их свойства, выделенные функции и т. п.) [43].

Значениями типа  $F_T$  являются сколь угодно длинные, но конечные последовательности компонент типа  $T$  (изображаемые по правилам интерпретации типа компонент  $T$ ). Такое бесконечное множество значений можно определить формально с помощью операции *конкатенации* (слияния, склеивания, сцепления) двух файлов, состоящих из компонент одного и того же типа: если  $f_1 = \{x_1, \dots, x_m\}$  и  $f_2 = \{y_1, \dots, y_n\}$ , то  $f_1 \parallel f_2 = \{x_1, \dots, x_m, y_1, \dots, y_n\}$ , где  $\parallel$  — знак операции конкатенации.

Тогда множество значений файлового типа строго определяется следующими порождающими правилами [72]:

1.  $\{\}$  есть файл типа  $F_T$  (пустая последовательность, пустой файл);
2. если  $f$  есть файл типа  $F_T$  и  $t$  есть объект типа  $T$ , то  $f \parallel \{t\}$  есть файл типа  $F_T$ ;
3. никакие другие значения не являются файлами типа  $F_T$ .

Это определение является конструктивным, т. к. позволяет постепенно построить любой файл, начиная с пустого, последовательным дописыванием его компонент в необходимом порядке, как мы это уже делали раньше при нисходящей разработке схем Тьюринга путем уточнения цели (точки — пустой схемы).

Базовое множество атрибутов файлового типа:

1. операция конкатенации, определенная выше; ее свойство — несимметричность;

2. операция присваивания определяется как покомпонентное копирование одного файла в другой с сохранением порядка и количества компонент.
3. отношение равенства: два файла, состоящие из компонент одного и того же типа, равны тогда и только тогда, когда они имеют одинаковую длину, а их соответствующие компоненты — равные значения; отношение равенства симметрично;
4. функции: создание (порождение пустого файла), доступ (последовательный доступ к каждой компоненте, причем только для чтения значения: чтобы прочитать значение  $k$ -той компоненты файла, надо прочитать предварительно  $k - 1$  компоненту от начала файла), модификация (дозапись компоненты того же типа в конец файла — фактически это конкатенация файла и новой компоненты типа  $T$ ), уничтожение (стирание) файла.

### 5.2.2 Логическое описание и физическое представление

В Паскаль-программе может быть использован как именованный, так и неименованный файловый тип [43]. Именованный файловый тип должен быть описан в разделе описания типов:

```
type T = ...; { тип компоненты файла }
    <имя файлового типа> = file of T;
```

Тогда объект-файл должен быть описан в разделе переменных таким образом:

```
var <имя объекта-файла> : <имя файлового типа>;
```

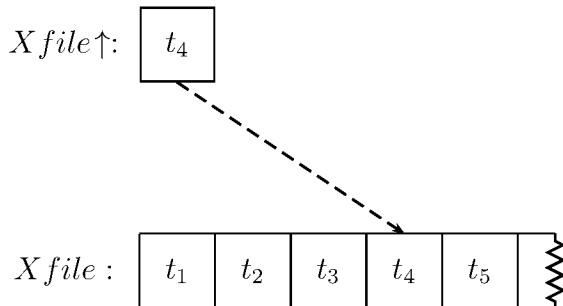
При использовании неименованного файлового типа описание такого же объекта выглядит следующим образом:

```
var <имя файловой переменной> : file of T;
```

При обработке описания объекта-файла  $f$  транслятор должен выделить память в генерируемой программе под буфер файла, достаточную для размещения значения одной компоненты этого файла (тем самым порождая буферную переменную типа  $T$ ). Обозначается буфер файла таким образом:

<имя файловой переменной> $\uparrow$

Например, если имя файла —  $Xfile$ , то его буфер —  $Xfile\uparrow$ , то после прочтения трех его компонент возникнет следующая ситуация:



Последовательный характер обработки и наличие буферной переменной (буфера файла) предполагают, что файлы могут ассоциироваться с внешней (вторичной) памятью и другим периферийным оборудованием. Буферная переменная является Паскаль-абстракцией буфера внешнего устройства операционной системы. Как и где хранятся компоненты — зависит от реализации, мы лишь допускаем, что некоторые из них могут находиться в какой-то момент в основной памяти и только компонента  $f \uparrow$  всегда непосредственно доступна [8, 9], если, конечно, файл не пуст.

Для описания файла в Си необходимо объявить переменную предопределённого типа **FILE\***. Созданный компилятором виртуальный файловый дескриптор может быть динамически связан с конкретным файлом, потоком или устройством с помощью функций стандартной библиотеки языка Си.

**FILE\* <имя объекта-файла>;**

Создание и модификацию файла можно реализовать в Паскаль-программе, используя стандартную функцию **eof(f)** и процедуры **reset(f)**, **rewrite(f)**, **get(f)** и **put(f)**, где  $f$  — имя файла (операторов ввода/вывода ни в Паскале, ни в Си нет, что объясняется большим разнообразием внешних устройств). Алгоритмы выполнения этих процедур приведены в [9].

Известные нам процедуры **read** и **write** выражаются через низкоуровневые средства ввода/вывода Паскаля следующим образом:

<b>read(f, x);</b>	$x := f^\wedge;$ <b>get(f);</b>
<b>write(f, x);</b>	$f^\wedge := x;$ <b>put(f);</b>

Процедура **reset** готовит указанный файл для чтения начиная с первого элемента. Если файл не пуст, то в результате **reset** в буферную переменную  $f \uparrow$  заносится значение первой компоненты файла и функция **eof** получает значение **false**. Вторым параметром процедуры **reset** во многих расширениях Паскале является строка с именем реального файла ОС, сопоставляемого данной файловой переменной Паскаля. Поскольку параметр с именем файла передается операционной системе через программный интерфейс (системный вызов, аналогичный Shell), то в написании имени файла дозволены все допустимые стандартным Shell'ом возможности:

```
reset(f, ' ../../data/file1.txt');
```

Файл с данными берется из директории, находящейся на 2 уровня выше текущей директории, из которой запущена программа.

Процедура **reset** открывает файл для чтения, не стирая содержащихся в нем данных.

Процедура **rewrite** устанавливает указанный файл в режим записи (перезаписи), стирая возможно содержащиеся в нем данные. Если файл пуст, то **rewrite** также применим, хотя никаких заметных действий при этом не производится. Вторым параметром процедуры **rewrite** также иногда может быть строка с именем реального файла ОС, сопоставляемого данной файловой переменной.

Поскольку процедура **rewrite** открывает файл для записи, стирая содержащиеся в нем данные, то невозможно напрямую осуществить дозапись данных в существующий файл.

Для этого необходим новый файл, в который сначала надо скопировать содержимое старого файла, а затем, не прекращая сеанса записи, дописать новые компоненты.

Предикат **eof** принимает значение **true**, если из файла прочитана последняя компонента. Если из файла не было произведено ни одной операции чтения, то значение данной функции не определено и ее поведение зависит от реализации. Например, Borland Pascal 7.0 установит **eof** равным **false**. Иногда используется трюк с безопасным на пустом файле фиктивным чтением (**read(f)**), побочным эффектом которого является задание значения **eof** для указанного файла.

```
read(f);
if eof(f) then writeln('Файл_пуст!');
```

Часто в этом нет необходимости и значение **eof** устанавливается автоматически.

Поскольку файл, как и массив, можно трактовать как указатель на память, в данном случае внешнюю, а также ввиду того, что нерезидентные файловые переменные имеют большой размер и время доступа, встроенная операция присваивания файлов в Паскале не реализована.

```
procedure fcpy(var f1, f2 : file of T); { f1 := f2 }
begin
  rewrite(f1);
  reset(f2);
  while not eof(f2) do begin
    f1^ := f2^;
    put(f1);
    get(f2);
  end;
end;
```

Операция конкатенации двух файлов в языке Паскаль тоже не определена. Пополнение файла новой компонентой (добавление к концу последовательного файла новой компоненты), по существу, является конкатенацией файла и переменной того же типа  $T$ , что и компоненты файла, и осуществляется с помощью стандартной процедуры **put(f)**, где  $f$  — имя файла, при этом буферной переменной  $f \uparrow$  перед ее выполнением надо присвоить значение присоединяемой к файлу компоненты. Поэтому при необходимости конкатенации файлов ее несложно реализовать средствами языка Паскаль, последовательно присваивая значения компонент второго файла буферу первого и выполняя каждый раз процедуру **put**.

```
procedure fcat(var f, f1, f2 : file of T); { f := f1 // f2 }
begin
  rewrite(f);
  reset(f1);
  reset(f2);
  { f1 и/or f2 могут быть пустыми. С repeat работать не будет! }
  while not eof(f1) do begin
    f^ := f1^;
    put(f);
    get(f1)
  end;
```

```

while not eof(f2) do begin
    f^ := f2^;
    put(f);
    get(f2)
end
end;

```

Отношение равенства над файлами в языке Паскаль не определено. При необходимости его также несложно реализовать средствами языка.

```

function feq(var f1, f2 : file of T) : boolean;
{ feq := f1 = f2 ⇔ #(f1) = #(f2) & f1[i] = f2[i] (for i := 1 to #(f1)) }
var res : boolean;
begin
    reset(f1);
    reset(f2);
    res := true;
    { res – индикатор совпадения, сравнение файлов идет, их соответствующие компоненты равны, и ни один из них не закончился. }
    while res and ((not eof(f1)) or (not eof(f2))) do begin
        res := res and (f1^ = f2^);
        get(f1);
        get(f2)
    end;
    { Поправка индикатора, если компоненты файлов совпадали вплоть до окончания одного из них раньше другого }
    feq := res and eof(f1) and eof(f2)
end;

```

В этом примере предполагается, что переменные типа  $T$  сравнимы, что заведомо справедливо для скалярных и строковых типов Паскаля. В случае, если для типа  $T$  не определено отношение равенства, необходимо предусмотреть булевскую функцию сравнения.

В заключение еще раз заметим, что файлового типа в Си нет. Для работы с файлами в стандартной библиотеке Си имеются определения файловых дескрипторов и функций, доступных через заголовочный файл `<stdio.h>`. Вместо описания файловой переменной в Си необходимо определить переменную-дескриптор файла, которая является (недопустимой в Паскале!) ссылкой на файл. Далее (с большим динамизмом и гибкостью!) Си позволяет сопоставить этому дескриптору существующий либо вновь создаваемый реальный файл, установить режимы доступа ( чтение, запись, перезапись и др.), задать формат и типизацию (интерпретацию) данных файла. Набор функций работы с файлами в Си гораздо более развит, чем в Паскале.

Файлы обычно ассоциируются с устройствами внешней памяти и с устройствами ввода-вывода (МЛ, МД, принтер, терминал), о физической организации которых мы уже неоднократно говорили [72].

## 5.3 Вектор

*Статические* массивы, т. е. массивы с фиксированной длиной, удобны многим: их длина известна, поэтому компилятор в состоянии не только сгенерировать инструкции по выделению памяти (в стеке), но и автоматически рассчитать адреса компонент массива при обращении к ним. В случае такого массива легко проверять выход за его границы, уменьшая тем самым вероятность ошибок. Но фиксированная длина во многих случаях становится недостатком. В приложениях линейной алгебры, например, при решении систем линейных алгебраических уравнений размерность задачи редко бывает заранее известной. Поэтому использование статических массивов в таких задачах приводит одновременно как к серьезным непроизводительным затратам, так и к ограничениям: программа оказывается в состоянии решать задачи не более заданной максимальной размерности. Превышение размерности задачи всего на единицу по сравнению с максимумом приводит к отказу программы.

В этой ситуации оказываются полезными *динамические* массивы или *векторы*. Эти массивы располагаются в куче, и их размер может быть задан произвольно. В идеале длина такого массива ограничена лишь размером доступной памяти, что, разумеется, на много превышает размер стека, выделяемого программе. Поскольку длина массива может быть легко изменена, то непроизводительные издержки отсутствуют. Недостатком таких массивов является (возможно) большее среднее время доступа к элементу. Это связано с иерархичностью памяти современных ЭВМ, поддерживающих трех- или четырехуровневую систему основной памяти. Так, мало- или редкоиспользуемые данные, размещенные в виртуальной памяти, сбрасываются в swap-раздел диска. При необходимости эти данные подгружаются в основную память ЭВМ. Но современные процессоры никогда не работают напрямую с данными в ОП. Все операнды должны находиться в регистрах, куда они попадают через кэш-память, которая может иметь 2 и более уровней.

Главным источником критики векторов является обязанность программиста вручную распределять память, что, по мнению многих, в т. ч. и Н. Вирта, служит гарантированным источником ошибок. Но эта критика основывается на доводах 20–30-летней давности. Уже QuickBasic умел отлавливать выход за границу массива. Java и Delphi отслеживают обращения к памяти и возбуждают исключения при несанкционированном доступе. В стандартной библиотеке шаблонов (Standard Template Library, STL) языка C++ есть класс `std::vector`, и для него определены два способа доступа к элементам: непосредственный, (небезопасно) реализованный через `operator[]` и поддерживающий обычный синтаксис доступа к элементам массива, и через функцию-член `at`, которая проверяет, чтобы индекс удовлетворял условию  $0 \leq i < \text{size}()$ , где  $i$  — индекс, `size()` — длина вектора (индексация массивов в C/C++ ведется от 0). Проверка занимает некоторое время и приводит к замедлению работы программы, поэтому создатели STL C++ предусмотрели возможность избежать эту проверку.

### 5.3.1 Функциональная спецификация

Вектор является последовательностью переменной длины, и время доступа к элементам этой последовательности постоянно и не зависит от длины последовательности. Количество элементов вектора не фиксировано и всегда может быть изменено. Вектор с 0 компонент называется пустым (по аналогии с пустым файлом).

В математике операции, такие как сложение и скалярное произведение, можно производить только с векторами равной размерности (длины). Поэтому длина является важной характеристикой вектора. Обозначим через  $I_n$  множество допустимых индексов вектора,  $I = 0, \dots, n-1$ . Мощность этого множества есть ни что иное как длина вектора. В качестве индекса первого элемента берется 0, т. к. это удобно для реализации вектора в ЭВМ: как правило, операционные системы, выделяя блок памяти, возвращают адрес начала этого блока. Для доступа к компоненте вектора необходимо прибавить к адресу блока размер одной компоненты, умноженной на ее индекс. Если индексация ведется не с 0, то дополнительно необходимо провести сложение индекса с некоторым числом, что замедляет доступ.

Теперь определим поведение операции «изменение размера». При уменьшении размера вектора с  $n$  до  $m$  все компоненты с индексами  $0, \dots, m-1$  должны сохранить свое значение. При увеличении длины вектора с  $n$  до  $p$  компоненты с индексами  $0, \dots, n-1$  должны сохранить свое значение, а с индексами  $n, \dots, p-1$  могут быть или неопределены, или проинициализированы значениями по умолчанию.

Поскольку ОС не всегда оказывается в состоянии выделить дополнительную память сразу за блоком, отведенным под вектор, то операция изменения размера требует копирования всех оставшихся элементов. Мы сознательно определяем столь неэффективную операцию, время выполнения которой есть  $O(N)$ , чтобы операция доступа к данным имела постоянное время выполнения, как это определено для простого статического массива.

Пусть компоненты вектора имеют тип  $T$ . Тогда для типа  $V_{T,I_n}$  определены следующие операции:

СОЗДАТЬ:	$N \rightarrow V_{T,I_n}$
ПУСТО:	$V_{T,I_n} \rightarrow \text{boolean}$
ДЛИНА:	$V_{T,I_n} \rightarrow N$
ЗАГРУЗИТЬ:	$V_{T,I_n} \times I_n \rightarrow T$
СОХРАНИТЬ:	$V_{T,I_n} \times I_n \times T \rightarrow V_{T,I_n}$
ИЗМЕНИТЬ_ДЛИНУ:	$V_{T,I_n} \times N \rightarrow V_{T,I_m}$
РАВНЫ:	$V_{T,I_n} \times V_{T,I_n} \rightarrow \text{boolean}$
УНИЧТОЖИТЬ:	$V_{T,I_n} \rightarrow \emptyset$

### 5.3.2 Логическое описание и физическое представление

Поскольку ни в Паскале (ISO 7185), ни в Расширенном Паскале (ISO 10206) нет динамических массивов, то для изложения будет привлечена ныне широко распространенная система Delphi, возможности которой также поддерживаются компилятором **fpc**.

Сначала опишем сам тип вектор. Он должен включать в себя описатель хранимой последовательности и ее длину, т. к. только в Java у каждого массива есть член-функция `Length()`.

```
type Vector = record
  data : array of T;
  size : integer;
end;
```

```
typedef struct
{
  T* data;
  int size;
} Vector;
```

В процессе создания вектора определяется его длина:

```
procedure Create(var v : Vector; sz :  
    integer);  
begin  
    v.size := sz;  
    SetLength(v.data, v.size);  
end;
```

```
void Create(Vector* v, int sz)  
{  
    v->size = sz;  
    v->data = malloc(sizeof(T) * v->size)  
};  
}
```

Проверка на пустоту определяется путем сравнения длины с 0.

Принято считать, что булевского типа в С нет. Это верно, но в заголовке `<stdbool.h>` определены как тип `bool`, так и константы `true` и `false`, полностью моделирующие булевский тип. Нельзя не отметить, что описание и поведение этих констант полностью соответствует типу `bool` языка C++.

```
function Empty(var v : Vector) : boolean;  
begin  
    Empty := v.size = 0;  
end;
```

```
bool Empty(Vector* v)  
{  
    return v->size == 0;  
}
```

Как отмечалось раньше, длина является важной характеристикой вектора. Ее получение является простой операцией, выполняемой за постоянное время.

```
function Size(var v : Vector) : integer;  
begin  
    Size := v.size;  
end;
```

```
int Size(Vector* v)  
{  
    return v->size;  
}
```

Теперь определим операции чтения и записи компоненты по данному индексу. Реализация на Си будет длиннее, поскольку Delphi автоматически проверяет выход за границу массива и возбуждает исключение.

```
function Load(var v : Vector; i : integer)  
    : T;  
begin  
    Load := v.data[i];  
end;
```

```
T Load(Vector* v, int i)  
{  
    if((i >= 0) && (i < v->size))  
        return v->data[i];  
}
```

```
procedure Save(var v : Vector; i : integer  
    ; t : T);  
begin  
    v.data[i] = t;  
end;
```

```
void Save(Vector* v, int i, T t)  
{  
    if((i >= 0) && (i < v->size))  
        v->data[i] = t;  
}
```

Для написания функции изменения размера не придется писать копирование, поскольку библиотечные функции `SetLength` и `realloc` выполняют необходимые действия.

```

procedure Resize(var v : Vector, sz :
    integer);
begin
  v.size := sz;
  SetLength(v.data, v.size);
end;

```

```

void Resize(Vector* v, int sz)
{
  v->size = sz;
  v->data = realloc(v->data, sizeof(T)
    * v->size);
}

```

Отношение равенства векторов играет большую роль в математике, поэтому эта операция просто должна быть определена. Два вектора считаются равными тогда и только тогда, когда равны их длины, а также равны соответствующие компоненты.

```

function Equal(var l, r : Vector) :
  boolean;
var i : integer;
begin
  result := l.size = r.size;
  i := 0;
  while result and (i < l.size) do begin
    result := l.data[i] = r.data[i];
    i := i + 1;
  end;
end;

```

```

bool Equal(Vector* l, Vector* r)
{
  if(l->size != r->size)
    return false;
  for(int i = 0; i < l->size; i++)
    if(l->data[i] != r->data[i])
      return false;
  return true;
}

```

По окончании работы необходимо удалить вектор и вернуть память ОС. В Delphi для этого достаточно присвоить указателю на массив **nil**. В Си необходимо вызвать функцию **free**.

```

procedure Destroy(var v : Vector);
begin
  v.size := 0;
  v.data := nil;
end;

```

```

void Destroy(Vector* v)
{
  v->size = 0;
  free(v->data);
}

```

Приведенная здесь версия вектора проста и понятна, легка в реализации. Более совершенные и изощренные реализации используют сложные механизмы резервирования памяти для ускорения работы путем увеличения непроизводительных издержек.

## 5.4 Очередь

В очередь, сукины дети, в очередь!

*Хриплый крик П. П. Шарикова*

Очереди еще в большей степени, чем файлы встречаются в повседневной жизни. Даже в развитых странах с хорошо налаженным трудом, бытом и отдыхом [72] люди нередко оказываются в различных очередях, страстно требуя уважения основного принципа:

первым пришел — первым ушел. Очередь — стандартное средство сглаживания разницы в производительности процессов, когда поставщик работает быстрее потребителя.

Очередь — это структура с одной читающей головкой и одной записывающей головкой, последовательным доступом и неразрушающей записью [72]. Точнее, очередью называется упорядоченное множество с переменным (возможно, нулевым) числом элементов, на котором определены следующие операции:

- *постановка* в очередь нового элемента;
- *проверка пустоты* очереди;
- *просмотр первого* (самого давнего из имеющихся) элемента, если он есть;
- *извлечение* из очереди ее первого элемента, если очередь не пуста.

Это определение хорошо согласуется с интуитивной концепцией очереди; если не учитывать льготников или просто идущих без очереди. Сначала обслуживается тот, кто прибыл первым и еще не обслужен, он и покидает очередь раньше всех.

Очереди имеют большое значение в информатике; они применяются к двум классам задач:

- для моделирования реальных очередей: современная техника связи прибавила к человеческим очередям целую гамму проблем, касающихся очередей сообщений, поступающих от терминалов, которые соединены с одним или несколькими центрами связи. В процессе компьютерного моделирования исследуется поведение сетей, например, для вычисления максимального или среднего времени ожидания, моделируется реакция сети на случайно приходящие сообщения. Таким же образом можно имитировать появление посетителей в банке и определить число окон, открываемых в различные часы рабочего дня. Очереди захода самолетов на посадку и ожидания разрешения взлета, очереди автомобилей на проезд в узких местах дорожной сети, и т. п. тоже нуждаются в моделировании на ЭВМ;
- решение собственных задач информатики, в частности в области операционных систем ЭВМ. Система имеет дело с целой серией запросов к программным и аппаратным ресурсам: запуск и завершение процессов, доступ к какому-нибудь регистру, устройству или файлу (принтеру, консоли оператора и т. д.). Некоторые типы запросов приоритетны по отношению к другим, но однотипные запросы должны удовлетворяться, вообще говоря, в порядке их поступления.

Необходимо подчеркнуть, что математическая статистика, теория случайных процессов, теория массового обслуживания, теория надежности и базирующаяся на их основе теория очередей, способны дать некоторые теоретические результаты, касающиеся очередей, но зачастую требуют слишком упрощающих гипотез о способах, которыми реализуются входы и выходы. Практически же для получения более реалистических результатов приходится прибегать к моделированию.

Более редким вариантом очереди является дек (Double Ended Queue) — двухконцовная очередь, где чтение и доступ возможны с обоих концов. Ограничив удаление одним концом получают модель очереди с приоритетными льготными элементами. В литературе немного примеров, где применяются деки. Это различные железнодорожные разъезды

и трамвайные депо и один из методов внешней сортировки. Хорошие иллюстрации на тему линейных динамических структур данных (трамвайно-троллейбусного типа) см. в [63]. Но лучшим примером, конечно, является сортировочная станция Дейкстры, обеспечивающая преобразование выражения в обратную польскую запись с его последующим вычислением.

### 5.4.1 Функциональная спецификация.

Итак, очередь — упорядоченный, одномерный, динамически изменяемый набор компонент, в котором включение новых компонент (пополнение очереди) производится на одном конце набора, а всякий доступ к компонентам и их исключение (удаление обработанной компоненты) — на другом конце [43].

Количество компонент очереди называется ее длиной. Однако длина очереди в определении не фиксируется и может быть вычислена «вручную» последовательным перебором всех ее компонент. То есть это неэлементарная составная операция. Более того, для вычисления длины эти элементы придется извлечь из очереди и, если очередь требуется сохранить, поместить их в том же порядке в другую очередь. Ввиду строго последовательного доступа к классической очереди операция вычисления ее длины является довольно дорогой с временной сложностью  $O(N)$  и с такой же пространственной сложностью. Очередь может быть пустой, тогда ее длина равна нулю.

Компонента очереди может иметь любой тип, простой или составной, за исключением файлового типа Паскаля, для которого присваивание не определено. Обозначим тип элемента очереди идентификатором  $T$ .

Тип  $Q_T$  (очередь объектов типа  $T$ ) характеризуется следующими операциями:

СОЗДАТЬ:	$\emptyset \rightarrow Q_T$
ПУСТО:	$Q_T \rightarrow \text{boolean}$
В_ОЧЕРЕДЬ:	$Q_T \times T \rightarrow Q_T$
ИЗ_ОЧЕРЕДИ:	$Q_T \rightarrow Q_T$
ПЕРВЫЙ:	$Q_T \rightarrow T$
ДЛИНА:	$Q_T \rightarrow \mathbb{N}$
УНИЧТОЖИТЬ:	$Q_T \rightarrow \emptyset$

Если очередь находится в обеих частях спецификации, то исходная очередь подвергается модификации и создается скорректированная очередь того же типа  $Q_T$  подобно индуктивному (кумулятивному) присваиванию типа  $x += 1$ .

**Замечание.** Операции ИЗ\_ОЧЕРЕДИ и ПЕРВЫЙ можно рассматривать как единую операцию вида  $Q_T \rightarrow Q_T \times T$ , результатом которой является извлеченная компонента и новая, укороченная очередь.

Перечисленные операции имеют следующие свойства  $\forall t \in T$  и  $\forall q \in Q_T$ :

1. ПУСТО(СОЗДАТЬ) = **true**

СОЗДАТЬ всегда порождает пустую очередь;

2. ПУСТО(В\_ОЧЕРЕДЬ ( $q, t$ )) = **false**

Если добавляется компонента к очереди, то результирующая очередь *заведомо* непуста;

3. ПЕРВЫЙ(В\_ОЧЕРЕДЬ(СОЗДАТЬ, $t$ )) =  $t$ .

Внесенный в пустую очередь элемент всегда становится ее первым элементом.

4. Последовательность действий:

```
q:=СОЗДАТЬ
q:=В_ОЧЕРЕДЬ(q,  $t_1$ )
q:=В_ОЧЕРЕДЬ(q,  $t_2$ )
{ В отличие от файлов перемотка не требуется! }
t:=ПЕРВЫЙ(q) {  $t = t_1$  }
q:=ИЗ_ОЧЕРЕДИ(q)
t:=ПЕРВЫЙ(q) {  $t = t_2$  }
q:=ИЗ_ОЧЕРЕДИ(q)
```

заносящая во вновь созданную пустую очередь сначала элемент  $t_1$ , а затем элемент  $t_2$ , а затем извлекающая из нее два элемента, сохраняет их порядок. Это свойство по индукции может быть распространено на случай любой последовательности элементов  $\{t_1, t_2, \dots, t_n\}$ , даже если эти элементы вставляются в непустую очередь или «разбавлены» какими-то другими элементами, их порядок на выходе из очереди нарушен не будет.

Функциональная спецификация «очереди вообще» имеет не только теоретическое значение. В частности, указанные правила и свойства функциональной спецификации в точности соответствуют тестам для любой конкретной структуры, претендующей на право называться очередью.

### 5.4.2 Логическое описание и физическое представление

Для работы с очередями в какой-нибудь программной среде надо реализовать перечисленные выше операции с их свойствами. В универсальных языках программирования встроенный тип данных очередь, как правило, отсутствует. Для специализированных языков моделирования, таких как GPSS, этот тип данных, наоборот, характерен. Если же надо реализовать очереди на каком-то конкретном универсальном языке программирования, например, на Паскале, то это, скорее всего, придется делать вручную.

Для этого нужно отобразить абстрактную структуру очереди на имеющиеся в Паскале линейные структуры: файл или массив, — либо создать цепочку компонент на динамических структурах [43].

#### 5.4.2.1 Реализация на файле

Назовем тип очереди на файле именем queue:

```
type queue = file of T;
```

Для каждой очереди опишем переменную Паскаля типа очередь:

```
var q : queue;
```

причем для долговременных очередей соответствующий файл должен быть внешним, а имя соответствующей файловой переменной должно быть помещено в заголовок Паскаль-программы:

```
program Queues(input, output, q);
```

Процедура СОЗДАТЬ.

```
procedure Create(var q : queue);
begin
  rewrite(q);
end;
```

В результате мы получим пустой внешний файл, подготовленный для записи компонент очереди. Спецификатор **var** необходим, поскольку файл-параметр без **var** передавался бы по значению, что сводится к копированию файлов, нереализованному в Паскале.

Функция ПУСТО.

```
function Empty(var q : queue) : boolean;
begin
  reset(q);
  Empty := eof(q);
end;
```

Поскольку мы реализуем очередь на файле, пустой очереди должен соответствовать пустой файл. Предиката пустоты файла в Паскале нет, но можно воспользоваться предикатом **eof**(*q*), который совпадает с *Empty*(*q*) после перемотки файла к началу.

Функцию В\_ОЧЕРЕДЬ можно реализовать процедурой *Push*, выполняющей *неперезаписывающую* запись (добавление) компоненты в очередь. Церемония производится с соблюдением всех правил работы с файлами Паскаля.

Для дозаписи добавляемой компоненты в конец файла надо сначала добраться до его конца. Это можно сделать только последовательным чтением всех его компонент, но даже после этого в него нельзя дописать добавляемую компоненту, поскольку в соответствии со стандартом Паскаля операция **reset** устанавливает режим доступа «только чтение». Поэтому дозапись компоненты надо делать в едином сеансе записи, копируя исходный файл-очередь в локальный временный файл *q1* с последующей перезаписью в исходный файл и добавлением компоненты. Локальный файл для копирования создается только на время работы этой процедуры.

```
procedure Push(var q : queue; t : T);
var q1 : queue;
begin
  reset(q);
  rewrite(q1);
  while not eof(q) do begin { q1 := q }
    q1^ := q^;
    get(q);
    put(q1);
  end;
  reset(q1);
  rewrite(q);
  while not eof(q1) do begin { q := q1 || t }
    q^ := q1^;
    get(q1);
```

```

    put(q);
end;
q^ := t;
put(q);
end;

```

Функция ПЕРВЫЙ. В соответствии с выбранным отображением очереди на файл первый элемент очереди всегда находится в начале файла. Заметим, что несмотря на наличие спецификатора **var**, обязательного для файлов в Паскале, очередь не модифицируется, к соответствующему файлу даже не применяется операция чтения, а элемент берется из буфера в памяти  $q \uparrow$ .

```

function Top(var q : queue) : T;
begin
  reset(q);
  Top := q^;
end;

```

Функция ИЗ\_ОЧЕРЕДИ должна обеспечивать *фактическое* исключение выбранной (прочитанной) компоненты из файла, в котором хранится очередь. Альтернативное решение сделать просто **reset**(*q*) и затем пропустить компоненту **get**(*q*) не годится, так как повторное обращение к этой процедуре снова даст эту же компоненту. Для доступа к новому началу очереди (при всех последующих извлечениях компонент из очереди) можно было бы после **reset**(*q*) ( $n - 1$ ) раз выполнять **get**(*q*) и после этого  $x := q \uparrow$ , где  $n$  — номер обращения к чтению из очереди (напомним, что чтение из очереди — разрушающее, прочитанная компонента всегда удаляется). При таком фиктивном удалении необходимо помнить, сколько компонент очереди было удалено ранее и, следовательно, должно быть пропущено при доступе к текущему первому элементу очереди. Это число  $n$  не удается легко агрегировать с файлом, поскольку время жизни файла-очереди, как правило, больше области действия переменной-счетчика удаленных элементов, и существует опасность неправильной интерпретации остающихся в начале файла удаленных из очереди компонент.

Фактическое исключение прочитанной компоненты можно обеспечить более трудоемким способом, используя вспомогательный локальный файл. В него надо скопировать все компоненты файла-очереди, начиная со второй, с последующей переписью из вспомогательного файла обратно в файл-очередь.

Процедура Pop выталкивает элемент из очереди. Если надо сохранить извлеченный элемент, необходимо предварительно получить его при помощи функции Top и поместить в локальную переменную.

```

procedure Pop(var q : queue);
var q1 : queue; { Локальный рабочий файл для временного хранения копии очереди
  без первого элемента. В стиле тезиса «самый маленький файл больше самого
  большого массива» мы полагаем, что на устройстве внешней памяти всегда
  есть место еще для одного файла }
begin
  reset(q);
  rewrite(q1);

```

```

if not eof(q) then get(q); { Пропускаем первый элемент, если он есть. Будет
    работать для пустой очереди, но лучше явно проверять пустоту
    предикатом Empty перед вызовом Pop }
{ Копируем остаток очереди q (без первого элемента) в очередь q1. Для пустой
    очереди этот цикл тоже работает }
while not eof(q) do begin
    q1^ := q^;
    put(q1);
    get(q);
end;
{ Копируем скорректированную очередь q1 из временного локального файла (без
    первого элемента) назад, в постоянный внешний глобальный файл q }
reset(q1);
rewrite(q);
while not eof(q1) do begin
    q^ := q1^;
    put(q);
    get(q1);
end;
end;

```

К сожалению, замечательный Borland Pascal (в нарушение стандарта Паскаля!) не реализует процедуры **get** и **put** и эти программы должны быть переписаны с помощью процедур **read** и **write**.

Итак, очередь можно отобразить на файл, но для работы с ней потребуются дополнительные ресурсы: во-первых, время на перемотку файла из конца в конец при попеременном выполнении операций В\_ОЧЕРЕДЬ и ИЗ\_ОЧЕРЕДИ, во-вторых, память и время на выполнение операции ИЗ\_ОЧЕРЕДИ. Кроме того, необходимо помнить, что, в отличие от винчестера, имеющего воздушную смазку головок, реальные магнитные ленты выдерживают всего несколько сотен перемоток, поскольку головка НМЛ непосредственно контактирует с лентой. С другой стороны, CD-ROM, использующий для считывания данных луч лазера, еще более бесконтактен, чем винчестер. В заключение заметим, что сходство очередей и файлов особенно заметно во французском языке: там эти понятия обозначаются одним и тем же словом **queue** [72]. Слово файл — английского происхождения.

#### 5.4.2.2 Реализация на массиве

При реализации очереди на массиве необходимо зафиксировать размер этого массива, ограничив тем самым максимальную длину очереди. В каждый конкретный момент времени очередь будет представлена сплошным участком массива. Введем две индексные переменные: первую — для индекса начала очереди («голова»), вторую — для индекса первого свободного элемента массива после последнего элемента очереди («хвост»; наш хвост не является частью тела). Рассмотрим три стратегии отображения очереди на массив:

- *стратегия трудоголика*: никогда не откладывай на завтра то, что можно сделать сегодня. При такой стратегии голова очереди фиксируется на первом элементе

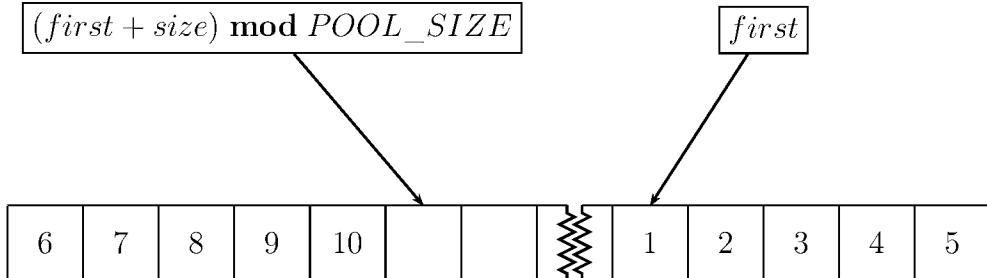
массива. Хвост очереди подвижен, поэтому время постановки элемента в очередь постоянно и невелико ( $O(1)$ ). При извлечении элемента из очереди оставшиеся компоненты сдвигаются к началу массива. Именно так ведут себя люди или автомобили, находящиеся в реальных очередях: хотя каждый участник очереди делает только один шаг вперед, цена этой операции для всей очереди довольно велика и пропорциональна ее длине  $O(N)$ . «Применяется: никогда!» (Д. В. Сошников).

- при *ленивой стратегии* и голова, и хвост очереди подвижны. Ленивая очередь не движется до тех пор, пока это возможно, но, если надо, сдвигается по максимуму. Извлечение элемента из очереди занимает постоянное время, т. к. необходимо всего лишь инкрементировать индексную переменную — головной указатель. Постановка в очередь осуществляется так: если хвост указывает на свободный элемент массива, то туда записывается новая компонента очереди. В противном случае, когда хвост упирается в конец массива, делается попытка воспользоваться свободным местом в начале массива, освобожденным ушедшими из очереди элементами (примитивная сборка мусора). В случае успеха вся очередь копируется в начало массива, после чего и происходит собственно добавление элемента. При малой длине очереди, много меньшей размера массива, постановка в очередь в среднем занимает постоянное время, т. к. на одно копирование происходит много добавлений. Если же длина очереди близка к размеру массива, то сдвиг происходит почти при каждой постановке в очередь, и время выполнения этой операции приближается к  $O(N)$  [43]. Хотя в этой стратегии что-то есть, грамотные программисты ее также не используют.
- *стратегия с кольцевым буфером* обобщает ленивую стратегию. Перемещения очереди в массиве не бывает никогда. Склейвая начало массива с его концом (с помощью знакомой нам еще со времен Цезаря операции **mod**, отбрасывающей нас от конца массива к его началу, мы получаем возможность неограниченного поступательного движения вперед по буферу, при условии, конечно, что его элементы так же быстро исчезают, как и появляются). По гениальности идея кольцевого буфера может соперничать с предложенными проф. Жоголевым Е. А. в 1958 г. встречными стеками. При использовании кольцевого буфера как только одна из переменных-индексов подходит к концу массива, ее скачкообразно переустанавливают на начало массива, на освобожденное ушедшими из очереди элементами место. При такой стратегии и постановка, и извлечение из очереди занимают постоянное и очень небольшое время  $O(1)$  [59, 54]. В отличие от предыдущих схем кольцевая буферизация является «экологически чистой» и не требует сборки мусора. И сама очередь, и свободная память образуют *сплошные* куски буферного массива (по модулю  $N$ ), между началом и концом которых мигрируют освобождаемые и занимаемые элементы.

Как видно, из всех стратегий оптимальной является стратегия с кольцевым буфером, поскольку обе операции занимают минимально возможное время. Может быть такая реализация немного сложнее остальных, но ее характеристики, несомненно, того стоят.



Итак, зафиксируем размер массива-буфера и обозначем его *POOL\_SIZE*. Пусть в переменной *first* хранится индекс головы очереди, в переменной *size* — ее длина. Зная это, легко указать на хвост очереди.



Заметим, что более очевидная система координат кольцевого буфера *first* и *last* не была принята нами в связи с невозможностью различить ситуации пустого и до отказа заполненного буферов.

Таким образом, тип очередь представляется в виде комбинированной структуры с полями *first*, *size* и *data*.

```
const POOL_SIZE = 100;

type queue = record
    first : 0..POOL_SIZE - 1;
    size : 0..POOL_SIZE;
    data : array[0..POOL_SIZE - 1] of T
;
end;
```

```
const int POOL_SIZE = 100;

typedef struct
{
    int first ;
    int size ;
    T data[POOL_SIZE];
} queue;
```

Тип переменных *first* и *size* в Паскаль-реализации сделан интервальным вместо целого для повышения надежности (как в период компиляции, когда будут недопустимы константы за пределами указанного диапазона, так и в период выполнения, облегчая скомпилированные проверки принадлежности текущих значений интервалу типа).

Поскольку очередь реализована на статическом массиве, создание очереди сводится к инициализации параметров доступа к нему. При необходимости действительно динамического создания очереди надо воспользоваться процедурой **new** (или функцией **malloc()**).

При инициализации очереди необходимо сделать ее пустой, установив в нуль ее размер. Заметим, что начальное значение переменной-индекса головы очереди не обязательно должно быть нулем: кольцевой буфер может начинаться в любом месте массива.

```
procedure Create(var q : queue);
begin
    q. first := 0;
    q. size := 0;
end;
```

```
void Create(queue* q)
{
    q->first = 0;
    q->size = 0;
}
```

Проверка пустоты очереди осуществляется путем сравнения ее длины с нулем.

```

function Empty(var q : queue) :
    boolean;
begin
    Empty := q.size = 0;
end;

```

```

bool Empty(const queue* q)
{
    return q->size == 0;
}

```

Размер очереди уже хранится в ней в виде отдельной переменной, поэтому достаточно лишь вернуть соответствующее значение.

```

function Size(var q : queue) : integer;
begin
    Size := q.size ;
end;

```

```

int Size(const queue* q)
{
    return q->size;
}

```

В функциях `Empty()` и `Size()` очередь передается по ссылке, с указанием спецификатора `var`, стандартного для массивов Паскаля. В Си массивы всегда передаются по ссылке, что изображается символом \* или [] . С теоретической точки зрения это нехорошо, т. к. параметры функции должны быть константами, а единственным результатом такой подпрограммы должно быть ее значение. Описатель `var` к тому же и небезопасен: для функций,читывающих характеристики очереди, он предоставляет возможность модификации всех ее компонент. К счастью, в Си мы можем добавить модификатор доступа `const`, запретив тем самым всякое изменение компонент очереди.

Операция постановки в очередь реализуется функцией двух аргументов `Push()`, которая возвращает истинностное значение: `true`, если постановка в очередь была проведена успешно и `false` в противном случае. Единственная причина, по которой функция `Push()` может вернуть `false` — попытка записать в массив *data* элементов больше, чем его размер `POOL_SIZE`.

Теперь вспомним о *first*. Запись новой компоненты в массив кольцевого буфера производится в свободный элемент, индекс которого вычисляется по формуле (*first*+*size*) **mod** `POOL_SIZE`, происходит так, что выхода за границу массива не возникает. Поскольку операция взятия остатка от деления однозначно определена лишь для неотрицательных чисел (подробнее см. [76]), то *first* должен быть неотрицательным числом.

```

function Push(var q : queue; t : T) :
    boolean;
begin
    with q do begin
        { Если буфер полон, вставка
        невозможна }
        if (size = POOL_SIZE) then
            Push := false
        else begin
            { Операция mod обеспечивает
            закольцованность буфера }
            data[( first + size) mod
                  POOL_SIZE] := t;
    end;
end;

```

```

bool Push(queue* q, const T t)
{
    if(q->size == POOL_SIZE)
        return false;
    // Оператор взятия остатка %
    // обеспечивает закольцованность
    // буфера
    q->data[(q->first + q->size++) %
              POOL_SIZE] = t;
    return true;
}

```

```

    size := size + 1;
    Push := true
  end
end
end;

```

Операция извлечения из очереди *Pop* также возвращает истинностное значение, указывающее на успех операции. Единственная причина неуспеха — это отсутствие компонент в очереди.

После успешного применения этой операции переменная *first* принимает некоторое значение из полуинтервала [0; POOL\_SIZE).

```

function Pop(var q : queue) : boolean;
begin
  with q do begin
    if (size = 0) then
      Pop := false
    else begin
      first := (first + 1) mod
        POOL_SIZE;
      size := size - 1;
      Pop := true;
    end;
  end;
end;

```

```

bool Pop(queue* q)
{
  if (!q->size)
    return false;
  q->first++;
  q->first %= POOL_SIZE;
  q->size--;
  return true;
}

```

Первый элемент очереди доступен, если очередь непуста. Функция *Top* проверяет непустоту очереди и возвращает ее первый элемент, если он есть. В противном случае значение функции *Top* не определено.

Эта операция может быть применена в любой момент, когда очередь непуста. Чтобы гарантировать отсутствие ошибок, *first* должен находиться в полуинтервале [0; POOL\_SIZE).

```

function Top(var q : queue) : T;
begin
  if (q.size <> 0) then
    Top := q.data[q.first];
end;

```

```

// В STL такая функция называется
// front()
T Top(const queue* q)
{
  if (q->size)
    return q->data[q->first];
}

```

Теперь вернемся к вопросу о начальном значений переменной *first*. Функции *Push()* и *Pop()* с целью обеспечения корректности применения мультипликативной операции **mod** ограничивают *first* неотрицательными значениями. Поскольку *Top()* не производит никаких проверок, то возникает более строгое ограничение:  $first \in [0; POOL\_SIZE]$ , т. е. начальные значения переменной должны быть в этом диапазоне.

В отличие от описанного ранее типа вектор и типа список, который будет описан позже, специально уничтожать очередь на статическом массиве не требуется, и эта функция ничего не делает. Но если возникнет потребность перейти от очереди на массиве к очереди на динамических структурах (в том числе к очереди на динамическом массиве), которую необходимо специально уничтожать, возвращая память ОС, то не придется дописывать уничтожение очередей в конец каждого блока, где эта очередь появилась. Эта унификация полезна и при обратном переходе к очереди на массиве.

```
procedure Destroy(var q : queue);
begin
  { Чтобы воспрепятствовать
    использованию уничтоженной
    очереди, можно положить q.size
    := 0; }
end;
```

```
void Destroy(queue* q)
{
  // Чтобы воспрепятствовать
  // использованию уничтоженной
  // очереди, можно положить q->
  size = 0;
}
```

Поскольку очередь реализована на статическом массиве, нет возможности вернуть ресурсы, занимаемые ею. Можно очистить уничтожаемую очередь от элементов, установив *size* равным 0.

#### 5.4.2.3 Реализация на динамических структурах

Связем ссылками одиночные элементы очереди в цепочку в том порядке, в котором они должны находиться в очереди. Для этого заведем комбинированный тип «элемент очереди», где наряду с полем данных предусмотрим переменную-ссылку на следующий элемент того же типа. Обратите внимание на рекурсивность соответствующего описания. Она позволяет задавать самоподобные (фрактальные) структуры, части которых имеют тот же самый тип. Заметим, что рекурсивные описания в Паскале должны быть помещены в одном предложении **type**.

```
type Item = record
  data : T;
  next : PItem;
end;
PItem = ^Item;
```

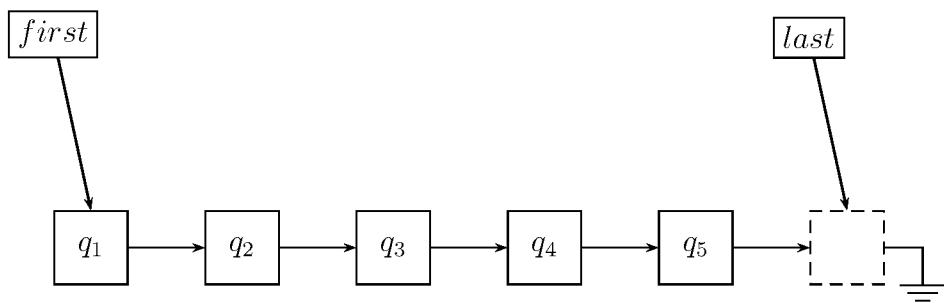
```
struct Item
{
  T data;
  struct Item* next;
};
```

Реализуем тип очередь на динамических структурах. Во-первых, понадобится ссылка на начало очереди. Чтобы обеспечить выполнение операции постановки в очередь за постоянное время, необходимо дополнительно завести ссылку на конец очереди. В противном случае цена вставки элемента в очередь повышается до неприемлемого  $O(N)$ . Чтобы уделешевить подсчет длины очереди с  $O(N)$  до  $O(1)$  применим неклассический подход, включив в представление очереди переменную *size* — длину очереди; актуализация ее значения возлагается на операции модификации.

При реализации очереди на массиве хвост указывал на *первый свободный элемент массива за концом очереди*. Здесь мы поступим аналогично, введя пустой служебный элемент — *терминатор*. Для его хранения потребуется дополнительный элемент, но это оправдано простотой реализаций всех функций для работы с очередью. Терминатор

очереди не хранит никакого значения и не может быть прочитан из очереди либо разыменован для получения хранимого элемента. Поэтому важен не сам терминатор, а ссылка на него, которая завершает очередь.

Необходимо заметить, что графическая иллюстрация структур данных является особенно полезной для динамических ссылочных объектов. При этом ссылки изображаются стрелками, связывающими ссылочные переменные и указываемые значения. Сами ссылочные переменные изображаются малыми прямоугольниками, из которых исходят стрелки. Указываемые объекты изображаются более крупными прямоугольниками, обычно разграфленными на поля компонент. Стрелки для пустых ссылок загибаются вниз и заканчиваются знаками заземления.



**Замечание.** Непроизводительные затраты на терминатор могут быть существенными, если размер элемента типа достаточно велик (очередь матриц на транспонирование). В этом случае отказываются от представления терминатора тем же типом, что и элементы очереди, и описывают его специальным типом, единственным требованием к которому является ненулевая длина (т. к. невозможно разместить в памяти объект нулевой длины). При этом придется отказаться от типизированных указателей, поскольку они не позволяют ссылаться на элементы разных типов. Бестиповой указатель *next*, в зависимости от положения в очереди, будет указывать либо на ее элемент, либо на терминатор.

Более совершенное решение, используемое, в частности, в STL, будет описано в разделе, посвященном полиморфизму.

Теперь опишем тип данных для очереди на динамических структурах. Ввиду динамизма этой цепной структуры фиксируются только начальная и конечная ссылки. Конкретная конфигурация очереди всякий раз должна интерпретироваться относительно этих указателей. Поскольку длина очереди на динамических структурах потенциально бесконечна и не имеет явного ограничения, в качестве типа данных для ее размера *size* взят **integer**. Любители защитного программирования должны описать тип *cardinal* = *0..MAXINT* для переменной *size*.

```

type queue = record
  first : PItem;
  last : PItem;
  size : integer;
end;
  
```

```

typedef struct
{
  struct Item* first ;
  struct Item* last;
  int size ;
} queue;
  
```

Функция создания очереди выделяет память под терминирующий элемент. Правоассоциативность оператора присваивания в Си, и тот факт, что этот оператор возвращает ссылку-сионим типа  $T\&$ , позволяет записать всю инициализацию одной строкой.

В отличие от очереди на массиве это действительно *создание* очереди, а не просто инициализация статической структуры.

```
procedure Create(var q : queue);
begin
  new(q.first);
  q.last := q.first;
  q.size := 0;
end;
```

```
void Create(queue* q)
{
  q->first = q->last = malloc(sizeof(
    struct Item));
  q->size = 0;
}
```

Если головой очереди является терминатор, то она пуста, хотя в данной реализации это можно также проверить сравнением размера очереди с нулем.

```
function Empty(var q : queue) :
  boolean;
begin
  Empty := q.first = q.last;
end;
```

```
bool Empty(const queue* q)
{
  return q->first == q->last;
}
```

Размер очереди хранится в явном виде, функция *Size()* лишь возвращает его.

```
function Size(var q : queue) : integer;
begin
  Size := q.size;
end;
```

```
int Size(const queue* q)
{
  return q->size;
}
```

Функция постановки в очередь *Push()* выдает запрос на выделение памяти языковой среде или непосредственно ядру операционной системы. Если память не была выделена, то указателю присваивается нулевое значение (в Си) или **nil** в Паскале, и функция возвращает **false**. Если все операции были проведены успешно, то значение нового элемента заносится в терминатор очереди и указатель на хвост сдвигается к новому созданному элементу, который становится терминатором.

```
function Push(var q : queue; t : T) : boolean;
begin
  new(q.last^.next); { Создание нового
    терминатора }
  if (q.last^.next = nil) then
    Push := false { Память не выделена, Push
      () не выполнен }
  else begin
    q.last^.data := t; { Заполнение поля
      данных элемента }
  end;
end;
```

```
bool Push(queue* q, const T t)
{
  if (!(q->last->next = malloc(
    sizeof(struct Item))))
    return false;
  q->last->data = t;
  q->last = q->last->next;
  q->size++;
  return true;
}
```

```

q.last := q.last ^ .next; { Сдвигаем хвост
очереди путем переназначения
терминатора на вновь созданный
элемент }

q.size := q.size + 1; { Корректировка
длины очереди после успешного
добавления элемента }

{ Предыдущие два оператора
индуктивного присваивания
фактически дают нам следующие по
порядку элементы (succ(q.last) и succ(
q.size) для двух разных упорядоченных
множеств }

Push := true;
end;
end;

```

Возврат булевского значения функции *Push()* — чистая прагматика, попытка реального программирования. В теории *Push()* должен аварийно завершать программу при нехватке памяти в куче. В современных языках программирования в таких случаях возбуждаются исключительные ситуации.

**Замечание.** Если терминатор был бы описан отдельным, более компактным типом, то указатель на хвост очереди пришлось бы зафиксировать, а вставку нового элемента всегда производить *перед* ним. В нашем случае терминатор однотипен элементу очереди и используется для размещения вновь пришедшего элемента, и его приходится всякий раз порождать заново. При использовании терминатора специального типа необходимо соответствующим образом переделать функции *Create()*, *Push()* и *Destroy()*.

Функция *Pop()*, копируя указатель на первый элемент очереди, готовит этот элемент к отсоединению от очереди, переустановливая соответствующую ссылку на следующий элемент, с последующим уничтожением отсоединеной компоненты по копии ссылки. При этом также декрементируется счетчик элементов очереди.

```

function Pop(var q : queue) : boolean;
var pi : PItem;
begin
  if (q.first = q.last) then
    Pop := false { Очередь пуста, извлечение
      невозможно }
  else begin { Очередь непуста }
    pi := q.first ; { Копируем указатель на
      начало очереди }
    q.first := q.first ^ .next; { Перемещаем
      указатель начала очереди на новый
      первый элемент или на терминатор,
      если извлечен единственный элемент
      очереди }
  end;
end;

```

```

bool Pop(queue* q)
{
  if (q->first == q->last)
    return false;
  struct Item* pi = q->first;
  q->first = q->first->next;
  q->size--;
  free(pi);
  return true;
}

```

```

q.size := q.size - 1; { Корректировка
    размера очереди }
dispose(pi); { Освобождение памяти,
    занимаемой удаляемым элементом }
Pop := true;
end;
end;

```

Как и при реализации очереди на массиве, функция *Top()* определена только на непустых очередях.

```

function Top(var q : queue) : T;
begin
  if(q.first <> q.last) then
    Top := q.first^.data;
  end;

```

Для уничтожения очереди надо *поочередно* удалить все ее элементы, сохраняя в локальной переменной *pi* (на мгновение!) перед удалением каждого элемента содержащуюся в нем ссылку на следующий элемент.

```

procedure Destroy(var q : queue);
var pi : PItem;
begin
  while(not Empty(q)) do begin { Или:
    q.first <> q.last == пока
    терминатор не станет головой
    очереди }
    pi := q.first ;
    q.first := q.first^.next;
    dispose(pi); { Освобождаем
      память очередного элемента
      очереди }
  end;
  dispose(q.first ); { Уничтожаем
    терминатор, удаляя
    опустошенную очередь }
  q.first := nil; { Уничтожаем
    ссылки на уничтоженный объект
    }
  q.last := nil;
  q.size := 0;
end;

```

Эта программа совмещает *обход* очереди и посещение *всех* ее элементов с одновременным удалением пройденных компонент.

```

T Top(const queue* q)
{
  if(q->first != q->last)
    return q->first->data;
}

```

```

void Destroy(queue* q)
{
  while(!Empty(q))
  {
    struct Item* pi = q->first;
    q->first = q->first->next;
    free(pi);
  }
  free(q->first);
  q->first = q->last = 0;
  q->size = 0;
}

```

Воспользуемся рекурсивностью определения очереди для ее рекурсивного уничтожения:

```

procedure Destroy(var q : queue);
var pi : PItem;
begin { Уничтожение очереди сводится к
    удалению первого элемента и
    уничтожению остатка, который тоже
    является очередью }
{ Для остановки рекурсии по исчерпании
    очереди установим неиспользуемую
    ссылку «вперед» из терминатора в nil }
q.last^.next := nil; { Эта операция
    правомерна, т. к. в очереди есть по
    крайней мере термирующий элемент
}
pi := q.first ;
q.first := q.first^.next;
dispose(pi);
if(q.first <> nil) then { Условие
    прекращения рекурсии; выполнимо,
    поскольку длина конечна и уменьшается
    на 1 при каждом вызове функции }
{ Здесь процедуре Destroy() подается
    очередная обезглавленная очередь }
Destroy(q);
{ К моменту возврата из всех
    иницированных удалений подочередей
    очередь уничтожена полностью,
    включая терминатор }
q.last := nil ;
q.size := 0;
end;

```

```

void Destroy(queue* q)
{
    q->last->next = 0;
    struct Item* pi = q->first;
    q->first = q->first->next;
    free(pi);
    if(q->first)
        Destroy(q);
    q->last = 0;
    q->size = 0;
}

```

Рекурсивная версия процедуры уничтожения очереди математически прямолинейна (кипячение чайника математиком) и подобна итеративной, но, помимо затрат ресурсов на рекурсию, требует специальных знаний и культуры программирования.

Говоря о логическом описании очереди, нельзя не упомянуть ее реализацию в стандартной библиотеке шаблонов STL C++ (ISO/IEC 14882). Как писал Дж. Бэкус в своей Тьюринговской лекции [26], язык должен иметь небольшое ядро и расширяться за счет изменяемой части — в данном случае воплощенной в виде библиотечных функций. Ниже приводится фрагмент функциональной спецификации типа очередь из Стандарта C++ (с. 479–480):

```

namespace std
{

```

```

template <class T, class Container = deque<T> >
class queue
{
public:
    explicit queue(const Container& = Container());
    bool empty() const;
    size_type size() const;
    reference front();
    const_reference front() const;
    reference back();
    const_reference back() const;
    void push(const value_type&);
    void pop();
};

template <class T, class Container>
bool operator==(const queue<T, Container>&, const queue<T, Container>&);

template <class T, class Container>
bool operator<(const queue<T, Container>&, const queue<T, Container>&);

template <class T, class Container>
bool operator!=(const queue<T, Container>&, const queue<T, Container>&);

template <class T, class Container>
bool operator>(const queue<T, Container>&, const queue<T, Container>&);

template <class T, class Container>
bool operator<=(const queue<T, Container>&, const queue<T, Container>&);

template <class T, class Container>
bool operator>=(const queue<T, Container>&, const queue<T, Container>&);

}

```

Не надо быть знатоком C++, чтобы понять: данное описание полностью соответствует вышеописанной функциональной спецификации.

Теперь рассмотрим пример реверсирования очереди (перестановки элементов в обратном порядке) с использованием вышеописанных функций (благодаря единому интерфейсу различных реализаций очередь программа может быть использована с любой из них):

|

```

procedure Reverse(var q : queue);
var t : T; { В этой локальной переменной будет
    храниться очередная голова постепенно
    обезглавливаемой очереди. При N рекурсивных
    вызовах этой процедуры будет создано N
    экземпляров этой переменной, причем новизна

```

```

void Reverse(queue* q)
{
    if (!Empty(q))
    {
        T t = Top(q);
        Pop(q);

```

*этих значений будет обратна времени их создания. Исчерпав очередь в эти переменные при отработке возвратов, мы помещаем элементы из последовательности экземпляров переменной  $t$  в очередь  $q$  в порядке, обратном их извлечению, поскольку это делается при возвратах из рекурсии }*

Reverse( $q$ );  
Push( $q, t$ );

}

```

begin
  if not Empty( $q$ ) then begin
     $t := \text{Top}(q);$ 
     $\text{Pop}(q);$ 
    { Реверсируем очередь без очередного первого
      элемента }
    Reverse( $q$ );
    { Возвращаем в очередь элементы, изъятые
      перед входом в рекурсию }
    Push( $q, t$ );
  end
end;

```

*Замечание.* Экземпляры переменной  $t$  образуют своеобразный временной стек, в который в обратном порядке заносятся головы подочередей. С этим инверсирующим свойством стека мы познакомимся позднее.

## 5.5 Стек

Стековые структуры и дисциплины доступа нередко встречаются в повседневной жизни. Бюрократ складывает бумаги на стол в пачку — своеобразный стек — так, что пришедшие первыми документы лежат месяцами, прежде чем выберутся на поверхность... Пришедший первым уходит последним — определяющий принцип такой структуры данных [49, 63, 59].

Стек — это структура с единственной читающей-записывающей головкой, последовательным доступом и неразрушающей записью. Чтение из стека, как и у очереди, разрушающее (удаление!). Более строго, стеком называется множество некоторого переменного (возможно, нулевого) числа данных, на котором выполняются следующие операции:

- *пополнение* стека новыми данными;
- *проверка*, определяющая, пуст ли стек;
- *просмотр верхнего* (самого нового) элемента, если он существует;
- *уничтожение последнего* добавленного, но еще не уничтоженного элемента, если он есть.

Другой пример — пачка книг, из которой легко снимать и класть книгу только сверху.

Уровни протоколов TCP/IP также принято складывать в стек, закрывая низкоуровневые протоколы высокоуровневыми в строго указанном порядке.

В информатике стеки используются очень часто. Во-первых, для изучения физических процессов, в которых встречаются настоящие стеки. Например, при продаже творожных сырков в супермаркете, их удобно подкладывать в охлаждаемый прилавок сверху; сверху их берут и покупатели. В результате, снизу залеживаются более старые сырки, срок хранения которых истекает и их приходится выбрасывать [72]. Пополнение сырков не должно идти слишком часто, так как это не только лишняя работа продавцов, но и транспортные и тяжелажные расходы.

Стек позволит промоделировать и оптимизировать последовательность добавлений и извлечений сырков из стопки; частота покупок в разное время дня может быть определена экспериментально. Более сложная модель может учесть, что некоторые покупатели не берут верхний сырок. Таким образом, чтобы лучше организовать работу секции, надо смоделировать поведение одного или нескольких стеков, элементы которых характеризуются предельным сроком хранения. Полагая известной повременную выборку сырков, надо определить идеальную последовательность пополнения, такую, что:

1. стек никогда не становится пустым;
2. элементы, находящиеся на дне стека, не остаются в стеке после предельной даты;
3. частота пополнения не слишком велика.

Практическое решение представляет некоторый разумный компромисс между этими целями.

Стеки весьма полезны при решении различных собственных задач информатики:

- синтаксический анализ текста, выполняемый трансляторами с языков высокого уровня, существенно использует стеки; вложенная структура блоков, процедур, выражений и описаний такова, что омонимичные переменные образуют стеки;
- в стеке удобно отводить память под локальные переменные программных единиц блочной структуры;
- выполнение рекурсивной процедуры также организуется с помощью стека; вложенный вызов может породить еще один, который завершается до того, как управление возвращается на породивший его уровень. В данном случае в стек откладываются копии множеств локальных переменных предыдущих незавершенных вызовов одной и той же процедуры;
- некоторые ЭВМ (Burroughs, Эльбрус, ICL), микропроцессоры (многие), калькуляторы и языки программирования имеют существенно стековую архитектуру; другие — только обеспечивают программную (шаблон std::stack в C++) или аппаратную поддержку стеков (ЭВМ PDP, VAX и другие). Машины со стековой нуль-адресной системой команд имеют в 3–4 раза более короткие программы за счет экономии на адресах операндов, почти как и у наиболее изощренных представителей CISC архитектур (VAX), в сочетании с простотой RISC процессоров. Это же свойство стековой архитектуры позволило успешно использовать язык ФОРТ для программирования самых примитивных и маломощных микропроцессоров.

Разновидностью стека является *магазин* винтовки, автомата или пулемета; патроны засылаются туда в порядке, обратном последующей стрельбе. Таким образом, согласно известной шутке студентов ВМК, автомат Калашникова действительно является устройством для преобразования стека в очередь [85], с. 194. Стрелок на привале берет патроны из коробки: 1, 2, ..., 30, заталкивая их в магазин в том же порядке: 1, 2, ..., 30. В результате сверху оказывается 30-ый патрон, а первый патрон, конечно же, внизу. Присоединяя магазин к автомату и передергивая затвор, стрелок досыпает верхний 30-ый патрон в патронник и все патроны поднимаются к верху магазина с сохранением порядка следования. При автоматической стрельбе (*очередями!*) патроны поочередно поступают в патронник из магазина и выстреливаются через ствол в порядке, обратном их закладке в магазин: 30, 29, ..., 1. То есть помещение в стек инвертирует порядок следования элементов. Выходная очередь обратна входной очереди патронов, заносимых в магазин.

Отличие магазина от стека, видимо, состоит в том, что стек имеет неподвижное дно и плавающую верхушку, а магазин — фиксированную верхушку и бездонный колодец, куда в идеале можно заталкивать сколь угодно много элементов (бесконечная гостиница в фантастическом романе Ивана Ефремова «Туманность Андромеды»).

Используемые на практике стеки не всегда подчиняются классической функциональной спецификации. Так, для доступа к глобальным переменным транслятору может потребоваться неверхушечное чтение (так называемое чтение по стрелке из ранее засланной в стек таблицы описаний объемлющего блока). Такой стек становится полумассивом и эффективно реализуется именно в сплошном представлении. В связи с этим вспомним, что стек использовался для распределения памяти для программ с вложенной блочной структурой, в том числе для организации рекурсивных вызовов. В случае необходимости быстрого возврата из глубины рекурсии, минуя промежуточные стадии, для пропуска соответствующего участка стека в стандартной библиотеке языка C/C++ имеются «нелинейные» функции работы с неявным стеком вызовов `setjmp()` и `longjmp()`. Такие функции могут быть реализованы для явных стеков.

Заметим, что терминология стеков «вертикальная» (пачки, стопки и т. д.), в то время как очереди были нами вербализованы «горизонтально». Более того, в английском языке так и говорят: «положить на стек» и «снять со стека». В русском языке стек — это магазин, и элементы «заталкивают в стек» или «извлекают из него» [85].

### 5.5.1 Функциональная спецификация

Тип  $S_T$  или стек объектов типа  $T$ , характеризуется операциями [72]:

СОЗДАТЬ:	$\emptyset \rightarrow S_T$
ПУСТО:	$S_T \rightarrow \{\text{true}, \text{false}\}$
ГЛУБИНА:	$S_T \rightarrow \mathbb{N}$
В_СТЕК:	$S_T \times T \rightarrow S_T$
ИЗ_СТЕКА:	$S_T \rightarrow S_T$
ВЕРХ:	$S_T \rightarrow T$
УНИЧТОЖИТЬ:	$S_T \rightarrow \emptyset$

*Замечание.* Функции ВЕРХ и ИЗ\_СТЕКА можно объединить в единую операцию

со спецификацией

$$S_T \rightarrow T \times S_T$$

результатом которой является и элемент, и измененный стек.

Для всякого элемента  $t$  типа  $T$  и всякого  $s$  типа  $S_T$  имеют место следующие свойства:

1. ПУСТО(СОЗДАТЬ) = **true** // созданный стек пуст
2. ПУСТО(B\_СТЕК( $s, t$ )) = **false** // занесение элемента в стек делает его заведомо непустым
3. ВЕРХ(B\_СТЕК( $s, t$ )) =  $t$

Только что занесенный в стек элемент становится его верхушкой.

4. **if not** ПУСТО( $s$ ) **then**  
 $t :=$  ВЕРХ( $s$ );  
 $s_1 :=$  ИЗ\_СТЕКА( $s$ );  
B\_СТЕК( $s_1, t$ ) =  $s$ ;  
**end;** /\* после выборки элемента из верхушки стека с последующим его возвращением в стек получается исходный стек \*/

5. Последовательность действий:

```
 $s :=$  СОЗДАТЬ  
 $s :=$  B_СТЕК( $s, t_1$ )  
 $s :=$  B_СТЕК( $s, t_2$ )  
 $t :=$  ВЕРХ( $s$ ) //  $t = t_2!$   
 $s :=$  ИЗ_СТЕКА( $s$ )  
 $t :=$  ВЕРХ( $s$ ) //  $t = t_1!$   
 $s :=$  ИЗ_СТЕКА( $s$ )
```

заносящая во вновь созданный пустой стек сначала элемент  $t_1$ , а затем элемент  $t_2$ , а затем извлекающая из нее два элемента, инвертирует их порядок. Это свойство по индукции может быть распространено (транзитивное замыкание) на случай любой последовательности элементов  $\{t_1, t_2, \dots, t_n\}$ , даже если эти элементы вставляются в непустой стек или «разбавлены» какими-то другими элементами, их строго обратный порядок на выходе из стека нарушен не будет.

**Замечание.** Требование непустоты стека ограничивает использование некоторых операций и свойств этих операций, блокируя соответствующие выводы из ложных посылок, но мы не стали усложнять функциональную спецификацию стека рассмотрением этих случаев.

**Замечание.** Обратите внимание, что функциональная спецификация стека отличается от функциональной спецификации очереди только последними свойствами (LIFO и инверсия).

## 5.5.2 Логическое описание

Самый свежий элемент стека, т. е. последний введенный и еще не уничтоженный играет особую роль; именно его можно рассмотреть или уничтожить. Этот элемент называется верхушкой стека. Оставшаяся часть можно назвать телом стека и оно само является, по существу, стеком; если снять со стека верхушку, то тело превращается в стек. Таким образом, стек, как и очередь, является рекурсивной структурой данных и может быть описан рекурсивно как конкатенация элемента типа  $T$  и некоторого стека  $s$  типа  $S_T$ . Для обеспечения эффективной вычислимости необходим терминатор, ограничивающий глубину рекурсии описания, роль которого, как в файлах и очередях, играет пустой стек [72]:

```
type СТЕК-Т = (ПУСТО | НЕПУСТОЙСТЕК)
```

```
type НЕПУСТОЙСТЕК = ( верхушка: Т; тело: СТЕК-Т)
```

Заметим, что рекурсивность описания может быть не только по глубине стека, но и по сложности типа его компонент  $T$ : этот тип может быть произвольным, в том числе и рекурсивным, и даже стеком...

### 5.5.2.1 Язык программирования со стековой моделью вычислений

Стек как структура данных является базовой для такого своеобразного языка программирования, как Форт. Как мы увидим позднее, стек оказывается чрезвычайно удобным для вычисления и представления арифметических выражений. В языке Форт используется так называемая *постфиксная* (*обратная польская*, названная так в честь Я. Лукашевича) форма записи выражений, при которой сначала записываются операнды, а затем знак операции. Например, выражение  $1 + 2$  записывается как  $1\ 2\ +$ , а выражение  $(1+2)\cdot 4$  — как  $1\ 2\ +\ 4\ *$ . В постфиксной форме порядок выполнения операций определяется исключительно их местоположением в строке; отпадает необходимость в использовании скобок и в таком понятии, как приоритет операций [48].

Главное преимущество постфиксной записи заключается в ее линейности. Обычная инфиксная форма записи нелинейна.

Для вычисления представленных в постфиксной форме выражений достаточно воспользоваться следующим простым алгоритмом: встречающиеся во входной строке операнды помещаются *на* стек, а встречающиеся операции производятся над двумя верхними значениями *со* стека, с помещением результата *на* стек. Таким образом, при корректной записи выражения к концу входной строки на стеке оказывается результат вычисления.

Входная строка	1	2	+	4	*
Стек	1	2	3	4	12
	1		3		

Также стек идеально подходит для передачи параметров функциям и процедурам. В языке Форт функции (как и стандартные операции  $+$ ,  $-$  и др.) берут аргументы со стека и помещают результат на стек. Например, функция возведения в квадрат (которую мы обозначим **SQR**), берет со стека один аргумент и кладет результат на стек. Пример

использования **SQR** в составе выражения:  $(10 - 3)^2 + 4^2$  запишется как 10 3 - SQR 4 SQR +. С синтаксической точки зрения **SQR** ведет себя как одноместная постфиксная операция. Форт, как и всякая стековая машина, исполняет программу — очередь  $q_P$  операндов и операций в постфиксном порядке.

$Top(q_P)$	10	3	-	<b>SQR</b>	4	<b>SQR</b>	+
	10	3	7	49	4	16	65
		10		49	49	49	

Рассмотренная функция может быть определена через умножение следующим образом:

: SQR DUP \* ;

где : **SQR** — заголовок объявления функции, **DUP** — встроенная функция дублирования вершины стека.

Таким образом при выполнении функции **SQR** стек неявно используется и для промежуточных вычислений:

$Top(q_P)$	10	3	-	<b>DUP</b>	*	4	<b>DUP</b>	*	+
	10	3	7	7	49	4	4	16	65
		10		7		49	4	49	
						49			

### 5.5.2.2 Реализация на массиве

Как и при реализации очереди на массиве ограничим максимальную глубину стека величиной **POOL\_SIZE**. Все элементы стека в таком случае размещаются в массиве *data* длины **POOL\_SIZE**. Переменная *size* играет двойную роль: это длина стека, а величина *size* - 1 является индексом его вершины.

```

const POOL_SIZE = 100;

type stack = record
  size : integer;
  data : array[0..POOL_SIZE - 1] of T
  ;
end;

procedure Create(var s : stack);
begin
  s.size := 0;
end;

function Empty(var s : stack) : boolean

```

```

const int POOL_SIZE = 100;

typedef struct
{
  int size;
  T data[POOL_SIZE];
} stack;

void Create(stack* s)
{
  s->size = 0;
}

bool Empty(stack* s)

```

```

;
begin
    Empty := s.size = 0;
end;

function Size(var s : stack) : integer;
begin
    Size := s.size;
end;

function Push(var s : stack; t : T) :
    boolean;
begin
    if(s.size >= POOL_SIZE) then
        Push := false
    else begin
        s.data[s.size] := t;
        s.size := s.size + 1;
        Push := true;
    end;
end;

function Pop(var s : stack) : boolean;
begin
    if(s.size = 0) then
        Pop := false
    else begin
        s.size := s.size - 1;
        Pop := true;
    end;
end;

function Top(var s : stack) : T;
begin
    if(s.size <> 0) then
        Top := s.data[s.size - 1];
end;

procedure Destroy(var s : stack);
begin
end;
}

int Size(stack* s)
{
    return s->size == 0;
}

bool Push(stack* s, T t)
{
    if(s->size >= POOL_SIZE)
        return false;
    s->data[s->size++] = t;
    return true;
}

bool Pop(stack* s)
{
    if(!s->size)
        return false;
    s->size--;
    return true;
}

T Top(stack* s)
{
    if(s->size)
        return s->data[s->size - 1];
}

void Destroy(stack* s)
{
}

```

### 5.5.2.3 Реализация на динамических структурах

Теперь опишем реализацию стека на динамических структурах. Для этого понадобится тип «элемент стека», который содержит ссылку на следующую компоненту.

```
type PItem = ^Item;
Item = record
  data : T;
  prev : PItem;
end;
```

```
struct Item
{
  T data;
  struct Item* prev;
};
```

Само определение стека заключается в описании двух переменных: указателя на вершину стека и целой переменной-глубины стека.

```
type stack = record
  top : PItem;
  size : integer;
end;
```

```
typedef struct
{
  struct Item* top;
  int size;
} stack;
```

```
procedure Create(var s : stack);
begin
  s.top := nil;
  s.size := 0;
end;
```

```
void Create(stack* s)
{
  s->top = 0;
  s->size = 0;
}
```

```
function Empty(var s : stack) : boolean
;
begin
  Empty := s.top = nil;
end;
```

```
bool Empty(stack* s)
{
  return s->top == 0;
}
```

```
function Size(var s : stack) : integer;
begin
  Size := s.size;
end;
```

```
int Size(stack* s)
{
  return s->size;
}
```

```
function Push(var s : stack; t : T) :
  boolean;
var i : PItem;
begin
```

```
bool Push(stack* s, T t)
{
  struct Item* i = malloc(sizeof(struct
    Item));
```

```

new(i);
if i = nil then
    Push := false
else begin
    i^.data := t;
    i^.prev := s.top;
    s.top := i;
    s.size := s.size + 1;
    Push := true;
end;
end;

```

```

function Pop(var s : stack) : boolean;
var i : PItem;
begin
    if (s.size = 0) then
        Pop := false
    else begin
        i := s.top;
        s.top := s.top.prev;
        s.size := s.size - 1;
        dispose(i);
        Pop := true;
    end;
end;

```

```

function Top(var s : stack) : T;
begin
    if (s.top <> 0) then
        Top := s.top.data;
    end;

```

```

procedure Destroy(var s : stack);
var i : PItem;
begin
    while(s.top <> nil) do begin
        i := s.top;
        s.top := s.top.prev;
        dispose(i);
    end;
    s.top := nil;
    s.size := 0;
end;

```

```

if (!i)
    return false;
i->data = t;
i->prev = s->top;
s->top = i;
s->size++;
return true;
}
|
```

```

bool Pop(stack* s)
{
    if (!s->size)
        return false;
    struct Item* i = s->top;
    s->top = s->top->prev;
    s->size--;
    free(i);
    return true;
}
|
```

```

T Top(stack* s)
{
    if (s->top)
        return s->top->data;
}
|
```

```

void Destroy(stack* s)
{
    while(s->top)
    {
        struct Item* i = s->top;
        s->top = s->top->prev;
        free(i);
    }
    s->top = 0;
    s->size = 0;
}
|
```

Мы не приводим здесь программы реверса стека, поскольку для этого всего лишь надо переместить все его элементы во вспомогательный стек, а затем скопировать эту вспомогательную стековую переменную в исходную. Инверсия элементов выполняется автоматически при занесении их во вспомогательный стек. Следовательно, стек может быть использован и для итеративного реверсирования очереди. Рекурсивный реверс стека представляет лишь умозрительный интерес.

В заключение еще раз подчеркнем сходство функциональных спецификаций стека и очереди, различия между которыми также проявляются в их разноголовочности: стек — одноголовочная структура, а очередь — двухголовочная. Однако это различие можно сгладить, моделируя двумя одноголовочными стеками одну двухголовочную очередь. Определим операцию постановки в очередь через добавление в стек, а извлечение из очереди будем осуществлять путем выемки из первого стека верхнего элемента и помещения его во второй стек. Когда в первом стеке останется только один элемент, извлечем его и «перельем» содержимое второго стека обратно в первый. При этом две инверсии погасят друг друга, а первый элемент очереди, лежавший на дне стека, в результирующую очередь не попадет. Кстати, реализация очереди двумя стеками небессмыслена, т. к. в отличие от очередей, стеки имеют аппаратную поддержку во всех современных процессорах.

Вспоминая дек (двуихконцовую очередь), мы можем предложить еще одну его реализацию, базирующуюся на двух стеках или очередях, контролирующих общую последовательность. Верно и обратное: ограничивая разными способами функциональность дека, мы получим очередь или стек.

Отечественная информатика обогатила мировую сокровищницу жемчужиной встречных стеков, когда два стека располагаются навстречу друг другу в одном массиве, что позволяет двум стекам поочередно пользоваться общим свободным пространством, сохраняя простоту и быстроту отображения реализации стека на массив. Пара стеков требуется, например, при переводе выражения из инфиксной в обратную польскую запись. К сожалению, идея не обобщается на случай трех и более стеков.

## 5.6 Линейный список

Линейные списки являются обобщением ранее изученных последовательных структур с ограниченным доступом: файлов, очередей и стеков [49, 20]. Они позволяют нам представить последовательность элементов так, чтобы, в отличие от очереди и стека, *каждый* элемент был бы доступен и для этого не нужно было бы извлекать из структуры предшествующие элементы. В терминологии Д. Кнута [63] линейные списки являются частным случаем нелинейных списков общего вида, в число которых входят деревья, графы и другие произвольные сложные структуры.

*Линейный список* — это представление в ЭВМ конечного упорядоченного динамического множества элементов типа  $T$ . Точнее, это не множество, а *мультимножество*, т. к. в последовательности могут находиться элементы с одинаковыми значениями. Элементы этого множества линейно упорядочены, но порядок определяется не номерами (индексами), как в массиве, а относительным расположением элементов [85]. Отношений порядка на этом множестве может быть не одно, а два, и тогда мы имеем дело с двусвязными (двусторонними или симметричными) списками. Для закольцевания списка необходимо доопределить в качестве следующего за последним первый элемент списка, а в качестве

предыдущего первому элементу — последний (голова указывает на хвост, а хвост на голову). Таким образом отношения порядка (предыдущий и/или следующий) становятся определенными для всех смежных (с точностью до закольцования) пар элементов списка. Список обозначается простым перечислением элементов в заданном порядке в круглых скобках через запятую. Например, список  $l$  из пяти элементов типа  $T$  выглядит так:

$$l = (t_1, t_2, t_3, t_4, t_5)$$

Линейные списки естественно использовать всякий раз, когда встречаются упорядоченные множества переменного размера, где включение, поиск и удаление элементов должны выполняться не систематически в голове или хвосте, как для файлов или стеков, а в произвольных последовательно достижимых местах, *но с сохранением* порядка следования остальных элементов. Таким порядком могли бы быть, например, приоритеты, присвоенные заданиям, ожидающим обработки в операционной системе или распечатки на сетевом принтере; принцип очереди: первым пришел — первым обслуживается, — негибок и недостаточен. Для постановки приоритетного задания в такую очередь, необходимо последовательно пройти список с самого начала и вставить это задание перед первым элементом, имеющим меньший приоритет.

Заметим, что порядок следования элементов в списке не предполагает упорядоченности хранимых в этих элементах значений.

Наиболее простой пример линейных списков в информатике — списки переменных в описаниях:

```
int x, y, z;

typedef struct
{
    double a;
    bool b;
    char c;
    short int d;
} S;
```

Для *поиска* элемента в списке надо просматривать его с начала, сравнивая искомое значение со значением, содержащимся в очередном хранимом элементе. В более сложном случае аргументом поиска (ключом) является одно из полей значения. Элементы списка не считаются пронумерованными, но можно прописать номера в дополнительных полях элементов (но при вставке и удалении номера необходимо снова вычислять ценой  $O(N)$ ) или генерировать их при прохождении списка. Цена поиска элемента в линейном списке есть  $O(N)$ : искомый элемент может с равной вероятностью находиться в любом месте списка от первого до  $N$ -ого. Если элемент находится в начале списка, то для поиска надо сделать один шаг, если на втором месте — два шага и т. д. Если элемент находится на последнем месте, то необходимо сделать  $N$  шагов. В *среднем* надо сделать

$$\frac{1 + 2 + \dots + N}{N} = \frac{N(N + 1)}{2N} = \frac{N + 1}{2}$$

шагов. То есть список, как очередь и стек, является последовательной структурой с линейным временем доступа — максимальным для линейных структур. Для некоторых

нелинейных структур (например, разреженных матриц, представляемых в виде списков списков элементов) время поиска элемента пропорционально  $N^2$ .

Для *добавления* нового элемента в список необходимо указать значение, *перед* (или после) которого надо сделать вставку. Для начала надо найти искомый элемент в среднем за  $N/2$  шагов (если такого элемента в списке нет, то он обычно вставляется в конец списка). Если же искомый элемент найден, надо выполнить необходимый вариант вставки. Вставка спереди легче выполняется в двусвязном списке; в односвязном для этого требуется запоминание ссылки на предпоследний элемент. Вставка сзади технически проще, здесь достаточно односвязного списка. Вставка спереди более логична, например, при включении с сохранением порядка в упорядоченные списки. В книге Н. Вирта [53] приводятся своеобразные технические приемы вставки и удаления элементов списка: вместо работы с элементами списка производятся манипуляции со значениями элементов (перестановка).

*Удаление* элемента из списка обычно предваряется его поиском, занимающим в среднем  $N/2$  шагов. Цена самого удаления обычно невелика и пропорциональна  $O(1)$ .

**Замечание.** Вставка и удаление элемента очереди и стека обходится всего лишь в  $O(1)$ , что существенно дешевле среднесписочной  $O(N)$ . Это плата за произвольность места вставки и удаления.

### 5.6.1 Функциональная спецификация

Полная функциональная спецификация двусвязного линейного списка  $L_T$  объектов типа  $T$  достаточно длинна [72]:

СОЗДАТЬ:	$\emptyset \rightarrow L_T$
ПУСТО:	$L_T \rightarrow \text{boolean}$
ДЛИНА:	$L_T \rightarrow \mathbb{N}$
ПЕРВЫЙ:	$L_T \rightarrow T$
ПОСЛЕДНИЙ:	$L_T \rightarrow T$
СЛЕДУЮЩИЙ:	$L_T \times T \rightarrow T$
ПРЕДЫДУЩИЙ:	$L_T \times T \rightarrow T$
ВСТАВКА:	$L_T \times T \times T \rightarrow L_T$
УДАЛЕНИЕ:	$L_T \times T \rightarrow L_T$
УНИЧТОЖИТЬ:	$L_T \rightarrow \emptyset$

В левой части спецификации операции ВСТАВКА находится декартово произведение трех множеств: всевозможных списков, в которые должна быть проведена вставка, всех элементов *перед* которыми необходимо ее произвести и различных вставляемых элементов.

Помимо тривиальных свойств типа:

1. ПУСТО(СОЗДАТЬ) = **true**
2. ПУСТО(ВСТАВКА( $l, t, t$ )) = **false**

можно отметить следующие:

1. ПРЕДЫДУЩИЙ(СЛЕДУЮЩИЙ( $t$ )) =  $t$

## 2. СЛЕДУЮЩИЙ(ПРЕДЫДУЩИЙ( $t$ )) = $t$

Эти записи следует читать справа налево: каждый элемент является предыдущим к следующему за ним или следующим за предшествующим ему.

### 3. $l := \text{СОЗДАТЬ}$

$l := \text{ВСТАВКА}(l, t_2, t_2)$

$l := \text{ВСТАВКА}(l, t_2, t_1)$

$\text{ПЕРВЫЙ}(l) = t_1$

$\text{ПОСЛЕДНИЙ}(l) = t_2$

Это свойство сохранения порядка двух элементов может быть по индукции обобщено на произвольное число элементов. При этом, в отличие от очереди и стека, картину могут испортить произвольные вставки и удаления элементов.

Для кольцевых списков верно также

$$1. \text{ПРЕДЫДУЩИЙ}(\text{ПЕРВЫЙ}(l)) = \text{ПОСЛЕДНИЙ}(l)$$

$$2. \text{СЛЕДУЮЩИЙ}(\text{ПОСЛЕДНИЙ}(l)) = \text{ПЕРВЫЙ}(l)$$

### 5.6.2 Логическое описание

В отличие от очередей и стеков списковые структуры, хотя и не входят в стандартные языки программирования, но хорошо реализованы в альтернативных достаточно распространенных языках Лисп [50] и Пролог [58]. Более того, в языке Лисп это основная структура.

Рассмотрим сначала как списки реализованы в Прологе. По определению, пустой список является списком, он обозначается символом  $[]$ . Функтор «.» (точка) образует новый список путем добавления нового элемента в начало исходного списка. Так список

$$(t_1, t_2, t_3, t_4, t_5)$$

может быть записан на Прологе как

$$\cdot(t_1, \cdot(t_2, \cdot(t_3, \cdot(t_4, \cdot(t_5, [])))))$$

Однако Пролог не настолько консервативный язык. В нем существует более простой и практичный способ задания списка явным перечислением:

$$[t_1, t_2, t_3, t_4, t_5]$$

Логическое описание списков на Прологе базируется на встроенном предикате отдельения головы списка  $\text{«|»}$  и реализованных на самом Прологе предикатах (программных единицах этого языка), более или менее соответствующих вышеприведенной функциональной спецификации: проверка членства элемента или подсписка в заданном списке, вычисление длины, конкатенация, генерация перестановок элементов списка. Остальное может быть легко дописано на Прологе.

Примеры:

```
[T1, T2, T3 | Tail] /* Отделение от списка трех первых элементов. В
результате переменные T1, T2 и T3 получают значения первых трех
элементов списка, а Tail становится остатком списка */
```

```
% Определение длины производится рекурсивно
length([], 0).
length([_ | Tail], N) :- length(Tail, N1), N is N1 + 1.
```

Ознакомление со списками в Лиспе мы предоставляем читателю [50].

Задача проектирования списков на обычные языки программирования уже давно интересовала программистов: уже в 1963 году Дж. Вейценбаумом в САСМ был опубликован фортрановский код для SLIP (Symmetric LList Processor), который явился еще одним конструктивным доказательством эквивалентности алгоритмических систем.

Обсудим возможные варианты реализации списков на Паскале и Си. Поскольку в Паскале и в Си встроенных списков нет, а есть только библиотечные (`std::list` в STL), их логическое описание тесно переплетено с их физическим представлением.

Хотя в простых файловых системах типа FAT файл представляется логическим списком физических блоков, мы не рассматриваем реализацию списков на файлах, поскольку это плохо согласуется с сугубой последовательностью файлов Паскаля.

### 5.6.3 Физическое представление

Пока список не меняется, его удобно отображать на сплошной участок памяти (вектор). Сплошное представление списков обладает преимуществами для представления фиксированных списков или хронологических, добавление в которых всегда идет в конец: обращение к элементу списка может быть осуществлено за постоянное время за счет вычисления его адреса. При удалении элементов из сплошного списка образуются «дырки» (неиспользуемые фрагменты). Не существует простого способа пометить удаленные элементы, кроме сдвига всех элементов, начиная с удаленного, за  $N/2$  шагов. Так же высока и цена вставки в сплошном представлении. Если мы хотим удешевить вставку и удаление элемента в список на сплошном представлении, то мы можем надстроить над вектором хранения списка управляющий индексный вектор, суррогат ссылочной структуры [85], с. 202–206, п. 11.3. Целочисленными значениями компонент вектора можно указывать на следующие элементы списка и регулировать выделение и освобождение элементов массива. Можно предложить различные стратегии дефрагментации и сборки мусора: ведение списка свободных элементов памяти, ленивые либо трудоголические чистки, учет фактического использования областей памяти.

Как известно, динамические структуры, ценой некоторого расхода машинных ресурсов, освобождают нас от вышеупомянутых недостатков. Точнее, они перекладывают соответствующую работу на плечи среды языка и ядра операционной системы. Но мы все-таки рассмотрим впоследствии и более совершенную реализацию списка на массиве.

#### 5.6.3.1 Итераторы

Для реализации сложных динамических структур данных цепного или сплошного характера принято использовать т. н. итераторы.

Перед реализацией списка по его функциональной спецификации обсудим вопрос о навигации по списку. Спецификация утверждает, что всегда непосредственно доступны первый и последний элементы списка. Кроме того, в ней отражена возможность получения предыдущего и следующего элемента относительно любого данного элемента.

Для удобства организации списка и обеспечения единообразия доступа к нему определим объекты, обладающие функциями перехода от данного элемента списка к соседним. Зададим для них отношения равенства и неравенства. Два таких объекта равны тогда и только тогда, когда они указывают на один и тот же элемент списка. Также предоставим возможность чтения и записи элемента списка посредством введенных объектов. Такие объекты принято называть *итераторами*, и, конечно же, для них надо определить соответствующий тип данных.

Итак, итератор предназначен для навигации по списку, чтения и перезаписи элементов списка. Но остается открытый вопрос об инициализации итератора. Существует несколько способов сделать это. Во-первых, итератор в результате обычного присваивания может получить значение другого, ранее созданного итератора. Во-вторых, это может быть сделано функциями работы со списком ПЕРВЫЙ и ПОСЛЕДНИЙ. Возвращаемым значением функции ПЕРВЫЙ будет не сам элемент, а итератор, который на него указывает. А вот функция ПОСЛЕДНИЙ в этом случае будет выдавать не последний элемент списка, а итератор, *указывающий на фиктивный элемент после конца списка — терминатор*.

Введенные таким образом итераторы дают более удобный доступ к элементам списка. Например, для обхода массивов (посещения всех элементов в определенном порядке по одному разу) в Си используется такая идиома:

```
int i;  
  
for(i = 0; i < len; i++)  
{  
    // Действие с массивом  
}
```

Эта идиома работает в любом случае правильно, даже тогда, когда в массиве вовсе нет элементов. Часто используется другой вариант этой идиомы ( $T$  — тип элемента массива!):

```
T arr[MAX];  
T* i;  
  
for(i = &arr[0]; i != &arr[MAX]; i++)  
{  
    // Действие с массивом  
}
```

Последний вариант хорош еще и тем, что вместо цикла по счетчику дискретного времени  $i$ , используемому для выборки элементов пространства, явно повторяются операции с элементами массива, образующими некоторый его фрагмент. Это позволяет обрабатывать части массива таким же способом, как и целые массивы. Универсальность идиомы позволит записать обход элементов списка в той же манере:

```
// Пусть  $l$  — некоторый, возможно пустой, список  
Iterator i, last = Last(&l); // Элемент после конца
```

```

for(i = First(&l); NotEqual(i, last); Next(&i))
{
    // Действие со списком
}

```

Богатство языка Си позволяет «сочинить песню племени девятыю и еще шестидесятю способами» (Р. Киплинг). В Паскале вышеупомянутые примеры теряют свою эстетичность, но сохраняется другое достоинство принятой схемы с концевым маркером: ее простота, универсальность и скорость работы.

Простота и универсальность схемы, использующей терминатор, состоит в том, что один и тот же код работает и тогда, когда в списке есть элементы, и тогда, когда он пуст. Отказ от нее приведет к потребности рассматривать два случая при всяком обходе списка, что ведет как к увеличению размера исходного кода, так и к ошибкам при его написании. Скорость работы проявляется в функциях модификации списка: отказ от рассмотрения двух случаев позволяет сократить код программы и одновременно сэкономить на одной инструкции проверки. Правда, такой подход усложняет код функций СОЗДАТЬ и УНИЧТОЖИТЬ, но поскольку эти функции вызываются гораздо реже функций ВСТАВИТЬ и УДАЛИТЬ, то описанная схема приведет к увеличению быстродействия программы.

Итак, приступим к программированию итераторов. Сначала опишем тип элементов двунаправленного списка, указываемых итераторами:

```

type PItem = Item^;
Item = record
    prev : PItem;
    next : PItem;
    data : T;
end;

```

```

struct Item
{
    struct Item* prev;
    struct Item* next;
    T data;
};

```

Итератор, как было отмечено выше, всего лишь указывает на некоторый элемент. Упакуем ссылку на элемент в запись, чтобы предотвратить использование таких (определенных по умолчанию) операций ссылочного типа, как проверка на равенство, разыменование и, особенно для С/С++, инкремент и декремент. Дело в том, что оператор инкремента (**operator++**) в Си определен для любого указателя, и он возвращает указатель с адресом, на **sizeof(T)** большим исходного.

```

type Iterator = record
    node : PItem;
end;

```

```

typedef struct
{
    Item* node;
} Iterator;

```

Такое решение уже применялось выше при обходе массива, но ясно, что просто так увеличивать адрес и таким образом получить следующий элемент списка нельзя, поскольку эти элементы совершенно необязательно хранятся в памяти последовательно. Более того, в случае долгой и интенсивной работы программы со списком адреса вновь создаваемых компонент могут быть *меньше* уже существующих и не могут быть получены путем обычной инкрементации.

Теперь, следуя функциональной спецификации, опишем функции сравнения двух

итераторов на равенство и неравенство. Два итератора считаются равными тогда и только тогда, когда указывают на один и тот же элемент:

```
function Equal(var lhs, rhs : Iterator) :  
    boolean;  
begin  
    Equal := lhs.node = rhs.node;  
end;
```

... и неравными во всех остальных случаях:

```
function NotEqual(var lhs, rhs : Iterator)  
    : boolean;  
begin  
    NotEqual := not Equal(lhs, rhs);  
end;
```

Введение *NotEqual()* может показаться избыточным, но оно не только проясняет условия ветвлений и циклов, особенно в Си, но и упрощает эти условия, удаляя из них один унарный оператор.

Функция *Next()* продвигает итератор к следующему элементу списка. Чтобы не загромождать эту функцию защитным кодом, проверок легитимности предпринимаемой операции не осуществляется. Тем не менее, правильная организация списка (например, кольцевание) позволит предотвратить обращение по неопределенному адресу. Целостность списка и навигация по нему не нарушается.

```
function Next(var i : Iterator) :  
    Iterator;  
begin  
    i.node := i.node^.next;  
    Next := i;  
end;
```

Функция *Prev()* является зеркальным отражением функции *Next()*:

```
function Prev(var i : Iterator) :  
    Iterator;  
begin  
    i.node := i.node^.prev;  
    Prev := i;  
end;
```

Функция *Fetch()* нужна для чтения элемента списка, на который указывает итератор. Хотя итератор может указывать на терминатор, никакой проверки этого факта функция не осуществляет. Защитные действия загромождают код, снижают производительность и не являются *абсолютно* надежными, и поэтому все проверки должен осуществлять

```
bool Equal(const Iterator* lhs, const  
          Iterator* rhs)  
{  
    return lhs->node == rhs->node;  
}
```

```
bool NotEqual(const Iterator* lhs, const  
              Iterator* rhs)  
{  
    return !Equal(lhs, rhs);  
}
```

```
Iterator Next(Iterator* i)  
{  
    i->node = i->node->next;  
    return i;  
}
```

```
Iterator Prev(Iterator* i)  
{  
    i->node = i->node->prev;  
    return i;  
}
```

пользователь. Впрочем, это нетрудно, т. к. терминатор используется лишь с одной целью — указывать на конец списка.

```
function Fetch(var i : Iterator) : T;  
begin  
    Fetch := i.node^.data;  
end;
```

```
T Fetch(const Iterator* i)  
{  
    return i->node->data;  
}
```

Противоположная для *Fetch()* функция *Store()* позволяет записать данные в элемент списка.

```
procedure Store(var i : Iterator; t : T);  
begin  
    i.node^.data := t;  
end;
```

```
void Store(const Iterator* i, const T t)  
{  
    i->node->data = t;  
}
```

Изменяя функциональность итераторов путем запрещения функций *Fetch()* или *Store()* можно получить списки *только для чтения* или *только для записи* подобно тому, как это сделано при помощи *свойств (properties)* в среде Delphi.

Аналогично описанным *прямым* итераторам, можно дополнительно определить *обратные* итераторы, которые позволяют пройти список с последнего элемента до первого, полностью сохраняя семантику и идиомы прямых итераторов.

### 5.6.3.2 Реализация на динамических структурах

Подобно тому, как это было сделано в реализации очереди, соединим начало и конец списка. Поскольку список двунаправленный, каждый его элемент содержит ровно по две ссылки: на предыдущий и на следующий элементы списка. Воспользуемся этим, создав фиктивный элемент списка — терминатор. Его указатель *next* будет ссылаться на первый элемент списка (если он есть, в противном случае на терминатор), а *prev* — на последний значащий элемент списка (опять-таки, если он есть).

Такой подход позволит сократить количество указателей, организующих список, до минимально возможного — всего 1. Посредством этой единственной ссылки и осуществляется любой доступ к списку. Сначала обращаемся к терминатору, через него, в свою очередь, — к первому и последнему элементам списка, а через них — строго последовательно и к остальным. Еще одно достоинство подхода заключается в том, что при ошибочном применении операции *Next()* к итератору, указывающему на терминатор, с последующей записью значения в несуществующий элемент, оно будет занесено в очередной элемент закольцованного списка, не испортив какую-либо другую область памяти, в частности, не нарушив саму списковую структуру.

Итак, описание структуры типа список содержит всего две переменные: указатель на фиктивный элемент — голову-хвост списка и его длину:

```
type List = record  
    head : PItem;  
    size : integer;  
end;
```

```
typedef struct  
{  
    Item* head;  
    int size;
```

```
}
```

Функция *Create()* выделяет память под терминатор, создает пустой кольцевой список путем присваивания компонентам *prev* и *next* значения указателя на терминатор и устанавливает равной 0 длину списка.

```
procedure Create(var l : List);
begin
  new(l.head);
  l.head^.prev := l.head;
  l.head^.next := l.head;
  l.size := 0;
end;
```

```
void Create(List* l)
{
  l->head = malloc(sizeof(struct Item));
  l->head->next = l->head->prev =
    l->head;
  l->size = 0;
}
```

Функция *First()* создает итератор из ссылочной компоненты *next* терминатора списка. Мы определили итератор как запись, и функция *First()* получилась нескалярной, что ограничивает сферу ее использования расширенными версиями языка Паскаль. Впрочем, существует стандартный рецепт скаляризации функции структурного типа, когда возвращается не объект, а ссылка на него.

```
function First(var l : List) : Iterator;
var res : Iterator;
begin
  res.node := l.head^.next;
  First := res;
end;
```

```
Iterator First(const List* l)
{
  Iterator res = { l->head->next };
  return res;
}
```

Функция *Last()* создает итератор из указателя на терминатор списка.

```
function Last(var l : List) : Iterator;
var res : Iterator;
begin
  res.node := l.head;
  Last := res;
end;
```

```
Iterator Last(const List* l)
{
  Iterator res = { l->head };
  return res;
}
```

По построению списка он пуст, если итераторы начала и конца совпадают.

```
function Empty(var l : List) : boolean;
var fst, lst : Iterator;
begin
  fst := First(l);
  lst := Last(l);
  Empty := Equal(fst, lst);
end;
```

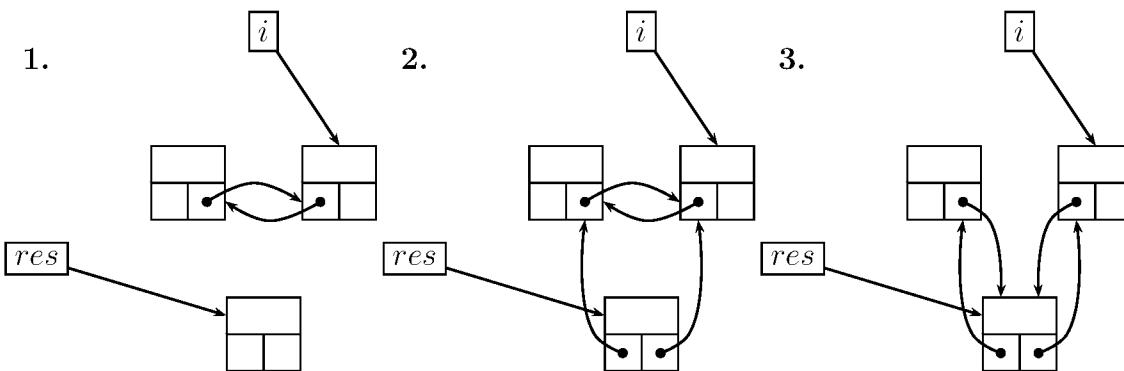
```
bool Empty(const List* l)
{
  Iterator fst = First(l);
  Iterator lst = Last(l);
  return Equal(&fst, &lst);
}
```

Функция *Size()* всего лишь возвращает хранимый размер списка. В качестве альтернативы функция могла бы вычислять длину путем перебора элементов списка с начала и до встречи с терминатором.

```
function Size(var l : List) : integer;
begin
    Size := l.size ;
end;
```

```
int Size(const List* l)
{
    return l->size;
}
```

Функция *Insert()* согласно спецификации вставляет элемент в список *перед* заданным. Сначала функция запрашивает память и в случае неуспеха этой операции возвращает итератор, указывающий на терминатор. В случае успеха результатом функции является итератор, указывающий на добавленный элемент. Приведем графическую иллюстрацию процесса вставки элемента [63]:



У этой функции есть потенциальный недостаток, связанный с отсутствием проверки принадлежности указанного итератором элемента списку, поэтому программист должен быть более внимательным, передавая аргументы этой функции. Впрочем, вставка не по своему итератору не приведет к катастрофе, но вот значение длины станет неактуальным. Вычисление длины перебором списка вместо получения ее текущего значения, ассоциированного со списком, устраниет проблему. Однако частое определение длины ( $O(N)$ ) может серьезно отразиться на быстродействии программы работы с таким списком.

```
function Insert(var l : List; var i :
    Iterator; t : T) : Iterator;
var res : Iterator;
begin
    new(res.node);
    if (res.node = nil) then
        Insert := Last(l) { Возвращ
        терминатор! }
    else begin
        res.node^.data := t;
        res.node^.next := i.node;
        res.node^.prev := i.node^.prev;
    end
end;
```

```
Iterator Insert(List* l, Iterator* i,
    const T t)
{
    Iterator res = { malloc(sizeof(struct
        Item)) };
    if (!res.node)
        return Last(l);
    res.node->data = t;
    res.node->next = i->node;
    res.node->prev = i->node->prev;
    res.node->prev->next = res.node;
    i->node->prev = res.node;
```

```

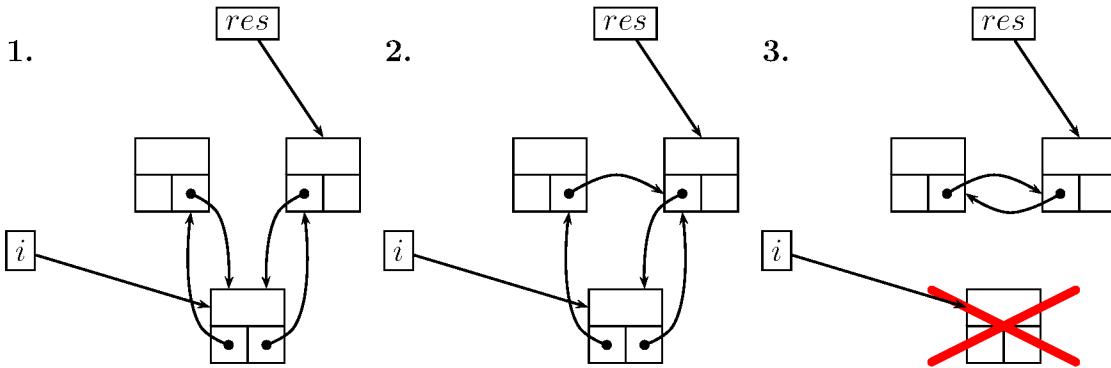
res.node^.prev^.next := res.node;
i.node^.prev = res.node;
l.size := l.size + 1;
Insert := res; { Возврат
    итератора на добавленный
    элемент }
end;
end;

```

```

l->size++;
return res;
}
|
```

Функция *Delete()* удаляет элемент, на который указывает итератор, возвращая в качестве результата итератор, ссылающийся на следующий за удаленным элементом. Если итератор указывает на терминатор, то удаления не происходит, и функция принимает значение терминатора списка *Last()*.



```

function Delete(var l : List , var i :
    Iterator ) : Iterator ;
var res : Iterator ;
begin
    res := Last(l);
    if(Equal(i, res)) then
        Delete := res
    else begin
        res.node := i.node^.next;
        res.node^.prev := i.node^.prev;
        i.prev^.next = res.node;
        l.size := l.size - 1;
        dispose(i.node);
        i.node := nil;
        Delete := res;
    end;
end;

```

```

Iterator Delete(List* l, Iterator* i)
{
    Iterator res = Last(l);
    if(Equal(i, &res))
        return res;
    res.node = i->node->next;
    res.node->prev = i->node->prev;
    i->prev->next = res.node;
    l->size--;
    free(i->node);
    i->node = 0;
    return res;
}
|
```

Функция *Destroy()* уничтожает список, перебирая его элементы и удаляя их. Заметим, что можно было бы вызывать функцию *Delete()*, передавая ей в качестве аргумента указатель на текущий первый элемент списка *First()* до тех пор, пока список не станет

пуст, после чего удалить терминатор. Но тогда код функции стал бы короче всего на одну-две строки, а накладные расходы на вызов функций увеличили бы время работы *Destroy()*.

```
procedure Destroy(var l : List);
var i, pi : PItem;
begin
  i := l.head^.next;
  while(i <> l.head) do begin
    pi := l.head;
    i := i^.next;
    dispose(pi);
  end;
  dispose(l.head);
  l.head := nil;
  l.size := 0;
end;
```

```
void Destroy(List* l)
{
  struct Item* i = l->head->next;
  while(i != l->head)
  {
    struct Item* pi = l->head;
    i = i->next;
    free(pi);
  }
  free(l->head);
  l->head = 0;
  l->size = 0;
}
```

Как и в случае очереди, возможна рекурсивная версия программы уничтожения, но никакого практического эффекта она не дает.

#### 5.6.3.3 Реализация на массиве

Приведенная выше реализация списка на динамических структурах обладает существенными преимуществами, например, потенциальной бесконечностью списка: его длина статически не определена и ограничена лишь размером доступной памяти всей вычислительной системы. Однако при очень частых операциях модификации списка запросы к распределителю памяти могут опять-таки нелучшим образом отразиться на быстродействии программы. В этом случае, а также если известно, что длина списка ни при каких условиях не превысит заданной величины (обозначим ее уже известным идентификатором *POOL\_SIZE*), прибегают к более эффективной реализации списка на массиве.

При переходе от динамического выделения памяти к статическому есть возможность повторного использования большей части уже написанного кода. В частности, предложенные итераторы полностью совместимы с динамической версией списка, поскольку нет никакой необходимости менять тип элемента списка (можно было бы отказаться от указателей и перейти к индексам, но поскольку и целочисленные индексы, и ссылки занимают в памяти машинные слова одинакового размера, никакого выигрыша в памяти это не даст, но потребует весьма существенной модификации кода). Однако, программировать все-таки придется.

Попробуем сохранить итераторы и отказаться от использования стандартного распределителя памяти, вызываемого посредством **new** и **dispose**.

Описание типа список в данной реализации будет несколько шире. К уже описанным полям *head* и *size* добавим *top* и *data*. В поле *top* мы будем хранить *ссылку* на первый свободный элемент массива *data*. В самом массиве *data* кроме места для хранения *POOL\_SIZE* элементов предусмотрим еще одну дополнительную компоненту для хранения терминатора. Введение полей *top* и *data* в структуру списка вызвано необходимостью явного представления ранее находившегося за кулисами резидентного массива для реали-

зации списка; *top* играет роль ссылочного выходного параметра процедуры порождения нового элемента списка **new**, указывая ссылочным (не индексным!) образом на первый свободный элемент массива.

```
const POOL_SIZE = 100;

type List = record
  head : PItem;
  size : integer;
  top : PItem;
  data : array[0..POOL_SIZE] of Item;
end;
```

```
const int POOL_SIZE = 100;

typedef struct
{
  struct Item* head;
  int size;
  struct Item* top;
  struct Item data[POOL_SIZE + 1];
} List;
```

Теперь рассмотрим функционирование списка. Сначала список пуст и весь массив *data* считается свободным, а элемент *data[POOL\_SIZE]* отводится под терминатор. Свободные элементы списка с индексами от 0 до *POOL\_SIZE* – 1 связываются друг с другом, причем полю *next* *i*-того элемента присваивается ссылка на *i* + 1-ый элемент массива. Последний свободный элемент массива не связан ни с каким другим, и в его поле *next* хранится значение **nil**. Компонента *top* любого, в данном случае пустого, списка указывает на первый из свободных элементов, а терминатор пустого списка так же, как и его динамический аналог, по-прежнему удостоверяет пустоту списка и устанавливает обе ссылки на себя. Функция *Create()* создает необходимую адресную разметку — суррогат вышеописанной динамической структуры, пользуясь адресной функцией Borland/GNU Pascal **@** (сионим **Addr**). Другим способом унификации кода с динамической версией было бы первоначальное получение ссылки на массив посредством **new** с последующим преобразованием ее в целое число с помощью вариантной записи и дальнейшими манипуляциями с индексными переменными, базирующимися на полученном таким образом адресе. Этот способ менее приемлем для компилируемых языков без динамических структур данных, таких, как Фортран и Бейсик.

```
procedure Create(var l : List);
var i : integer;
begin
  for i := 0 to POOL_SIZE – 1 do
    l.data[i].next := @(l.data[i + 1]);
  l.data[POOL_SIZE – 1].next := nil;
  l.head := @(l.data[POOL_SIZE]);
  l.head^.prev := @(l.head);
  l.head^.next := @(l.head);
  l.top := @(l.data[0]);
  l.size := 0;
end;
```

```
void Create(List* l)
{
  int i;
  for(i = 0; i < POOL_SIZE; i++)
    l->data[i].next = &(l->data[i + 1]);
  l->data[POOL_SIZE – 1].next = 0;
  l->head = &(l->data[POOL_SIZE]);
  l->head->prev = l->head->next
    = l->head;
  l->top = &(l->data[0]);
  l->size = 0;
}
```

Заметим, что в сформированном таким образом массиве свободные элементы обра-

зуют односвязный список. Тип *List* содержит ссылку на его начало, поэтому из списка легко брать элементы. Вместо вызова функции **new** необходимо лишь взять значение переменной *top* и передвинуть *top* на следующий элемент списка, удаляя тем самым этот элемент из списка свободных. Правда, не всегда можно передвинуть *top*, поскольку в нем может содержаться **nil**, разыменование которого приведет к исключительной ситуации. Функция *Insert()* проверяет все условия и не допускает такой ситуации, возвращая *Last()* в случае переполнения буфера.

Сравнение этой функции с ее вариантом для динамического списка показывает, что благодаря избранному подходу пришлось лишь переписать одну строку и добавить одну строку. Весь остальной код остался неизменным:

```

function Insert(var l : List; var i :
    Iterator; t : T) : Iterator;
var res : Iterator;
begin
    { Предоставление свободного
        элемента массива для очередной
        компоненты списка (a la new()) }
    res.node := l.top;
    if (res.node = nil) then
        Insert := Last(l)
    else begin
        l.top := l.top^.next;

        res.node^.data := t;
        res.node^.next := i.node;
        res.node^.prev := i.node^.prev;
        res.node^.prev^.next := res.node;
        i.node^.prev = res.node;
        l.size := l.size + 1;
        Insert := res; { Возврат
            итератора на добавленный
            элемент }

    end;
end;

```

```

Iterator Insert(List* l, Iterator* i,
    const T t)
{
    /* Предоставление свободного
        элемента массива для очередной
        компоненты списка (a la malloc())
        */
    Iterator res = { l->top };
    if (!res.node)
        return Last(l);
    l->top = l->top->next;

    res.node->data = t;
    res.node->next = i->node;
    res.node->prev = i->node->prev;
    res.node->prev->next = res.node;
    i->node->prev = res.node;
    l->size++;
    return res;
}

```

Функция *Delete()*, как и прежде, удаляет элемент из списка, но возвращает память не операционной системе, а списку свободных элементов, помещая этот элемент в *начало* списка. Можно было бы помещать элемент и в конец, но пришлось бы или заводить еще одну переменную-указатель на хвост массива, или вообще проходить его от начала до конца, тем самым нивелировав все преимущества отказа от стандартного распределителя памяти. Мы видим, что доступ к списку свободных элементов ограничен и фактически реализует дисциплину *стека*: все свободные элементы равнозначны и одинаково пригодны для хранения элементов списка. Стек свободных элементов позволяет вернуть память так же быстро, как и получить ее, и к тому же обойтись без лишних указателей (на конец списка). Таким образом в одном буферном массиве мы организовали эффективный

симвоз различных структур.

```
function Delete(var l : List, var i :  
    Iterator) : Iterator;  
var res : Iterator;  
begin  
    res := Last(l);  
    if(Equal(i, res)) then  
        Delete := res  
    else begin  
        res.node := i.node^.next;  
        res.node^.prev := i.node^.prev;  
        i.prev^.next := res.node;  
        l.size := l.size - 1;  
  
        { Освобождение занятого  
            элемента буфера }  
        i.node^.next := l.top;  
        l.top := i.node;  
  
        i.node := nil;  
        Delete := res;  
    end;  
end;
```

Функция *Destroy()* не возвращает память ОС, т. к. массив статический и заранее и навсегда выделен компилятором. Во избежание повторного ошибочного использования списка все его служебные указатели устанавливаются в **nil**.

```
procedure Destroy(var l : List);  
begin  
    l.head := nil;  
    l.size := 0;  
    l.top := nil;  
end;
```

```
Iterator Delete(List* l, Iterator* i)  
{  
    Iterator res = Last(l);  
    if(Equal(i, &res))  
        return res;  
    res.node = i->node->next;  
    res.node->prev = i->node->prev;  
    i->prev->next = res.node;  
    l->size--;  
  
    // Освобождение занятого элемента  
    // буфера  
    i->node->next = l->top;  
    l->top = i->node;  
  
    i->node = 0;  
    return res;  
}
```

```
void Destroy(List* l)  
{  
    l->head = 0;  
    l->size = 0;  
    l->top = 0;  
}
```

Итак, мы перешли от списков на динамических структурах к спискам на массиве, переписав всего четыре функции из 11! Исправления коснулись лишь кода, осуществляющего управление памятью. Этот факт наталкивает на мысль, что функцию *Create()* можно дополнительно параметризовать распределителем памяти. В частности, в STL C++ все контейнеры (векторы, деки, списки) в качестве аргумента по умолчанию принимают стандартный *распределитель памяти (allocator)*, который, при необходимости, можно всегда заменить. Так std::list по умолчанию является динамическим списком, но его легко превратить в список на массиве при помощи собственного распределителя, например, аналогичного только что описанному.

## 5.7 Дек

Дек - двухсторонняя очередь, динамическая структура данных, в которой элементы можно добавлять и удалять как в начало, так и в конец.

Операции над деком:

- включение элемента справа;
- включение элемента слева;
- исключение элемента справа;
- исключение элемента слева;
- определение размера;
- очистка.

Физическая структура дека в статической памяти идентична структуре кольцевой очереди. Динамическая реализация является очевидным объединением стека и очереди.

Приведем реализацию на языке Си. Для начала напишем заголовочный файл и описание структуры данных.

```
typedef struct Item{  
    int data; //хранимое значение  
    struct Item * next; //указатель на следующий элемент  
    struct Item * prev; //указатель на предыдущий элемент  
} Item;  
  
typedef struct deque{  
    Item *left ; //указатель на левый конец дека  
    Item *right; //указатель на правый конец дека  
    int size; //размер дека  
} deque;  
  
deque * CreateDeque(vector * v); //создание дека из вектора  
bool isEmptyDeque(deque *d); //проверка отсутствия элементов в деке  
bool PushLeftDeque(deque * d, int i); //добавить элемент с левого конца  
bool PushRightDeque(deque * d, int i); //добавить элемент с правого конца  
bool PopLeftDeque(deque * d); //взять элемент с левого конца  
bool PopRightDeque(deque * d); //взять элемент с правого конца  
int TopLeftDeque(deque * d); //посмотреть элемент с левого конца  
int TopRightDeque(deque * d); //посмотреть элемент с правого конца  
bool DeleteDeque(deque * d); //очистить дек
```

Далее опишем функции для работы с деком.

```

//создание дека из вектора
deque * CreateDeque(vector * v){
    deque * d = (deque*)malloc(sizeof(deque));
    d->size = 0;
    d->left = 0;
    d->right = 0;

    for (int i = 0; i < v->size; ++i)
        PushLeftDeque(d, v->array[i]);

    return d;
}

//проверка отсутствия элементов в деке
bool isEmptyDeque(deque * d){
    if (d->size == 0)
        return true;
    return false;
}

//добавить элемент с левого конца
bool PushLeftDeque(deque * d, int i){
    Item *q = 0;

    if (d->left){
        q = d->left;
        d->left = d->left->prev;
    }

    d->left = (Item*)malloc(sizeof(Item));
    d->left->data = i;
    d->left->next = q;

    if (d->left->next)
        d->left->next->prev=d->left;

    d->left->prev = 0;

    if (!q)
        d->right=d->left;

    d->size++;

    return true;
}

```

```

//добавить элемент с правого конца
bool PushRightDeque(deque * d, int i){
    Item *q = 0;

    if(d->right!=0){
        q = d->right;
        d->right = d->right->next;
    }

    d->right = (Item*)malloc(sizeof(Item));
    d->right->data = i;
    d->right->next = 0;
    d->right->prev = q;

    if(d->right->prev)
        d->right->prev->next=d->right;

    if (!q)
        d->left=d->right;

    d->size++;

    return false;
}

```

```

//взять элемент с левого конца
bool PopLeftDeque(deque * d){

```

```

    if(isEmptyDeque(d))
        return false;

    Item *q;
    q = d->left->next;
    free(d->left);
    d->size--;
    d->left = q;

    if(d->size == 0){
        d->right = d->left;
        return true;
    }

    d->left->prev = 0;

    return true;
}

```

```

//взять элемент с правого конца
bool PopRightDeque(deque * d){
    if(isEmptyDeque(d))
        return false;

    Item *q;
    q = d->right->prev;
    free(d->right);
    d->size--;
    d->right=q;

    if(d->size == 0){
        d->left = d->right;
        return true;
    }
    d->right->next = 0;

    return true;
}

//посмотреть элемент с левого конца
int TopLeftDeque(deque * d){
    if(isEmptyDeque(d));
        //ошибка, обращение к нулевому указателю

    return d->left->data;
}

//посмотреть элемент с правого конца
int TopRightDeque(deque * d){
    if(isEmptyDeque(d))
        //ошибка, обращение к нулевому указателю

    return d->right->data;
}

//удалить дек
bool DeleteDeque(deque * d){
    if(isEmptyDeque(d))
        return false;

    Item *q;
    q = d->left->next;
    free(d->left);
    d->size--;

```

```

if(q == 0){
    d->left = d->right=q;
    return true;
}

d->left = q;
d->left->prev = 0;

return DeleteDeque(d);
}

```

Задачи, требующие структуры дека, встречаются в вычислительной технике и программировании гораздо реже, чем задачи, реализуемые на структуре стека или очереди, но на мероприятии Russian Code Cup 2012 была задача для решения которой требовалось использовать дек. Рассмотрим ее подробнее.

#### **Постановка задачи:**

Как известно, птицы любят сидеть на проводах. В уездном городе К есть один длинный, особенно полюбившийся им провод. В один прекрасный момент на нем сидело несколько птиц. Все было бы хорошо, да вот только пробежавшая под проводом свинья подняла такую тучу пыли, что очень разозлила птиц.

Став злыми, птицы начали бежать по этому проводу в разные стороны — кто-то налево, кто-то направо. При этом все птицы стали бежать с одинаковой скоростью, равной одному метру в минуту. При встрече двух птиц, двигающихся навстречу друг другу, они немедленно разворачиваются и начинают бежать с той же скоростью в противоположном направлении.

Этот процесс продолжался бы бесконечно долго, но только провод оказался все-таки конечным, и как только какая-та из птиц добегает до конца провода, она тут же взлетает, а все остальные птицы, ошеломленные этим, разворачиваются и начинают бежать в противоположном направлении. Если же до края добегают одновременно две птицы, то происходит два разворота, или, что то же самое, ничего не происходит.

Вам необходимо написать программу, которая по длине провода, начальными позициям и направлениям бега птиц выяснит для каждой птицы, в какой момент времени она улетит с провода.

#### **Формат входных данных:**

В первой строке задано единственное целое число  $L(1 \leq L \leq 109)$  — длина провода в метрах. Во второй строке записано число  $n(0 \leq n \leq 100000)$  — количество птиц, бегущих направо. В третьей строке записано  $n$  различных целых чисел  $a_i(0 < a_i < L)$  — расстояния в метрах от левого конца провода до птиц, бегущих направо. В четвертой строке записано число  $m(0 \leq m \leq 100000)$  — количество птиц, бегущих налево. В пятой строке записано  $m$  различных целых чисел  $b_i(0 < b_i < L)$  — расстояния в метрах от левого конца провода до птиц, бегущих налево. Никакие две птицы не находятся исходно в одном и том же месте. Гарантируется, что на проводе сидит хотя бы одна птица.

#### **Формат выходных данных:**

В первой строке выведите  $n$  целых чисел  $t_i$  — через сколько минут улетит  $i$ -я по порядку

описания во вводе птица, бегущая направо. Во второй строке выведите  $m$  целых чисел  $u_i$  — через сколько минут улетит  $i$ -я по порядку описания во вводе птица, бегущая налево.

В итоге решение задачи сводится к ведению двух деков, в которой удаление первого или последнего элемента осуществляется за  $O(1)$ . При взлете птицы из дека удаляется крайний элемент, а сами деки меняются местами. Храня пары  $\langle$ дек; сколько надо добавить к числам в нем $\rangle$  можно за  $O(1)$  находить, через какое время и в какую сторону улетит очередная птица. Поскольку порядок птиц не меняется, дополнительно следует хранить сортированный массив координат оригинального расположения птиц, чтобы определить, какая из птиц покинула провод.

Приведем решение данной задачи на языке Си с использованием ранее написанного дека и вектора.

```
#include <stdio.h>
#include "vector.h"
#include "deque.h"

#define INF 2*10000000000L

int main()
{
    int length, n, m;
    int addLeft = 0, addRight = 0;

    scanf("%d", &length); //длина провода

    scanf("%d", &n);

    vector right_input = CreateVector(n); //создаем вектор right_input
    for(i = 0; i<n; ++i)
        scanf("%d", &right_input.array[i]); //читываем данные в вектор

    scanf("%d", &m);

    vector left_input = CreateVector(m); //создаем вектор right_input
    for(i = 0; i<m; ++i)
        scanf("%d", &left_input.array[i]);

    SortVector(right_input); //сортируем вектор
    SortVector(left_input);

    deque * right = CreateDeque(right_input); //создаем дек из вектора
    deque * left = CreateDeque(left_input);

    ans = 0;
```

```

while (!isEmptyDeque(right) || !isEmptyDeque(left)) //пока left и right не пусты
{
    int L;
    if (!isEmptyDeque(left))
        L = TopLeftDeque(left) + addLeft;
    else
        L = INF;

    int R;
    if (!isEmptyDeque(right))
        R = length - (TopRightDeque(right) + addRight);
    else
        R = INF;

    m = min(L, R); //минимум из L и R
    ans += m;

    addLeft -= m;
    addRight += m;

    if (L < R)
    {
        PopLeftDeque(left); //достаем первый элемент слева из дека left
        swap(&addLeft, &addRight); //обмен значений addLeft и addRight
        SwapDeque(left, right); //обмен значений деков left и right
    }
    else if (L == R)
    {
        PopLeftDeque(left);
        PopRightDeque(right); //достаем первый элемент справа из дека right
    }
    else
    {
        PopRightDeque(right);
        swap(&addLeft, &addRight);
        SwapDeque(left, right);
    }
}

printf ("%d", ans);

return 0;
}

```

Наиболее красивым решением является решение на Python3 от авторов задачи:

```

from collections import deque

```

```

length = int(input())
n = int(input())
right = deque(sorted(map(int, input().split())))
m = int(input())
left = deque(sorted(map(int, input().split())))
addLeft, addRight = 0, 0

INF = 2*10**9
ans = 0
while right or left:
    L = left[0] + addLeft if left else INF
    R = length - (right[-1] + addRight) if right else INF
    m = min(L, R)
    ans += m
    addLeft -= m
    addRight += m
    if L < R:
        left.popleft()
        left, addLeft, right, addRight = right, addRight, left, addLeft
    elif L == R:
        left.popleft()
        right.pop()
    else:
        right.pop()
        left, addLeft, right, addRight = right, addRight, left, addLeft
print(ans)

```

## 5.8 Списки общего вида

В монографии [63] рассматриваются нелинейные непоследовательные списки, ортосписки, суперсписки и другие обобщения списковых структур. Интересной разновидностью сложных структур данных являются разреженные матрицы [51].

## 5.9 Понятие о рекурсии

Рекурсивным называется объект, частично состоящий или определяемый с помощью самого себя [54]. Рекурсия встречается не только в математике, но и в реальной жизни. Если встать между двумя зеркалами, то можно увидеть бесконечную последовательность уменьшающихся отражений исходной картинки [47]. Другой способ получить рекурсивное изображение — направить видеокамеру на телемонитор, на который приходит картинка с этой камеры. Лучше всех сказал о рекурсии немецкий поэт-сатирик Иоахим Рингельнац [1]:

... daß dieser Wurm an Würmern litt,

*die wiederum an Würmern litten.*<sup>1</sup>

Рекурсивные (рекуррентные) определения представляют собой мощный математический аппарат [20]. Например, натуральные числа определяются следующим соотношением:

- 0 есть натуральное число (по Бурбаки);
- число, следующее по порядку за натуральным, есть натуральное число.

Другой известный пример — функция факториал  $n!$  ( $n \geq 0$ ):

- $0! = 1$ ;
- $n > 0: n! = n(n - 1)!$ .

Мощь рекурсивного определения заключается в возможности *конструктивно* определить (породить) бесконечное множество объектов с помощью небольшого количества правил.

Программы, как и функции, тоже могут быть описаны рекурсивно. Явных повторений (итераций) рекурсивная программа не содержит. В общем виде рекурсивная программа  $P$  есть некоторая композиция  $\mathcal{P}$  из множества операторов  $S$ , не содержащих  $P$  и самой  $P$ :

$$P \equiv \mathcal{P}[S, P].$$

Не существует иного способа программной рекурсии, кроме как вызов подпрограммой самой себя по имени. Такая рекурсия называется *прямой*. Если программа  $P$  вызывает себя посредством другой  $Q$ , то рекурсия *косвенная*. Многоуровневая косвенная рекурсия тоже возможна, но часто неочевидна по тексту программы.

Существует более сложная классификация рекурсий: линейная, повторная (концевая, хвостовая), каскадная, удаленная, взаимная [20].

Выполнение рекурсивных вызовов процедур производится аналогично вызову одной процедуры из другой. Просто здесь все вызываемые процедуры одинаковы. Тем не менее при каждой рекурсивной активации блока процедуры порождается (и распределяется в стеке данных) новый экземпляр множества локальных переменных, включая формальные параметры. Предыдущие одноименные локальные переменные (и формальные параметры) откладывают в стек и тем самым экранируются.

То есть рекурсия — это углубляющееся многократное самоприменение, своеобразное повторение, аналогичное циклу. Поскольку мы прибегаем к рекурсии, чтобы постепенно автоматически свести задачу к более простой и даже тривиальной, этот процесс, как и цикл, должен заканчиваться. Для этого внутри рекурсивно-итерируемого тела  $P$  должно находиться некое условие завершения рекурсии  $B$ , которое под влиянием операторов тела рано или поздно становится истинным. Поэтому можно представить схему рекурсивных алгоритмов в одной из следующих форм:

$$P \equiv \mathbf{if} \ B \ \mathbf{then} \ \mathcal{P}[S, P] \ \mathbf{end}$$

---

<sup>1</sup> «... а этот глист страдал глистами,  
что мучались глистами сами» (нем.). Перевод Л. Макаровой

$$P \equiv \mathcal{P}[S, \text{if } B \text{ then } P \text{ end }]$$

Завершимость рекурсивного или итерационного процесса обычно имеет место, если он реализует функцию, монотонно убывающую с каждым повторением цикла на конечную величину, так что при достижении функцией некоторого барьерного (порогового) значения условие окончания рекурсии или цикла становится истинным ( $\neg B$ ). Конечность декремента и начального значения функции гарантирует, что повторение закончится за конечное число шагов. При рекурсивном вызове процедур удобно ввести формальный параметр перечислимого типа, по которому и будет отслеживаться и завершаться рекурсия. Уменьшая этот параметр арифметически  $n := n - 1$  или перечислительно  $w := \text{pred}(w)$ , мы постепенно редуцируем задачу, снижая ее размерность. Если в качестве условия завершения  $B$  использовать  $n > 0$ , то окончание по достижении 0 гарантировано. Наши схемы рекурсии могут быть расписаны следующим образом:

$$\left\{ \begin{array}{l} \text{Концевая или хвостовая рекурсия} \\ P(n) \equiv \text{if } n > 0 \text{ then } \mathcal{P}[S, P(n - 1)] \text{ end} \end{array} \right.$$

$$P(n) \equiv \mathcal{P}[S, \text{if } n > 0 \text{ then } P(n - 1) \text{ end}]$$

Как нас учит практическая теория алгоритмов и вычислимости, надо добиваться не только конечной, но и небольшой повторяемости рекурсивных вычислений. В противном случае накладные расходы на обслуживание рекурсивных вызовов превысят выигрыши от простого и красивого программирования.

Рассмотрим достоинства и недостатки рекурсии на примере вычисления факториала. Формула  $f_{i+1} = (i + 1) \cdot f_i$  содержит инкремент и индуктивное умножение. Схема вычисления факториала в наших обозначениях такова:

$$P \equiv \text{if } B \text{ then } S; P \text{ end}$$

где  $S \equiv \text{begin } i := i + 1; f := i * f \text{ end}$ , а  $B \equiv i < n$ .

Сводя вычисление факториала к вычислению факториала предыдущего числа, мы можем соответствующий декремент аргумента вписать прямо в фактические параметры рекурсивной процедуры-функции, возвращаемое значение которой заменит нам локальную переменную  $f$  итеративного варианта.

```
function f(i : integer) : integer;
{ i – формальный параметр,
  передаваемый по значению,
  локальный объект процедуры,
  размещаемый во временному стеке }
begin
  if i > 0 then
    f := i * f(i - 1)
  else
    f := 1
end;
```

```
int f(int i) /* i – формальный
параметр, передаваемый по
значению, локальный объект
процедуры, размещаемый во
временному стеке */
{
  return (i > 0) ? i * f(i - 1) : 1;
}
```

Для того, чтобы проиллюстрировать выполнение рекурсивной программы вычисления факториала, снабдим ее код сплошной трассировкой выполнения. Ниже приведен текст и протокол работы подобной программы для 64-битного целого типа на ЭВМ DEC Alpha:

```

program fact20(input, output);

type integer = integer64;

var n : integer;

function f(i : integer) : integer;
var res    : integer;
    offset  : integer;
    j      : integer;
begin
    offset := 2 * (n - i);
    writeln('' : offset, 'function f(, i : 1, ) : integer;');
    writeln('' : offset, 'begin');
    if i > 0 then begin
        writeln('' : offset, 'if i : 1, > 0 then');
        writeln('' : offset, 'f := , i : 1, *f(, i - 1 : 1,)');
        res := i * f(i - 1);
        write('' : offset, 'f := ');
        for j := i downto 1 do
            write(j : 1, '*');
        writeln('1', res : 1);
    end
    else begin
        writeln('' : offset, 'else');
        writeln('' : offset, 'f := 1');
        res := 1;
    end;
    f := res;
    writeln('' : offset, 'end;');
end; { f }

begin
    readln(n);
    f(n);
end. { fact20 }

function f(20) : integer;
begin
    if 20 > 0 then
        f := 20 * f(19)
    function f(19) : integer;
    begin
        if 19 > 0 then
            f := 19 * f(18)
        function f(18) : integer;

```

```

begin
  if 18 > 0 then
    f := 18 * f(17)
  function f(17) : integer;
begin
  if 17 > 0 then
    f := 17 * f(16)
  function f(16) : integer;
begin
  if 16 > 0 then
    f := 16 * f(15)
  function f(15) : integer;
begin
  if 15 > 0 then
    f := 15 * f(14)
  function f(14) : integer;
begin
  if 14 > 0 then
    f := 14 * f(13)
  function f(13) : integer;
begin
  if 13 > 0 then
    f := 13 * f(12)
  function f(12) : integer;
begin
  if 12 > 0 then
    f := 12 * f(11)
  function f(11) : integer;
begin
  if 11 > 0 then
    f := 11 * f(10)
  function f(10) : integer;
begin
  if 10 > 0 then
    f := 10 * f(9)
  function f(9) : integer;
begin
  if 9 > 0 then
    f := 9 * f(8)
  function f(8) : integer;
begin
  if 8 > 0 then
    f := 8 * f(7)
  function f(7) : integer;
begin
  if 7 > 0 then

```

```

f := 7 * f(6)
function f(6) : integer;
begin
  if 6 > 0 then
    f := 6 * f(5)
  function f(5) : integer;
  begin
    if 5 > 0 then
      f := 5 * f(4)
    function f(4) : integer;
    begin
      if 4 > 0 then
        f := 4 * f(3)
      function f(3) : integer;
      begin
        if 3 > 0 then
          f := 3 * f(2)
        function f(2) : integer;
        begin
          if 2 > 0 then
            f := 2 * f(1)
          function f(1) : integer;
          begin
            if 1 > 0 then
              f := 1 * f(0)
            function f(0) : integer;
            begin
              if 0 > 0 then
                f := 0 * f(-1)
              else
                f := 1
            end;
            f := 1*1 = 1
          end;
          f := 2*1*1 = 2
        end;
        f := 3*2*1*1 = 6
      end;
      f := 4*3*2*1*1 = 24
    end;
    f := 5*4*3*2*1*1 = 120
  end;
  f := 6*5*4*3*2*1*1 = 720
end;
f := 7*6*5*4*3*2*1*1 = 5040
end;
f := 8*7*6*5*4*3*2*1*1 = 40320
end;

```

```

        f := 9*8*7*6*5*4*3*2*1*1 = 362880
    end;
    f := 10*9*8*7*6*5*4*3*2*1*1 = 3628800
end;
f := 11*10*9*8*7*6*5*4*3*2*1*1 = 39916800
end;
f := 12*11*10*9*8*7*6*5*4*3*2*1*1 = 479001600
end;
f := 13*12*11*10*9*8*7*6*5*4*3*2*1*1 = 6227020800
end;
f := 14*13*12*11*10*9*8*7*6*5*4*3*2*1*1 = 87178291200
end;
f := 15*14*13*12*11*10*9*8*7*6*5*4*3*2*1*1 = 1307674368000
end;
f := 16*15*14*13*12*11*10*9*8*7*6*5*4*3*2*1*1 = 20922789888000
end;
f := 17*16*15*14*13*12*11*10*9*8*7*6*5*4*3*2*1*1 = 355687428096000
end;
f := 18*17*16*15*14*13*12*11*10*9*8*7*6*5*4*3*2*1*1 = 6402373705728000
end;
f := 19*18*17*16*15*14*13*12*11*10*9*8*7*6*5*4*3*2*1*1 = 121645100408832000
end;
f := 20*19*18*17*16*15*14*13*12*11*10*9*8*7*6*5*4*3*2*1*1 = 2432902008176640000
end;

```

Эта трасса наглядно подтверждает ресурсоемкость рекурсивного способа вычисления факториала.

Считая рекурсию своеобразным циклом, сведем ее к явной итерации:

```

i := 0;
f := 1;
while(i < n) do begin
    i := i + 1;
    f := f * i;
end;

```

```

int i = 0;
int f = 1;
while(i < n)
    f *= ++i;

```

Этот пример иллюстрирует метод сведения концевой рекурсии к итерации. Итеративная версия программы  $P$  имеет вид

$P \equiv [x := x_0; \mathbf{while} B \mathbf{do begin} S \mathbf{end}]$

```

function f(n : integer) : integer;
var s : Stack;
    res : integer;
begin
    res := 1;
    while(n <> 0) do begin

```

```

int f(int n)
{
    int res = 1;
    std::stack<int> s;
    while(n--)
        s.push(n);

```

```

Push(s, n);
n := n - 1;
end;
while(not Empty(s)) do begin
    res := res * Top(s);
    Pop(s);
end;
f := res;
end;

```

```

while(!s.empty())
{
    res *= s.top();
    s.pop();
}
return res;
}
|
```

Более сложные рекурсивные схемы также могут быть сведены к итерации. Например, числа Фибоначчи определяются рекуррентным соотношением:

$$fib_{n+1} = fib_n + fib_{n-1} \text{ для } n > 0$$

и  $fib_1 = 1$ ,  $fib_0 = 0$ . Трансляция этого соотношения на Паскаль приводит к рекурсивной функции:

```

function fib(n : integer) : integer;
begin
    if n = 0 then
        fib := 0
    else if n = 1 then
        fib := 1
    else
        fib := fib(n - 1) + fib(n - 2)
end;

```

Вычисление  $n$ -ого числа Фибоначчи приводит в двум рекурсивным активациям функции для вычисления  $fib(n - 1)$  и  $fib(n - 2)$ . Общее число вызовов функции будет экспоненциальным:

$$1 + 2 + 4 + 8 + \dots = 2^0 + 2^1 + 2^2 + 2^3 + \dots \leq 2^N \leq e^N$$

Таким образом, эта функция с практической точки зрения не вычислима.

Однако числа Фибоначчи можно вычислять по итеративной схеме, запоминая во вспомогательных переменных два предыдущих числа:

```

function fib(n : integer) : integer;
var i, f, f1, f2 : integer;
begin
    f := 1;
    f1 := 1;
    f2 := 0;
    i := 1
    while i < n do begin
        f := f1 + f2;
        f2 := f1;
        f1 := f;
        i := i + 1
    end;
    fib := f;
end;

```

```

    i := i + 1;
end;
fib := f;
end;

```

Полезная часть этих программ одинакова, но накладные расходы рекурсивной версии выше. Однако рекурсивная версия более ясна и подобна математическому определению. Не следует избегать рекурсивных программ, поскольку компьютерные ресурсы в настоящее время дешевы и доступны.

Приведем пример рекурсивной функции, не сводящейся к итерации: функция Аккермана

$$A(m, n) = \begin{cases} n + 1, & \text{если } m = 0, \\ A(m - 1, 1), & \text{если } n = 0, \\ A(m - 1, A(m, n - 1)), & \text{иначе} \end{cases}$$

Эта функция относится к типу удаленно-рекурсивных, здесь рекурсивный вызов встречается в фактических параметрах рекурсивной функции. Из описания функции Аккермана следует завершность ее вычисления, т. к. декрементации подвергаются все ее аргументы. Однако существует возможность нерекурсивного вычисления функции Аккермана. Как всегда в качестве суррогата рекурсии выступит стек.

```

function Ackermann(m, n : integer) :
  integer;
var s : Stack;
begin
  Create(s);
  Push(s, m);
  Push(s, n);
  while(Size(s) > 1) do begin
    n := Top(s);
    Pop(s);
    m := Top(s);
    Pop(s);
    if(m = 0) then
      Push(s, n + 1)
    else if(n = 0) then begin
      Push(s, m - 1);
      Push(s, 1)
    end
    else begin
      Push(s, m - 1);
      Push(s, m);
      Push(s, n - 1)
    end
  end;
  Ackermann := Top(s);
  Destroy(s);
end;

```

```

int Ackermann(int m, int n)
{
  std :: stack<int> s;
  s.push(m);
  s.push(n);
  while(s.size() > 1)
  {
    n = s.top();
    s.pop();
    m = s.top();
    s.pop();
    if (!m)
      s.push(n + 1);
    else if (!n)
    {
      s.push(m - 1);
      s.push(1);
    }
    else
    {
      s.push(m - 1);
      s.push(m);
      s.push(n - 1);
    }
  }
  return s.top();
}

```

## 5.10 Деревья

Среди структур данных наиболее важными являются *деревья* [72]; в частности, они наилучшим образом приспособлены для решения задач искусственного интеллекта и синтаксического анализа. В задачах *искусственного интеллекта* редко удается точно предусмотреть ход вычислительного процесса или обработки данных, идет ли речь о программе, играющей в шашки или шахматы, или определяющей план действий робота, доказывающей теоремы или правильность программ или анализирующей зрительные и звуковые образы. Алгоритмы этого типа эвристичны и содержат пробы с ошибками, после которых возможен возврат назад. Кроме того, на каждом этапе возможно несколько действий, каждое из которых приводит к новому состоянию, откуда могут выходить еще несколько путей. Поиск оптимальной стратегии требует, таким образом, ветвящегося процесса и построения дерева.

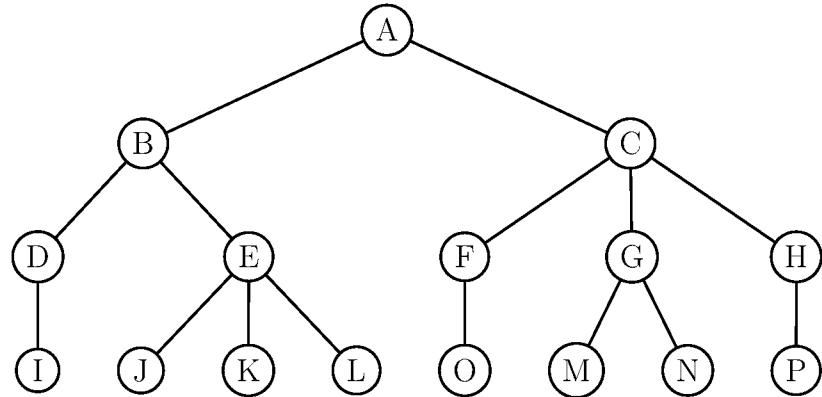
В *синтаксическом анализе* определяют *формальные грамматики*, которые представляют собой множество *правил подстановки* (как в алгоритмах Маркова). Так, выражение можно определить как множество *термов*, разделяемых знаками «+» или «-»; в свою очередь, терм есть совокупность *множителей*, разделяемых знаками «\*» или «/»; наконец, множитель может быть либо *идентификатором*, либо *константой*. Иерархическая структура выражения может быть описана древовидной схемой. При обработке текста программы транслятор может использовать деревья для декодирования таких выражений, а также для распознавания, например, иерархических структур данных.

Пусть  $T$  — некоторый тип данных. Деревом типа  $T$  называется структура, которая образована элементом типа  $T$ , называемым корнем дерева, и конечным, возможно пустым, множеством с переменным числом элементов — деревьев типа  $T$ , называемых поддеревьями этого дерева.

Это определение конструктивно определяет построение дерева. Дерево из одного элемента типа  $T$  (без поддеревьев) образует простейшее дерево с одним уровнем, в котором кроме корня ничего больше нет. Дерево с  $k$  уровнями получается из корня и поддеревьев, хотя бы одно из которых содержит ровно  $k - 1$  уровень [44], причём каждое из поддеревьев построено по таким же правилам.

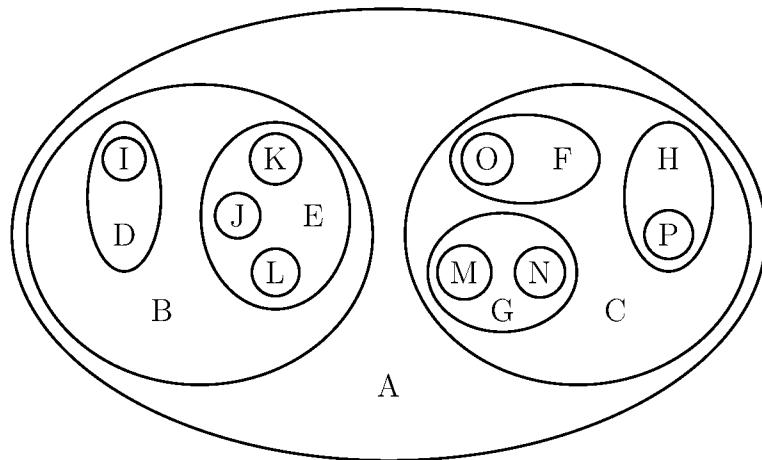
Это рекурсивное определение похоже на определения файла, очереди, стека и списка, но отличается от них нелинейностью ввиду наличия разветвлений. И наоборот: вырожденные формально древовидные структуры только с нулевым разветвлением являются линейными и последовательными или итеративными.

Деревья в информатике принято рисовать вверх корнем. Объясняется это традицией (генеалогическое и династическое деревья) и тем, что человеку свойственно рисовать и писать сверху вниз и, чаще всего, слева направо. Помимо общепринятого структурно-топологического изображения в виде графа:



существует несколько способов визуализации деревьев [63, 54]:

- вложенные диаграммы включения Эйлера-Венна



- иерархические скобочные структуры:  
 $(A (B (D (I), E (J, K, L)), C (F (O), G (M, N), H (P))))$
- многоуровневая ступенчатая запись

```

A
B
  D
    I
  E
    J
    K
    L
C
  F
  O
  G
  M
  N
H
  P
  
```

Кстати, вышеупомянутое топологическое дерево сгенерировано системой L<sup>A</sup>T<sub>E</sub>X при помощи макропакета **PSTricks** по следующему рекурсивно-вложенному (!) линейному текстовому описанию:

```
\pstree{\Tcircle{A}}{
    \pstree{\Tcircle{B}}{
        \pstree{\Tcircle{D}}{
            \Tcircle{I}
        }
        \pstree{\Tcircle{E}}{
            \Tcircle{J}
            \Tcircle{K}
            \Tcircle{L}
        }
    }
    \pstree{\Tcircle{C}}{
        \pstree{\Tcircle{F}}{
            \Tcircle{O}
        }
        \pstree{\Tcircle{G}}{
            \Tcircle{M}
            \Tcircle{N}
        }
        \pstree{\Tcircle{H}}{
            \Tcircle{P}
        }
    }
}
```

Приведенный пример не очень хорош с точки зрения быстрой визуальной проверки корректности расстановки фигурных скобок, но отступы прекрасно демонстрируют структуру дерева.

В дискретной математике распространены также теоретико-множественные способы представления деревьев (матрицы смежности и инцидентности), которые тоже могут использоваться для представления и обработки деревьев в памяти ЭВМ. Наконец, иногда прибегают к натурному моделированию — каркасно-проводочным моделям. Приведем определения, относящиеся к деревьям [63]:

1. Элементы типа  $T$ , входящие в дерево, называются *узлами* или *вершинами*.
2. Число поддеревьев данного узла называется *степенью* этого узла.
3. Узел, не имеющий поддеревьев, называется концевым узлом или *листом*.
4. Неконцевые узлы называются внутренними или узлами разветвления.
5. *Уровень* узла в дереве рекурсивно определяется так: корень дерева имеет уровень 1, а корень каждого поддерева данного узла имеет уровень на единицу больше уровня этого узла. Иногда уровень корня принимают равным нулю [54].
6. *Глубиной* дерева называется наибольшее значение уровня вершины. По определению уровня, внутренние вершины дерева не могут находиться на максимальном уровне. Таким образом, для определения глубины необходимо вычислить максимальный уровень для всех терминальных вершин дерева.

В приведенном примере дерева корнем является А, узлы I, J, K, L, M, N, O и Р — листья, вершины B, C, D, E, F, G и H — узлы разветвления. У узла А степень равна 2,

а у узла Е — 3. Дерево имеет 4 уровня и все его листья находятся на 4-ом уровне (в общем случае уровни листьев могут быть разными).

Для описания взаимного расположения узлов принятая терминология генеалогических деревьев, соответствующая родственным отношениям между лицами преимущественно по мужской линии (*отец*, *сын* или *предок* и *потомок*, но *дочерняя* вершина). Корень дерева является отцом для всех узлов 2-го уровня, которые составляют множество его сыновей. Аналогично определяются эти отношения и для других узлов дерева. Сыновья одного узла часто называются *братьями*. В примере дерева узел В является отцом узлов D и E и сыном для узла A. Узлы D и E — родные братья, узлы F, G и H — тоже родные братья, но не братья для D и E.

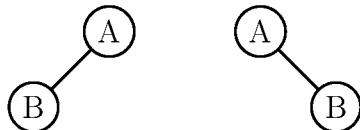
Если порядок поддеревьев существенен (а в большинстве приложений дело обстоит именно так), то сыновья упорядочиваются по номерам слева направо. Говоря о порядке сыновей, употребляют также термины *старший* и *младший* (чем меньше номер у сына, тем он старше). Старшинство сыновей будет удобно выражать явно в виде очереди.

Определенные нами таким образом деревья называются *деревьями общего вида* или сильно ветвящимися. Эти деревья представляют собой *непустые* структуры (есть по крайней мере один элемент — корень) с *многозвездным* разветвлением каждой вершины на непересекающиеся поддеревья. Иногда с целью унификации функциональных спецификаций допускаются пустые деревья общего вида.

### 5.10.1 Двоичные деревья

*Бинарное (двоичное) дерево* — это конечное множество узлов, которое или *пусто* или состоит из корня и двух непересекающихся поддеревьев, называемых левым и правым поддеревьями данного узла.

Бинарное дерево не является частным случаем дерева общего вида, это особый вид структуры данных, хотя и близкий к дереву. Различие этих понятий видно на рисунке:

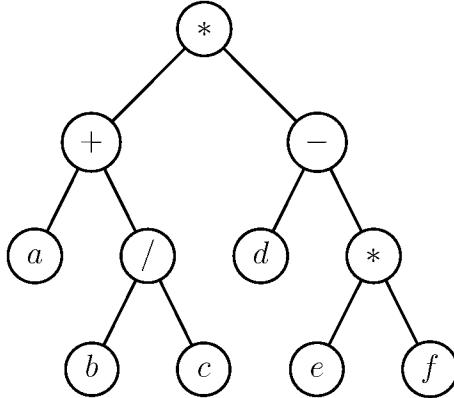


Два нарисованных дерева идентичны как деревья, но различаются как бинарные деревья, так как у первого из них есть левое поддерево, но нет правого, а у второго наоборот, нет левого поддерева, но есть правое. На этом же рисунке изображено и пустое (невидимое!) двоичное дерево. Таким образом, в бинарном дереве существенное значение имеет наклон ветвей, в то время как в обычном дереве он несущественен. Вторая особенность: бинарное дерево, в отличие от обычного, может быть пустым [72].

Бинарные деревья имеют важное значение в практике программирования; одна из причин этого заключается в том, что обычные деревья часто представляются с помощью специальных классов бинарных деревьев, т. к. их проще хранить и обрабатывать [63].

Важным для информатики классом двоичных деревьев являются деревья арифметических выражений с бинарными и унарными операциями, где каждой операции соответствует вершина, поддеревьями которой являются ее операнды [54]. Рассмотрим

представление выражения  $(a + b/c) * (d - e * f)$  в виде дерева. Подобно математическим формулам выражения в языках программирования представляют собой линейную запись со скобками, которые, вообще говоря, могут быть вложенными и на самом деле имеют существенно нелинейную интерпретацию. Нижеприведенное дерево выражения, оставаясь нелинейным объектом, представляет собой бесскобочную форму записи. Роль скобок, в том числе и отсутствующих(!), играют поддеревья дерева выражения:



В дальнейшем борьба со скобочностью и нелинейностью выражений будет продолжена. Это необходимо для автоматического анализа, интерпретации и вычисления выражений на ЭВМ.

Другие примеры: генеалогическое (семейное) дерево или родословная: родители каждого человека изображаются как его потомки (!), схема спортивного турнира, проходящего по кубковой (олимпийской) системе — проигравший выбывает. Ниже приведен типичный пример турнирного дерева (данные взяты с теннисного турнира Australia Open 2005 <http://2005.australianopen.com>):

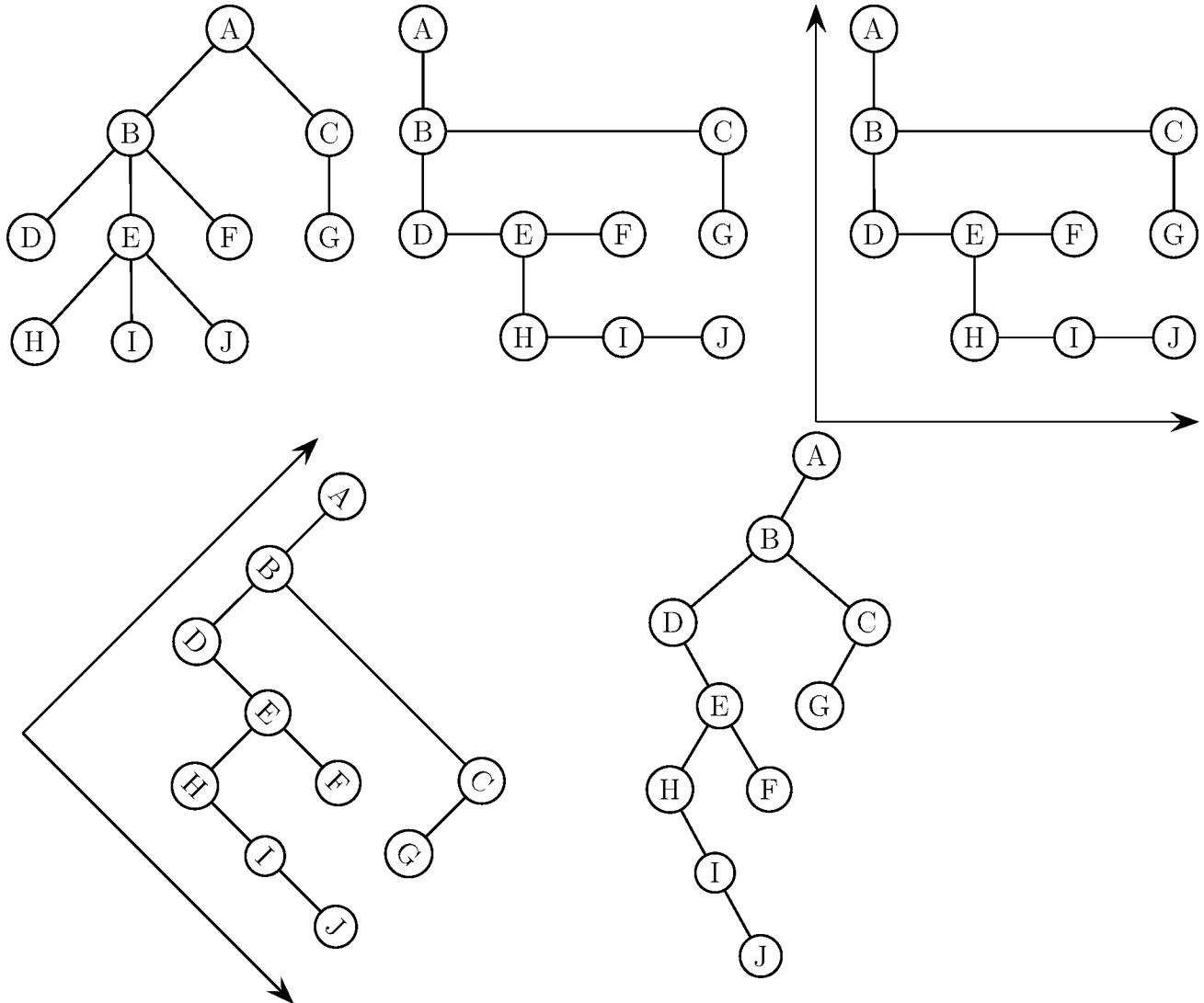


## 5.10.2 Двоичная интерпретация дерева общего вида

Для преобразования произвольного дерева в бинарное, надо в исходном дереве у каждого узла соединить его сыновей (от старшего к ближайшему младшему брату) и убрать все

связи узлов с сыновьями, сохранив лишь связь с первым сыном, за которым выстраивается своеобразная очередь сыновей — нисходящий правосторонний каскад [72].

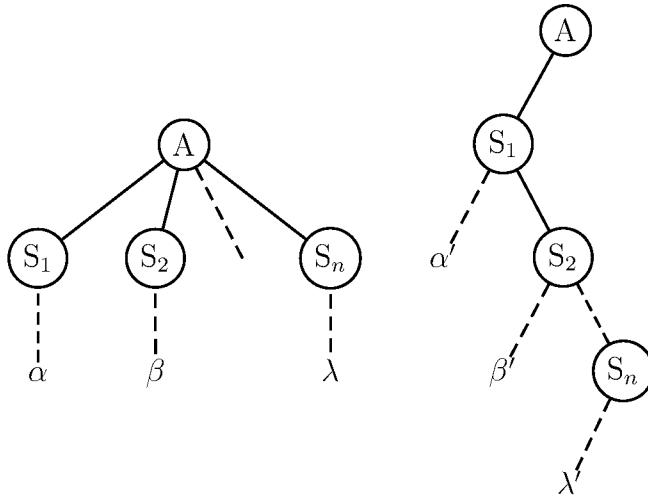
Для получения привычного вида дерева результат надо развернуть на  $45^\circ$  по часовой стрелке:



В общем виде алгоритм преобразования обычного дерева А в двоичное дерево В сводится к рекурсивному применению следующих правил:

- корень В есть корень А;
- левое поддерево двоичного дерева В есть результат этого же преобразования первого поддерева дерева А, если оно существует;
- правое поддерево двоичного дерева В есть результат этого преобразования очередного брата узла А, если он существует.

Этот алгоритм можно проиллюстрировать так [63]:



Здесь  $\alpha', \beta', \dots, \lambda'$  — результаты конвертации соответствующих поддеревьев  $\alpha, \beta, \dots, \lambda$  этим же алгоритмом.

Данный алгоритм применим и в обратную сторону: рекурсивно следуя по каскаду и преобразуя левые поддеревья каждого узла в его сыновей, а правые — в братьев данного узла, получим снова исходное дерево общего вида. Доказательство может быть проведено индукцией по глубине дерева. Для дерева, состоящего из одной вершины, преобразование верно (базис индукции). Теперь предположим, что для  $k$ -уровневого дерева утверждение справедливо (гипотеза индукции). Чтобы преобразовать дерево  $k + 1$ -го уровня, надо сделать левое поддерево сыном, а правое — братом, которые являются деревьями не более чем  $k$ -ого уровня.

**Замечание.** Изложенный рекурсивный алгоритм преобразования любого дерева общего вида в двоичное имеет и концептуальное значение. Подобно теореме Бойма-Джакопини-Миллса он конструктивно доказывает эквивалентность этих двух разновидностей деревьев: двоичные деревья есть деревья ограниченного разветвления аналогично тому, что схемы машин Тьюринга с ограниченным набором управляемых структур были полным эквивалентом диаграмм.

### 5.10.3 Функциональная спецификация

Тип  $BT_T$ , или *двоичное дерево типа  $T$* , определяется так [72]:

СОЗДАТЬ:	$\emptyset \rightarrow BT_T$
ПОСТРОИТЬ:	$BT_T \times T \times BT_T \rightarrow BT_T$
ПУСТО:	$BT_T \rightarrow \text{boolean}$
КОРЕНЬ:	$BT_T \rightarrow T$
СЛЕВА:	$BT_T \rightarrow BT_T$
СПРАВА:	$BT_T \rightarrow BT_T$
УНИЧТОЖИТЬ:	$BT_T \rightarrow \emptyset$

Функция СЛЕВА осуществляет доступ к левому поддереву непустого дерева, а СПРАВА — к правому. Функция ПОСТРОИТЬ создает дерево из корня и двух заданных поддеревьев, одно из которых становится левым, другое — правым.

Это определение подходит под спецификацию дека, хотя моделирование дерева на деке — задача нетривиальная.

Свойства операций:

1. ПУСТО(СОЗДАТЬ) = **true**
2. ПУСТО(ПОСТРОИТЬ( $bt_l, t, bt_r$ )) = **false**
3. КОРЕНЬ(ПОСТРОИТЬ( $bt_l, t, bt_r$ )) =  $t$
4. СЛЕВА(ПОСТРОИТЬ( $bt_l, t, bt_r$ )) =  $bt_l$
5. СПРАВА(ПОСТРОИТЬ( $bt_l, t, bt_r$ )) =  $bt_r$
6. ПОСТРОИТЬ(СЛЕВА( $bt$ ), КОРЕНЬ( $bt$ ), СПРАВА( $bt$ )) =  $bt$

Последнее может быть выражено словами следующим образом: каждое дерево получено построением из своего корня, левым поддеревом имеет свое левое поддерево, а правым поддеревом — свое правое.

Перечислим некоторые операции над деревьями. Операции заданы в самом общем виде, без спецификации параметров:

1. чтение данных из узла дерева;
2. создание дерева, состоящего из одного корневого узла;
3. построение дерева из заданных корня и нескольких поддеревьев;
4. присоединение к узлу нового поддерева;
5. замена поддерева на новое поддерево;
6. удаление поддерева;
7. получение узла, следующего за данным в определённом порядке.

В зависимости от решаемых задач и вида используемых деревьев могут быть предусмотрены и другие операции [63].

#### 5.10.4 Логическое описание

Тип данных дерево обычно отсутствует в универсальных языках программирования, и нам придется его моделировать программно. Детали для этого моделирования, увы, не могут быть заимствованы из готового набора Standard Template Library: ввиду сложности и многообразия деревьев и способов их обработки не существует стандартных универсальных пакетов для этой цели. Впрочем, для этой цели есть более подходящие языки — Лисп и Пролог. Однако, когда мы ограничены фон Неймановскими языками, процедурная реализация средств работы с деревьями не представляет особого труда.

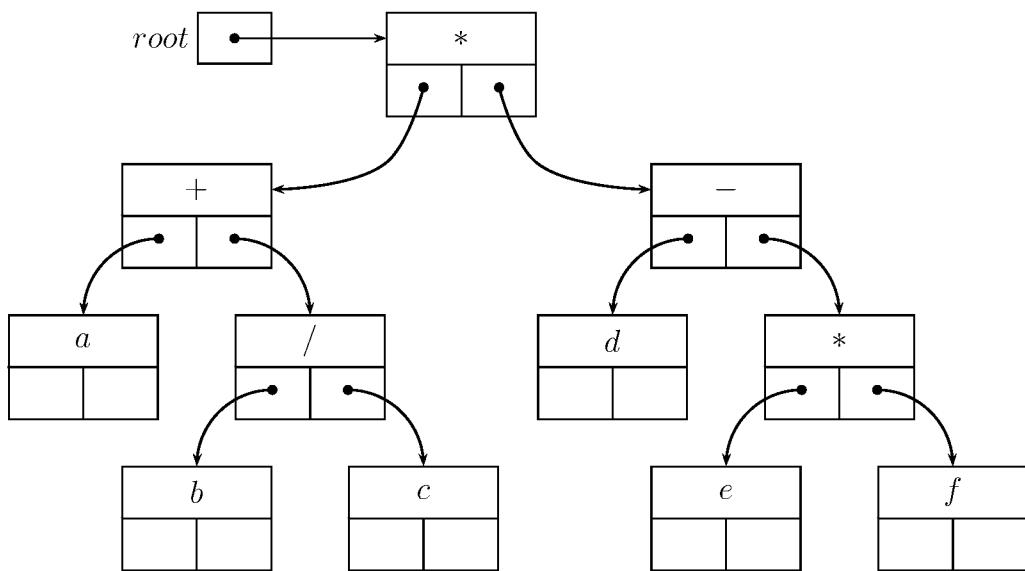
Так же, как и в списках, цепная структура дерева будет состоять из динамически порождаемых элементов, в которых предусмотрены ссылки на очередные компоненты структуры. В двунаправленных списках были указатели вперед и назад, а здесь будут

ссылки влево вниз и вправо вниз. Линейное предшествование и следование заменяется нелинейным разветвлением, которое может ветвиться дальше по каждой из ветвей. Тип данных для узла дерева внешне тот же:

```
type pnode = ^node;
node = record
  key : char;
  l, r : pnode;
end;
```

```
struct node
{
  char key;
  struct node* l;
  struct node* r;
}
```

Динамическая структура данных для вышеописанного дерева выражения может быть изображена так:



Ввиду нелинейности древовидных структур существенно усложняется их обход.

## 5.11 Алгоритмы обработки деревьев

### 5.11.1 Понятие обхода дерева

При анализе структур данных, заданных в виде дерева, применяются разные способы *просмотра* или *перебора* узлов дерева. Такой просмотр называется *обходом* дерева. Основная особенность обхода состоит в том, что просматриваются все без исключения узлы дерева в некотором порядке, причем каждый узел обрабатывается *ровно один раз*. Обход дерева представляет собой линейно упорядоченное рассмотрение всех его вершин, от предыдущего элемента к следующему. Если пройденные вершины заносить в очередь или в список, то этот порядок станет явной линеаризацией дерева. Из структуры двоичных деревьев вытекает три различных дисциплины упорядоченного прохождения вершин [63].

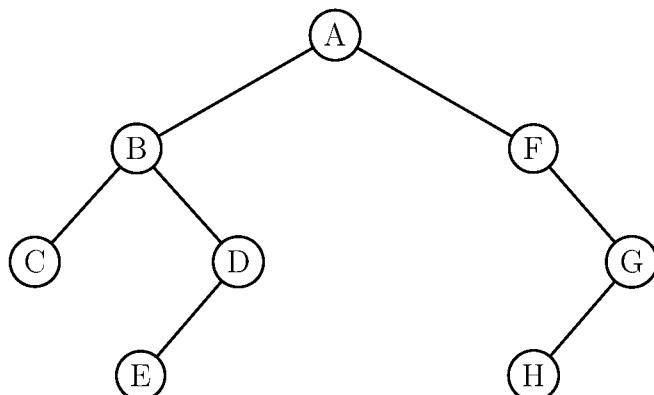
## 5.11.2 Алгоритмы обхода двоичных деревьев

Для бинарного дерева определены три способа обхода: *прямой* (или сверху вниз), *обратный* (или слева направо) и *концевой* (или снизу вверх). Эти способы определяются рекурсивно.

При обходах выполняются следующие действия [44]:

1. при прямом обходе (КЛП, preorder, корень прежде):
  - (a) если дерево пусто, то конец обхода;.
  - (b) берется корень;
  - (c) выполняется обход левого поддерева;
  - (d) выполняется обход правого поддерева.
2. при обратном обходе (ЛКП, inorder, корень между):
  - (a) если дерево пусто, то конец обхода;
  - (b) выполняется обход левого поддерева;
  - (c) берется корень;
  - (d) выполняется обход правого поддерева.
3. при концевом обходе (ЛПК, postorder, корень в конце):
  - (a) если дерево пусто, то конец обхода;
  - (b) выполняется обход левого поддерева;
  - (c) выполняется обход правого поддерева;
  - (d) берется корень.

Для примера дерева, приведенного ниже, при прямом обходе узлы будут взяты в таком порядке: A, B, C, D, E, F, G, H; при обратном обходе узлы будут взяты в таком порядке C, B, E, D, A, F, H, G; при концевом обходе узлы будут взяты в таком порядке: C, E, D, B, H, G, F, A.



При обходе деревьев могут использоваться либо рекурсивные процедуры, либо программы с циклом, зависящим от состояния стека, в котором отражается текущее положение в дереве [36]. Нерекурсивный обход дерева с явным использованием стека аналогичен

рекурсивному исполнению на внутреннем стеке среды языка, обеспечивающем рекурсию. Наконец, существуют такие способы представления деревьев, при которых возможен итеративный обход!

Теперь нетрудно составить рекурсивные процедуры выполнения трёх различных методов обхода двоичного дерева:

```

procedure preorder(t : p);
begin
  if t <> nil then begin
    { обработка вершины: t^.data := ... или write(t^.data) }
    preorder(t^.l);
    preorder(t^.r)
  end
end; {preorder}

procedure inorder(t : p);
begin
  if t <> nil then begin
    inorder(t^.l);
    { обработка вершины }
    inorder(t^.r)
  end
end; {inorder}

procedure postorder(t : p);
begin
  if t <> nil then begin
    postorder(t^.l);
    postorder(t^.r)
    { обработка вершины }
  end
end; {postorder}

```

Во всех трёх рекурсивных процедурах ссылка на текущий корень дерева передаётся по значению, и заносится при рекурсивном выполнении во временной стек экземпляров локальной переменной. При обходе изменение значений этой переменной не производится.

Нерекурсивное выполнение процедуры обхода может быть поддержано рекурсивной структурой данных типа стек.

```

procedure inorder2(tree : p)
var st : Stack; { of p! указателей корневых узлов, для которых начат, но еще не
                  завершен обход левого поддерева }
  done : boolean;
begin
  Create(st);
  done := false;
  repeat
    { Рассматриваемые вершины дерева заносятся в стек до тех пор, пока не

```

```

встретится вершина с пустым левым поддеревом (лист или вершина
только с правым потомком) }

if(tree <> nil) then begin
    Push(st, tree); { Указатель текущего узла помещается в стек }
    tree := tree^.l; { Для продолжения обхода выбирается указатель на левое
    поддерево }

end
else if(Empty(st)) then { В стеке нет вершин для обхода }
    done := true { Установка признака завершения итеративного обхода }
else begin { В стеке есть отложенные вершины для обхода }
    tree := Top(st); { Выбираем для обхода родителя текущего узла,
    находящегося в стеке вслед за потомком }
    Pop(st); { Указатель родителя текущего узла удаляется из стека. В
    результате этой операции в вершину стека поднимается родитель
    удаленного узла }
    { Здесь могла бы быть обработка узла! }
    tree := tree^.right; { Переход к обходу правого поддерева }

end;
until done;
Destroy(st);
end;

```

### 5.11.3 Построение и визуализация дерева

Типичный пример программы, в которой осуществляется обход дерева: рекурсивная процедура ступенчатой распечатки дерева. Но сначала надо создать дерево. Очевидно, непосредственно ввести его из текстового файла непросто. Но мы поступим также, как при вводе многочлена в Паскаль-программу дифференцирования: введём «коэффициенты дерева» — значения узлов в некотором регулярном порядке.

Рассмотрим программу построения дерева из  $n$  вершин, значения которых поступают из входного текстового файла **input**. Если нам нужно построить из этих вершин какое-нибудь дерево, то можно первую из них сделать корнем, вторую — правым поддеревом, третью — правым поддеревом правого поддерева, и т. д. В результате получим линейное дерево (список) максимальной глубины ( $n$ ). Интереснее обратная задача: построить из этих же вершин дерево минимальной глубины. Для этого при построении дерева надо добиться его равномерного заполнения (ау, сплошное турнирное представление!), размещающая приходящие вершины поровну слева и справа от каждой вершины по следующему рекурсивному(!) правилу:

- взять одну вершину в качестве корня;
- построить тем же способом левое поддерево с  $n_l = n \text{ div } 2$  вершинами;
- построить тем же способом правое поддерево с  $n_r = n - n_l - 1$  вершиной.

Этот алгоритм строит идеально сбалансированное дерево, поскольку число вершин в его левых и правых поддеревьях отличается не более, чем на 1. Соответствующая программа на Паскале:

```

program FirstTree(input, output);
{ H. Bupm }
type p = ^node;
  node = record
    key : char;
    l, r : p
  end;

var n : integer;
  root : p;

function buildtree(n : integer): p;
var node : p;
  x, nl, nr: integer;
begin { построение идеально сбалансированного дерева }
  if n = 0 then
    node := nil
  else begin { распределение вершин налево и направо }
    nl := n div 2;
    nr := n - nl - 1;
    read(x);
    new(node);
    with node^ do begin
      key := x; { переназначение построения двух полудеревьев }
      l := buildtree(nl);
      r := buildtree(nr)
    end;
  end;
  buildtree := node
end; {buildtree}

procedure printtree(t : p; h : integer);
begin { ступенчатая печать дерева с отступом, равным глубине узла }
  if t <> nil then
    with t^ do begin
      printtree(l, h + 1);
      writeln('`' : h + 1, key : 3);
      printtree(r, h + 1)
    end;
  end;
end; {printtree}

begin {program}
  read(n);
  root := buildtree(n);
  printtree (root, 0)

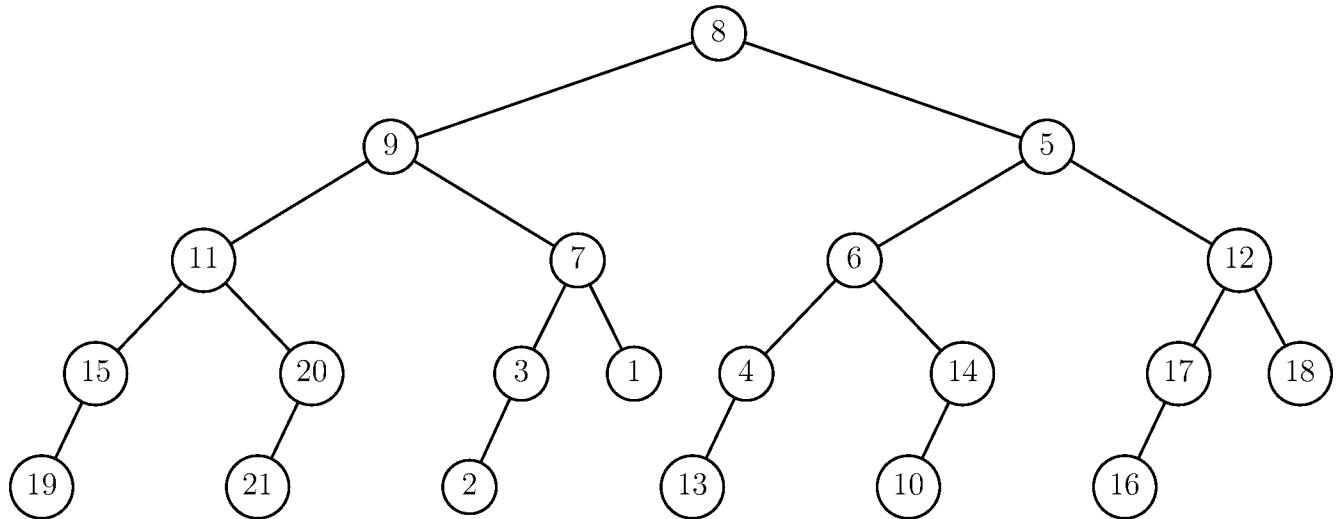
```

**end.** {program}

Если на вход этой программы подать последовательность

21 8 9 11 15 19 20 21 7 3 2 1 5 6 4 13 14 10 12 17 16 18

то будет построено идеально сбалансированное дерево из 21 вершины:



Программа напечатает т. н. ASCII-визуализацию дерева, лежащего на боку и без графических излишеств — рёбер и вершинных кружков, но с сохранением топологии.

Простота и ясность этой рекурсивной программы — лучшее доказательство уместности рекурсии, рекурсия выполнения хорошо отражает рекурсию структуры дерева. В мелочах тоже всё гладко: дерево из 0 элементов не строится и не печатается.

#### 5.11.4 Деревья выражений. Разнофиксные формы записи выражений

Выражения:

- — арифметические;
- — логические;
- — информатические.

Нелинейность и рекурсивность выражений. Фон Неймановская нежелательность скобочных структур.

Деревья выражений. Фон Неймановская причина их бинарности.

Бесскобочность представления выражений деревьями.

Разнофиксные формы записи выражений:

- — префиксная;
- — инфиксная;
- — постфиксная.

Их связь с обходами деревьев выражений.

Фон Неймановские преимущества бесскобочкой записи.

Пример: псевдокод программы построения и распечатки дерева выражения по его линейной скобочной (инфiksной записи).

Особенности примера:

- упрощённые арифметические выражения;
- рекурсивно-итеративная интерпретация при удалении распечатанного дерева.

Полурекурсивная версия: сочетание рекурсивного спуска направо с итеративной обработкой слева.

### **Реализация парсера по грамматике. Нормальный алгоритм.**

```
; Наглядный пример грамматики простого предшествования
; на основе грамматики: page 476 АНО (volume 1)
; Copyright (c) Иваницкий Николай, 1997.
; [E] ] <=> A
; ВХОД: На ленте - выражение, заключённое с двух сторон в символы $
; ВЫХОД: YES!, если цепочка допускается грамматикой.
#LENTA $(((a+a)*a+a*a)+((a+a)*a+a*a))$
```

; Отношения предшествования:

```
T) -> T>
T+ -> T>+
F) -> F>
F+ -> F>+
F* -> F>*
[T]) -> [T]>
[T]+ -> [T]>+
A) -> A>
A+ -> A>+
A* -> A>*
a) -> a>
a+ -> a>+
a* -> a>*
)) -> )>
)+ -> )>+
```

```

)* -> )>*
(E -> (<E
(T -> (<T
(F -> (<F
([T] -> (<[T]
(a -> (<a
( + -> (<+
(( -> (<(
+T -> +<T
+F -> +<F
+a -> +<a
+( -> +<(
*a -> *<a
*( -> *<(
T$ -> T>$
F$ -> F>$
[T]$ -> [T]>$
A$ -> A>$
a$ -> a>$
)$ -> )>$
$E -> $<E
E$ -> E>$
$T -> $<T
$F -> $<F
$[T] -> $<[T]
$a -> $<a
$( -> $<(
$+ -> $<+
; Правила свёртки (обычные продукции, вывернутые наизнанку)
<E+[T]> -> E
<+[T]> -> E      Грамматика взята из
<[T]> -> E      первого тома Axo (page 476)
<T> -> [T]
<T*F> -> T
<(A> -> F
<F> -> T
<a> -> F
<E)> -> A
<E> ->
; приведём ко всюду определённому алгоритму:
$$ -> . YES!
> -> . _Error:_пропущено_начало_основы_
< -> . _Error:_пропущен_конец_основы!_
-> . _General_fault_error!_
;
; P.S. а ведь если расширить синтаксис NAM до

```

```

;      a1 -> a2 [: b1 -> b2]
; причём команда после : выполняется <= выполняется команда до :,
; то получится

```

Другой пример: Преобразование выражения из инфиксной формы в постфиксную. Нерекурсивная версия.

### Алгоритм Рутисхаузера (1951).

«Танцевальная процессия вокруг скобочных скал».

Один из наиболее ранних алгоритмов.

Предполагает полную скобочную структуру выражения – такую форму записи, при которой порядок действий задается расстановкой скобок.

Неявное старшинство операций при этом не учитывается. Например:

$$D = ((C-(B*L))+K)$$

Обрабатывая выражение с полной скобочной структурой, алгоритм присваивает каждому символу – лексеме исходного выражения – номер уровня по следующему правилу:

- – если это открывающаяся скобка или переменная, то значение уровня увеличивается на 1;
- – если лексема – знак операции или закрывающаяся скобка, то уровень уменьшается на 1.

Для выражения  $(A + (B + C))$  значения уровней таковы:

№ литеры	1	2	3	4	5	6	7	8	9
Символы (лексемы!)	(	A	+	(	B	*	C	)	)
Номера уровней	1	2	1	2	3	2	3	2	1

Основные этапы алгоритма Рутисхаузера:

1. расставить уровни;
2. отыскать элементы строки с максимальным значением уровня;
3. выделить тройку – два операнда с максимальным значением уровня и операцию, которая заключена между ними;
4. результат выполнения выделенной операции обозначить вспомогательной переменной;

5. из исходной строки удалить выделенную тройку вместе с её скобками, а на ее место поместить вспомогательную переменную, обозначающую результат, со значением уровня на единицу меньшим, чем у выделенной тройки;
6. выполнять п.п. 2 - 5 до тех пор, пока во входной строке не останется одна переменная, обозначающая общий результат выражения.

Пример разбора:

Генерируемые тройки Выражение

	$( ( ( A + B ) * C ) / D ) - E )$
	0 1 2 3 4 5 4 5 4 3 4 3 2 3 2 1 2 1 0
+ A B - > R	$( ( ( R * C ) / D ) - E )$
	0 1 2 3 4 3 4 3 2 3 2 1 2 1 0
* R C -> S	$( ( S / D ) - E )$
	0 1 2 3 2 3 2 1 2 1 0
/ S D -> Q	$( Q - E )$
	0 1 2 1 2 1 0
- Q E -> T	T

### Алгоритм Бауэра и Замельзона

Ранний стековый метод.

Используются два стека и таблица функций перехода. Один стек (T) используется при трансляции выражения, а второй (E) – во время интерпретации выражения.

В таблице переходов задаются функции, которые должен выполнить транслятор при разборе выражения:

- f1: заслать операцию из входной строки в стек T; читать следующий символ строки;
- f2: выделить тройку – взять операцию с вершины стека T и два операнда с вершины стека E; вспомогательную переменную – результат операции занести в стек E; заслать операцию из входной строки в стек T; читать следующую лексему строки;
- f3: исключить лексему из стека T; читать следующий символ строки;
- f4: выделить тройку – взять операцию с вершины стека T и два операнда с вершины стека E; вспомогательную переменную – результат операции, занести в стек E; по таблице определить функцию для данной лексемы входной строки;

- f5: выдача сообщения об ошибке;
- f6: завершение работы.

Таблица переходов для алгебраических выражений будет иметь вид (знак \$ является признаком пустого стека или пустой строки):

	\$	(	+	-	*	/	)
Операция	\$	6	1	1	1	1	5
на вершине	(	5	1	1	1	1	3
стека Т	+	4	1	2	2	1	4
	-	4	1	2	2	1	4
	*	4	1	4	4	2	4
	/	4	1	4	4	2	4

Выражение просматривается слева направо и над каждой лексемой выполняются следующие действия: если очередная лексема входной строки является операндом, то она безусловно переносится в стек E; если это операция, то по таблице функций перехода определяется номер функции для выполнения.

Для выражения  $A + (B - C) * D$  будут проделаны следующие действия:

Стек E	Стек Т	Лексема	Функция	Тройка
\$	\$	A		
\$A	\$	+	1	
\$A	\$+	(	1	
\$A	\$+(	B		
\$AB	\$+(	-	1	
\$AB	\$+(-	C		
\$ABC	\$+(-)		4	- B C -> R
\$AR	\$+()		3	
\$AR	\$+	*	1	
\$AR	\$+*	D		
\$ARD	\$+*	\$	4	* R D -> Q
\$AQ	\$+	\$	4	+ A Q -> S
\$S	\$	\$	конец	

**Алгоритм сортировочной станции Дейкстры** — способ разбора математических выражений, представленных в обычной инфиксной нотации.

Результат — в виде обратнойпольской записи или в виде дерева выражения. Алгоритм изобретен Эдсгером Дейкстрой и назван им алгоритмом сортировочной станции, поскольку напоминает действие железнодорожной сортировочной станции. Имеется байка

про вагоны с туалетами. Так же, как и при вычислении значений выражений в обратной польской записи, алгоритм работает при помощи стека. Для преобразования в обратную польскую нотацию используется 2 очереди: входная и выходная, и стек для хранения операций, еще не добавленных в выходную очередь. При преобразовании выражения считывается лексема и производятся действия, зависящие от её конкретного вида.

Пока не все синтаксические символы (лексемы) обработаны:

Считать лексему.

Если это число, то добавить в очередь вывода.

Если — функция, то поместить в стек.

Если — разделитель аргументов функции (например запятая), то:

пока символ на вершине стека не открывающая скобка,  
перекладывать операции из стека в выходную очередь.

Если в стеке не было открывающей скобки,  
то в выражении пропущен разделитель аргументов  
функции (запятая), либо пропущена открывающая скобка.

Диагностика!

Если — операция op1, то:

пока присутствует на вершине стека лексема операции op2, и  
либо операция op1 левоассоциативна и её приоритет  
меньше чем у операции op2 либо равен ему,  
или операция op1 правоассоциативна и её приоритет  
меньше чем у op2,  
переложить op2 из стека в выходную очередь;  
положить op1 на стек.

Если — открывающая скобка, то положить её на стек.

Если — закрывающая скобка:

пока на вершине стека не открывающая скобка,  
перекладывать операции из стека в выходную очередь.  
вынуть открывающую скобку из стека, но не добавлять в  
выходную очередь.  
если на вершине стека лексема функции, добавить её в  
выходную очередь.  
если стек закончился до того, как была встречена  
открывающая скобка, то в выражении пропущена скобка.

Диагностика!

Если очередь лексем на входе пуста:

пока есть операции в стеке:

если операция на вершине стека — скобка, то в

выражении присутствует незакрытая скобка.

Диагностика!

Переложить операцию из стека в выходную очередь.

Конец!!!

Каждая лексема-число, функция или операция выводится только один раз.

Каждый символ-функция, операция или круглая скобка будут добавлены и удалены из стека по одному разу.

Постоянное количество операций на лексему, линейная сложность алгоритма  $O(n)$ .

### **Простой пример:**

Входная очередь лексем:  $3 + 4$

Добавим 3 в выходную очередь (если прочитан операнд, то он сразу направляется на выход).

Помещаем + в стек операций.

Операнд 4 следует на выход.

Выражение прочитано, все оставшиеся в стеке операции выталкиваются на выход. В нашем примере в стеке содержится только +.

Выходная строка:  $3 \ 4 \ +$

В данном примере проявляются некоторые правила: все числа переносятся в выходную строку сразу после прочтения; когда выражение прочитано полностью, все оставшиеся в стеке операции выталкиваются в выходную строку. Операнды для них занесены туда ранее!

### **Сложный пример.**

Приоритеты:

- ^ высокий
- \* / средний
- + - низкий

Трасса процесса обработки выражения  $3 + 4 * 2 / (1 - 5)^2$ :

Лексема: 3.

Тип лексемы – операнд.

3 идёт в выходную очередь

Выход: 3

Лексема: +

Операция.

+ идёт на стек.

Выход: 3

Стек: +

4

Операнд

4 идёт на выход

Выход: 3 4

Стек: +

\*

Операция, кладём \* на стек

Выход: 3 4

Стек: + \*

2

Операнд, 2 идёт на выход

Выход: 3 4 2

Стек: + \*

/

Выталкиваем \* из стека на выход, кладём / на стек

Выход: 3 4 2 \*

Стек: + /

(

Открывающая скобка ( идёт в стек

Выход: 3 4 2 \*

Стек: + / (

1

Операнд 1 идёт на выход

Выход: 3 4 2 \* 1

Стек: + / (

-

Операция - в стек

Выход: 3 4 2 \* 1

Стек: + / ( -

5

5 идёт на выход

Выход: 3 4 2 \* 1 5

Стек: + / ( -

)

Выталкиваем - из стека в выходную очередь,

Выталкиваем (

Выход: 3 4 2 \* 1 5 -

Стек: + /

$\wedge$  символ возведения в степень (\*\* в Фортране!)

$\wedge$  – на стек

Выход: 3 4 2 \* 1 5 -

Стек: + /  $\wedge$

2

Операнд 2 идёт на выход

Выход: 3 4 2 \* 1 5 - 2

Стек: + /  $\wedge$

Конец выражения

Выталкиваем все элементы из стека в выходную очередь (инверсия, однако!)

Выход: 3 4 2 \* 1 5 - 2  $\wedge$  / +

Программа на языке Си.

Обходя дерево выражения разными способами, мы получим три различные очереди вершин [43]:

- КЛП: \* + a / b c - d \* e f;
- ЛКП: a + b / c \* d - e \* f;
- ЛПК: a b c / + d e f \* - \*.

которые представляют собой ни что иное, как широко используемые в информатике различные формы записи выражений: префиксную (ассемблер, код операции *перед* (*pre*) операндами, называемую также польской), привычную инфиксную (математическую, знак операции *между* (*in*) операндами), но без скобок, задающих порядок выполнения операций, и постфиксную (обратную польскую). Бесскобочная форма записи представляет собой линейную последовательность действий, пригодную для выполнения обычным фон Неймановским компьютером. История бесскобочной записи такова:

«Польский математик и логик Ян Лукашевич печатал в 1920 г. свою докторскую диссертацию на старой немецкой пишущей машинке, у которой было не две клавиши переключения регистров, а три: строчные буквы и знаки препинания, прописные буквы, а также все скобки и другие специальные символы. Вместо того, чтобы вынужденно использовать все три клавиши переключения регистра (старая, с грохотом работавшая пишущая машинка часто заедала при переключении регистров), Лукашевич изобрел запись, которая не

требовала применения скобок. Для этого обычная запись *операнд оператор operand* была заменена на запись *оператор operand operand*. Таким образом, обычная запись  $A + B$  превратилась в запись  $+ A B$  [86].

Рассмотрим программу построения и распечатки дерева выражения по его линейной скобочной (инфиксной) записи.

```

program expr2tree(input, output);
{Лебедев А. В.}
{ Синтаксис выражения
expr -> term / expr + term / expr - term
term -> fact / term * fact / term / fact
fact -> (expr) / digit / letter
digit -> 0 / 1 / 2 / 3 / 4 / 5 / 6 / 7 / 8 / 9
letter -> a / b / ...           ... / z
}

type tree = ^node;
node = record
  data : char;
  l, r : tree;
end;

var exprtree : tree;

function isalnum(ch : char) : boolean;
begin
  isalnum := ((ch >= 'a') and (ch <= 'z')) or ((ch >= '0') and (ch <= '9'))
end;

function makenode(c : char; l, r : tree) : tree;
var t : tree;
begin
  new(t);
  t^.data := c;
  t^.l := l;
  t^.r := r;
  makenode := t
end;

procedure destroytree(t : tree);
var p : tree;
begin { рекурсивно-итеративное уничтожение дерева }

```

{рекурсивный спуск влево внутри итеративного спуска вправо}

```
while t <> nil do begin
    destroytree(t^.l);
    p := t;
    t := t^.r;
    dispose(p)
end;
end;
```

```
procedure printtree(t : tree);
```

```
procedure print(t: tree; tab : integer);
var i : integer;
begin
    if t^.r <> nil then
        print(t^.r, tab + 1);
    for i := 0 to tab - 1 do
        write(' ');
    writeln(t^.data);
    if t^.l <> nil then
        print(t^.l, tab + 1);
end;
```

{полурекурсивная версия процедуры распечатки: рекурсивный спуск по правой  
ветке сочетается с итеративным по левой}

```
{procedure print(t: tree ; tab : integer);
var
    i : integer;
begin
repeat
    if t^.r <> nil then
        print(t^.r, tab + 1);
    for i := 0 to tab - 1 do
        write(' ');
    writeln(t^.data);
    t := t^.l;
    tab := tab + 1;
until t = nil;
end;}
```

```
begin {уменьшим вложенность рекурсии}
```

```
if t <> nil then
    print(t, 0);
end;
```

```

function parse : tree; { разбор выражения }
var ch : char;

function expr : tree; forward; { разбор подвыражения }
{опережающее обявление процедуры для косвенной рекурсии
expr -> term -> fact -> expr}

function fact: tree; { разбор множителя/делимого/делителя }
begin
  read(ch);
  if ch = '(' then begin { множитель содержит подвыражение }
    fact := expr;
    { if ch <> ')' then диагностика отсутствия закрывающей скобки}
  end
  else if isalnum(ch) then
    fact := makenode(ch, nil, nil) { создание терминальной вершины (листа)
      для переменной или константы }
    { else диагностика некорректной литеры}
  end;

function term : tree; { Разбор слагаемого/умножаемого/вычитаемого }
var
  tm : tree;
  done : boolean;
begin
  tm := fact;
  done := false;
  while not eoln and not done do begin
    read(ch);
    if (ch = '*') or (ch = '/') then
      tm := makenode(ch, tm, fact) { Подвеска поддерева подвыражения с
        рекурсивным разбором очередного подвыражения }
    else
      done := true;
  end;
  term := tm;
end;

function expr; { разбор подвыражения: ранее отложенное описание тела
  процедуры }
var
  ex : tree;
  done : boolean;
begin
  ex := term;

```

```

done := false;
while not eoln and not done do begin
  if (ch = '+') or (ch = '-') then
    ex := makenode(ch, ex, term) { Подвеска поддерева подвыражения с
      рекурсивным разбором очередного подвыражения }
  else
    done := true;
  end;
  expr := ex;
end;

begin
  parse := expr;
end; {parse}

begin {program}
  exprtree := parse;
  printtree(exprtree);
  destroytree(exprtree);
end.

```

Другой пример — преобразование выражения из инфиксной формы в постфиксную.

```

program in2post(input, output);

procedure Create(var s : Stack);
function Empty(var s : Stack) : boolean;
function Size(var s : Stack) : integer;
function Push(var s : Stack; c : char) : boolean;
function Pop(var s : Stack) : boolean;
function Top(var s : Stack) : char;
procedure Destroy(var s : Stack);

var st : Stack; { При преобразовании из инфиксной формы в постфиксную помимо
  выноса знака операции назад, необходимо упорядочивать операции согласно их
  приоритетам, в том числе и в связи с отказом от скобок. Для необходимой
  инверсии порядка выполнения операций и будет использован стек }

c, z : char;
i : integer;

{ Функция сравнивает приоритеты операций, оказавшихся рядом в обратной
 польской записи }
function prcd(c1, c2 : char) : integer;
begin
  prcd := 0;
  { Умножение и деление приоритетнее сложения и вычитания }
  if (c1 in ['*', '/']) and (c2 in ['+', '-']) then prcd := 1;

```

```

if (c1 in ['+', '-']) and (c2 in ['*', '/']) then prcd := -1;
{ Одинаковые операции имеют одинаковый приоритет }
if (c1 in ['+', '-']) and (c2 in ['+', '-']) then prcd := 0;
if (c1 in ['*', '/']) and (c2 in ['*', '/']) then prcd := 0;
{ Открывающая скобка имеет высший приоритет (множественный) ... }
if (c2 = '(') or (c1 = '(') then
    prcd := 1;
{ ... а закрывающая — низший (аддитивный) }
if c1 = ')' then
    prcd := -1;
end;

{ Преобразование константного выражения из десятичных цифр в постфиксную
запись }

begin
    Create(st); { Создание стека откладываемых операций }
    while not eoln do begin
        read(c);
        if c in ['0' .. '9'] then { операнд }
            write(c)
        else begin
            { Знак операции and (стек пуст (отложенных операций нет) or текущая
            операция формулы приоритетнее отложенной в стек) }
            if (c in ['+', '-', '*', '/']) and (Empty(st) or (prcd(c, Top(st)) > 0)) then
                Push(c) { Операция заносится в стек }
            else begin
                while (not Empty(st)) and (prcd(c, Top(st)) <= 0) do begin
                    { Вынимаем из стека операцию и печатаем ее, если она не
                    приоритетнее текущей операции с }
                    z := Top(st);
                    Pop(st);
                    { ... если только это не скобка }
                    if z <> '(' then
                        write(z);
                    end;
                end;
                { Помещаем в стек текущую операцию, если только это не скобка }
                if c <> ')' then
                    push(c);
                end;
            end;
        end;
    end;
    { Опустошаем стек, печатая отложенные операции в порядке, обратном их
    поступлению }
    while not Empty(st) do begin
        if Top(st) <> '(' then begin
            write(Top(st));
        end;
    end;
end;

```

```

    Pop(st);
end;
end;
writeln;
Destroy(st);
end.
```

### 5.11.5 Обход дерева общего вида

Деревья общего вида имеют более сложную структуру, чем двоичные, и их обход в силу произвольности числа разветвлений каждого узла не может быть также легко перенаправлен по двум альтернативам налево и направо. Требуется цикл или рекурсия по переменному числу разветвлений. Чтобы перебрать всех сыновей, надо просмотреть до конца их очередь. При этом возможны два приоритета: сначала перебираются братья (обход в ширину) или сыновья (обход в глубину). Если представлять дерево общего вида как двоичное, то поиск в глубину аналогичен КЛП-обходу. Для обхода в ширину двоичного аналога нет.

1. при поиске в глубину:

- (a) если дерево пусто, то конец обхода;
- (b) берется корень;
- (c) выполняется поиск в глубину для поддерева старшего сына;
- (d) выполняется поиск в глубину для следующего брата.

2. при поиске в ширину:

- (a) поместить в пустую очередь корень дерева;
- (b) если очередь узлов пуста, то конец обхода;
- (c) извлечь первый элемент из очереди узлов и поместить в ее конец всех его сыновей по старшинству;
- (d) повторить поиск начиная с п. 2b.

## 5.12 Деревья поиска

Двоичные деревья часто употребляются для представления множеств данных, элементы которых должны быть найдены по некоторому ключу (полю данных) [54]. Если дерево организовано так, что для каждой вершины  $t_i$  справедливо утверждение, что все ключи левого поддерева  $t_i$  меньше ключа  $t_i$ , а все ключи правого поддерева  $t_i$  больше его, то такое дерево называют *деревом поиска*. В таком дереве можно быстро обнаружить элемент с заданным ключом: надо начиная с корня двигаться к левому или правому поддереву на основании лишь одного сравнения с ключом текущей вершины. По построению дерева поиска, переходя к одному из поддеревьев, мы автоматически исключаем из рассмотрения другое поддерево, содержащее половину узлов. При дальнейшем рассмотрении исключается половина половины и т. д. Тем самым будет выполнено исключение из рассмотрения

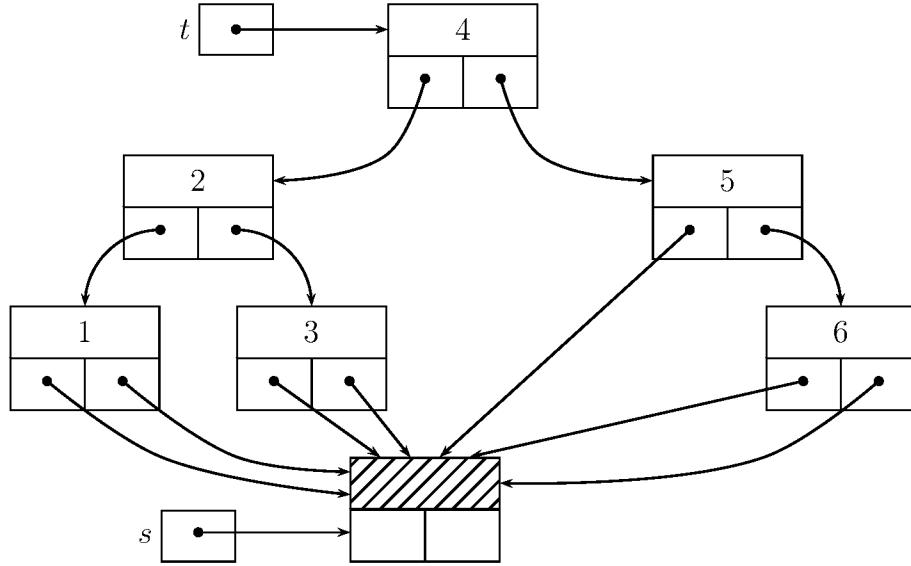
подавляющего большинства элементов дерева, кроме вершин, по которым идет принятие решения о направлении поиска, причем чем сбалансированнее дерево, тем большее число узлов будет исключено из поиска. Число таких вершин для сбалансированного двоичного дерева не может быть больше его глубины, оцениваемой как  $\log_2 n$ . Поэтому деревья поиска дают существенный выигрыш во времени поиска по сравнению с линейными структурами:  $\log_2 n \ll n$ . Например, для  $n = 2048$  выигрыш будет почти в 100 раз, и минуты превратятся в секунды! Более того, поскольку поиск идёт от корня дерева и до его листа по однозначно выбираемому пути, то для его реализации достаточно одного цикла с предусловием, а рекурсия и её верный оруженосяц стек могут отдохнуть:

```
function locate(x : integer; t : p) : p;
begin
  while (t <> nil) and (t^.key <> x) do
    if t^.key < x then
      t := t^.r
    else
      t := t^.l;
    locate := t
  end;
```

В случае неуспеха, когда в дереве с корнем  $t$  не было обнаружено ключа со значением  $x$ , функция  $locate(x, t)$  получает значение **nil**.

В эту процедуру поиска может быть внесено усовершенствование. Для упрощения условия поиска можно, также как и в списках, ввести терминирующий элемент  $s \uparrow$ , к которому гамаком [54] привязать пустые (неиспользуемые, кроме как для прошивки) ссылки всех концевых вершин дерева поиска. В результате из условия поиска удаляются два из трёх сравнений. В предусловии цикла остаётся только сравнение с барьерным элементом. Правда, в случае неудачного поиска функция примет значение не **nil**, а будет указывать на якорь нашей древовидной структуры, что придётся учесть при использовании модифицированной процедуры  $locate()$ .

```
function locate(x : integer; t : p) : p;
begin
  s^.key := x; { барьерное значение заносится в барьерный элемент }
  while {(t <> nil) and } (t^.key <> x) do { условие упрощено на 2/3! }
    if t^.key < x then
      t := t^.r
    else
      t := t^.l;
    locate := t
  end;
```



Справедливо ради необходимости заметить, что, ускоряя поиск, техника барьерного элемента усложняет более редкие, чем поиск, процедуры вставки и удаления элементов.

### 5.12.1 Поиск по дереву с включениями

Динамическое распределение памяти особенно удобно там, где структуры данных нелинейны и меняются с течением времени. Рассмотрим сначала случай, когда дерево поиска только растёт. Хрестоматийная задача такого рода — построение и обслуживание частотного словаря [54]. Требуется определить частоту вхождения каждого из слов в последовательность, поступающую из входного текстового файла. Надо либо найти слово в дереве и добавить 1 к находящемуся в соответствующем узле счётчику его вхождений, либо, в случае неуспеха, включить новый элемент в дерево с сохранением нелинейного иерархического порядка и поисковой структуры и установить для него единичное значение счётчика.

Пусть элемент дерева поиска имеет следующее рекурсивное описание:

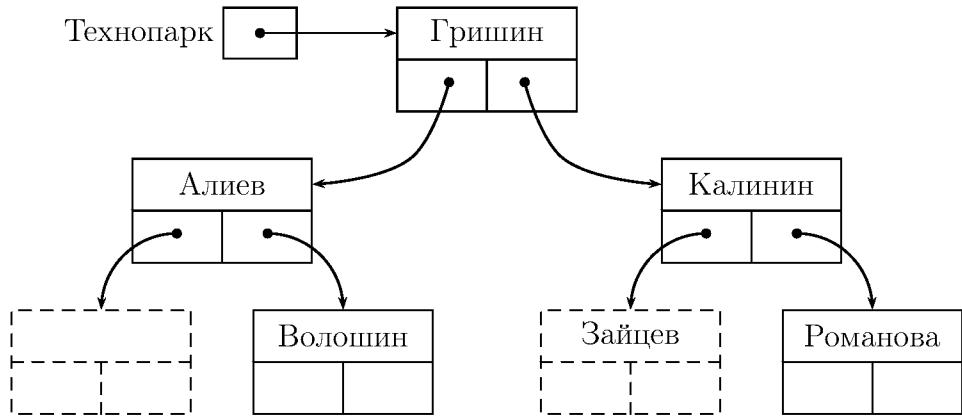
```
type wp = ^word;
word = record
  key : integer;
  count : integer;
  l, r : wp
end;
```

Программа поиска по дереву с включениями может выглядеть так.

```
readln(x);
while not eof do begin
  search(x, root);
  readln(x)
end;
```

Процедура поиска с включением *search()* сначала должна осуществлять поиск в дереве аналогично процедуре *locate()*. В случае неудачи поиск оканчивается на одном из листьев

дерева, к которому привела попытка поиска. Поскольку все предыдущие проверки на пути к этому листу пройдены, то мы имеем готовый маршрут поиска нового элемента, а данный лист представляет собой искомое место для вставки этого элемента в поисковое дерево.



```

program treesearch(input, output);

type wp = ^word;
word = record
    key : integer;
    count : integer;
    l, r : wp
end;

var root: wp;
n, key: integer;

procedure printtree(t : p; h : integer); {см. выше}

procedure search(x : integer; var p : wp);
begin
  if p = nil then begin { слова в дереве нет, включение }
    new(p);
    with p^ do begin
      key := x;
      count := 1;
      l := nil;
      r := nil
    end { продолжение поиска }
  else if x < p^.key then
    search(x, p^.l)
  else if x > p^.key then
    search(x, p^.r)
  else
    { поиск окончен, фиксация еще одного вхождения искомого слова }
  end
end;

```

```

    p^.count := p^.count + 1
  end
end; {search}

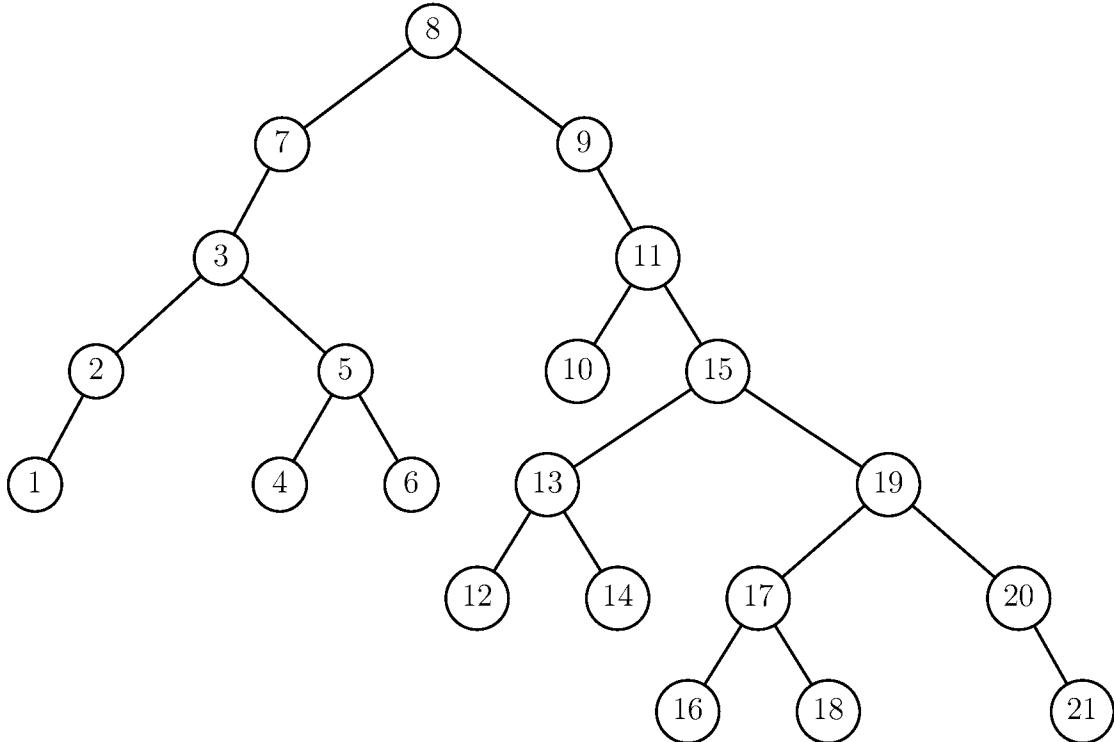
```

```

begin {program}
root := nil;
while not eof do begin
  readln(key);
  search(key, root)
end;
printtree (root, 0)
end.

```

Параметр  $p$  передаётся в процедуру  $search$  не по значению, а как переменная, поскольку в случае включения нового элемента в дерево поиска необходимо вернуть ссылку на этот элемент, как будто бы он только что был найден в модифицированном дереве. Если подать этой программе вышеописанную последовательность из 21 числа, то мы получим двоичное дерево поиска, топология и ординальность которого будут совсем другими:



Дерево поиска может быть использовано для упорядочивания данных. Для этого надо разместить эти данные в дереве поиска, а потом составить в результате его обхода упорядоченную последовательность. Но для решения этой задачи обычное дерево поиска не подходит: оно не допускает вхождения равнозначных элементов. Поэтому всякий новый элемент теперь должен быть включён в дерево. Для этого случай  $x = p^.key$  необходимо рассматривать вместе с одним из других. Если его объединить с  $x > p^.key$ , то порядок следования элементов с одинаковыми ключами совпадёт с хронологией их поступления в дерево. Возможность сочетания поиска и упорядочивания растущего множества данных, предоставляемая деревом поиска с включениями, даёт новое качество по сравнению

с хранением данных как в списках, так и в массивах. Например, этим способом можно эффективно решить задачу составления таблицы перекрёстных ссылок слов текста [54].

### 5.12.2 Исключение из деревьев

Перейдём теперь к решению обратной задачи — задачи исключения элементов из упорядоченного дерева поиска: необходимо найти и исключить из дерева элемент с ключом  $x$ . Если исключаемая вершина в дереве есть, то она может быть либо листом, либо внутренней вершиной с одним или с двумя потомками. Последний случай представляет определённую трудность в реализации. После изъятия элемента с двумя потомками одна ссылка на него оказывается на разветвлении к потомкам. По построению дерева поиска значение удаляемого узла таково, что в его левом поддереве находятся меньшие элементы, а в правом — большие. При этом самый правый элемент левого поддерева больше любого другого элемента этого поддерева и одновременно меньше любого элемента соседнего с левым правого поддерева. И наоборот, самый левый элемент правого поддерева меньше любого из элементов этого поддерева и больше любого элемента из соседнего левого поддерева. Таким образом, любой из этих элементов может быть помещен вместо удаляемой вершины, и порядок в дереве поиска сохранится. Итак, удаляемый элемент заменяется либо на самый правый элемент его левого поддерева, либо на самый левый элемент его правого поддерева, причём эти элементы не могут иметь больше одного потомка, как максимальный и минимальный элементы соответствующих поддеревьев. Все эти случаи предусматривает рекурсивная процедура *delete*:

```

procedure delete(x : integer; var p : wp);
var q : wp;

procedure del(var t : wp);
begin { Обработка случая двух потомков удаляемой вершины — поиск узла без
        правого поддерева }
  if t^.r <> nil then { Существует элемент больше данного }
    del(t^.r) { Продолжаем поиск релевантной вершины в онтологически
               градиентном направлении }
  else begin { Нужный элемент найден, все полезные данные релевантного
             элемента копируются в узел «удаляемого» элемента, после чего
             удаляется тот элемент, откуда были взяты данные }
    q^.key := t^.key;
    q^.count := t^.count;
    q := t; { Глобальной ссылке присваивается локальный указатель
              релевантного узла }
    t := t^.l { Подаем левого под дерева релевантного узла, если оно есть, на
               место перемещенного элемента с целью устранения разрыва в дереве
               поиска }
  end
end; {del}

begin {delete}
  if p <> nil then { Поиск удаляемого элемента продолжается }

```

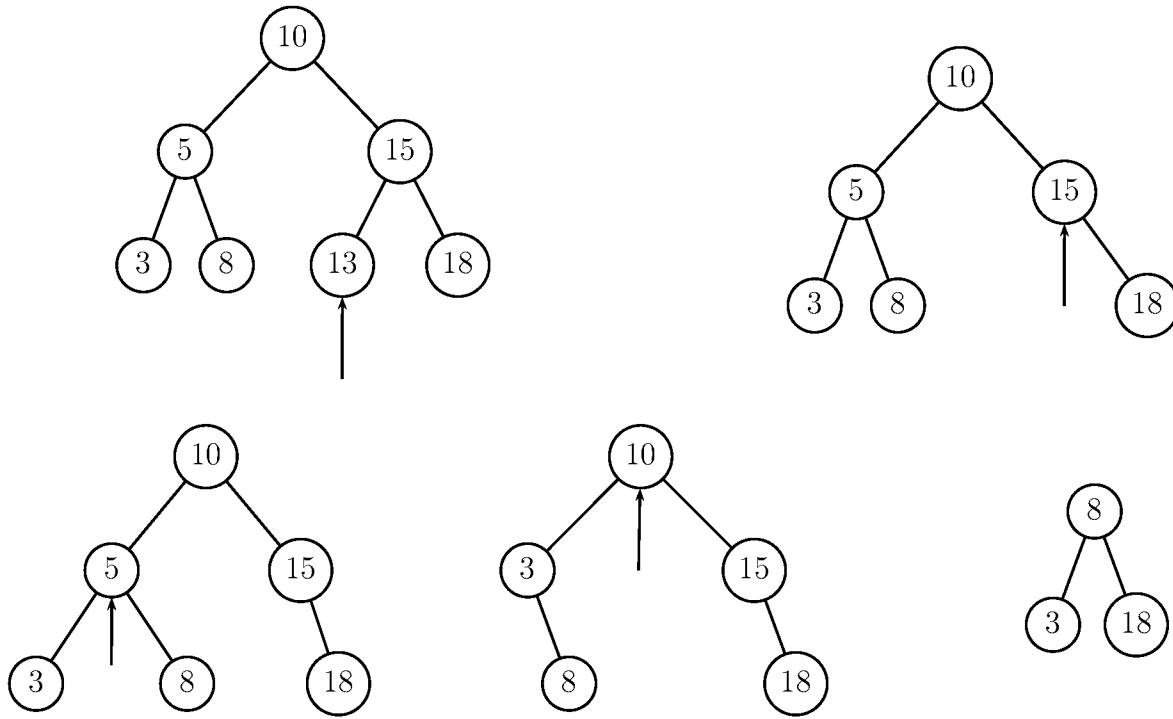
```

if x < p^.key then
    delete(x, p^.l)
else if x > p^.key then
    delete(x, p^.r)
else begin { Исключение p^ }
    q := p;
    if q^.r = nil then { Правого поддерева нет, вместо удаленного узла ставим
        корень левого }
    p := q^.l
    else if q^.l = nil then { Левого поддерева нет, вместо удаленного узла
        ставим корень правого }
    p := q^.r
    else { Удаляемая вершина имеет двух потомков. Поиск самого правого
        элемента левого поддерева – кандидата на замещение удаляемой
        вершины }
    del(q^.l);
    dispose(q) { Удаление самого элемента или кандидата на его замещение,
        найденного среди потомков }
    end;
end; { delete }

```

Вспомогательная рекурсивная процедура *del* осуществляет рекурсивный спуск вдоль правой ветви левого поддерева элемента  $q \uparrow$ , исключаемого из дерева, и заменяет его ключ и счётчик на соответствующие значения из самой правой компоненты  $t \uparrow$  левого поддерева, после чего динамический объект  $q \uparrow$  можно уничтожить деструктором **dispose** (можно было бы с тем же успехом делать рекурсивный спуск по самой левой ветви правого поддерева удаляемого элемента).

Проиллюстрируем процесс удаления нескольких элементов из дерева поиска. При этом возникают все рассмотренные случаи.



Проанализируем сложностные оценки алгоритма поиска по дереву с включениями. Во-первых ясно, что получаемое дерево не будет идеально сбалансированным и оценка для числа сравнений  $\log_2 n$  может ухудшиться [54]. Худший случай соответствует вырожденному несбалансированному дереву, сформированному строго возрастающей (или строго убывающей) последовательностью элементов, когда каждый ключ присоединяется непосредственно справа (или слева) от его предшественника, образуя линейный список с известным нам временем поиска  $O(N)$ . Но этот случай маловероятен: оценка в книге Н. Вирта [54] показывает, что среднее время поиска отличается от оптимального на 39%. Таков средний выигрыш от всевозможных усовершенствований структуры дерева по мере накопления статистики запросов на поиск. Они оправданы, когда число вершин в дереве достаточно велико или когда доступ к дереву превалирует над включением элементов.

### 5.12.3 Сбалансированные деревья

Поскольку каждое включение элемента приводит к разбалансировке дерева поиска, а значит, и к ухудшению времени доступа, то возникает мысль об оперативной балансировке дерева после операций вставки. Для упрощения операции восстановления дерева после случайного включения отечественными учёными Г. М. Адельсоном-Вельским и Е. М. Ландисом было предложено ослабить строгую сбалансированность дерева. Сбалансированным деревом ими было названо такое дерево, высоты поддеревьев каждой из вершин которого отличаются не более, чем на единицу. В честь авторов такие деревья названы *AVL-деревьями*. Идеально сбалансированные деревья также являются AVL-деревьями. Это определение приводит к простой процедуре ребалансировки дерева, причём средняя длина пути поиска практически совпадает с его длиной в идеально сбалансированном дереве.

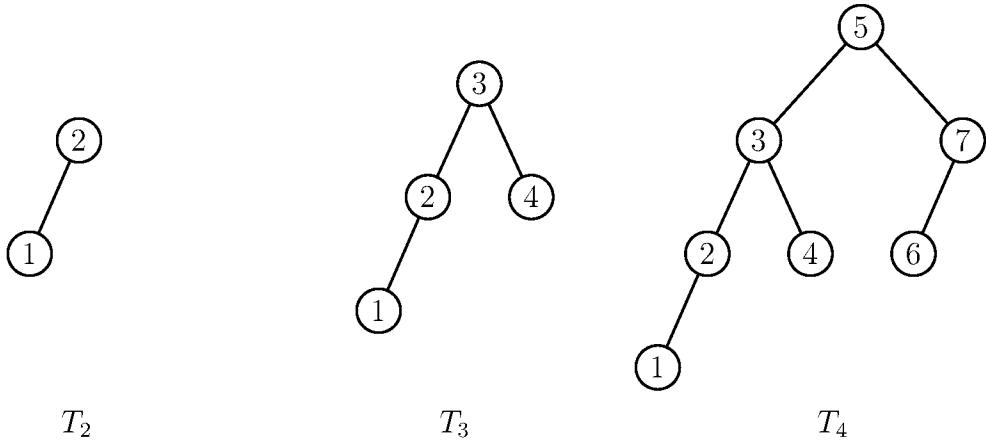
В сбалансированных деревьях за время, пропорциональное  $O(\log_2 n)$ , даже в худшем случае, можно выполнить следующие операции:

- найти вершину с данным ключом;
- включить новую вершину с заданным ключом;
- исключить вершину с указанным ключом.

По теореме Адельсона-Вельского–Ландиса сбалансированное дерево никогда не будет по высоте превышать идеально сбалансированное более чем на 45%, независимо от количества вершин. Высота сбалансированного дерева с  $n$  вершинами  $h_b(n)$  находится в пределах

$$\log_2(n+1) \leq h_b(n) < 1.4404 \cdot \log_2(n+2) - 0.328$$

Минимум достигается, если дерево идеально сбалансировано, т. е. при  $n = 2^k - 1$ . Интересно узнать и о максимуме для  $h_b(n)$ , т. е. о самом плохо сбалансированном дереве максимальной высоты для  $n$  вершин. Для построения такого дерева может быть применена следующая идея: будем строить сбалансированные деревья с минимальным числом вершин для фиксированной высоты  $h$ . Дерево с минимальным числом вершин  $T_h$  высоты  $h$  построим индукцией по высоте.  $T_0$  — пустое дерево, а  $T_1$  — дерево с одной единственной вершиной (база индукции). Для построения  $T_h$  для  $h > 1$  мы будем брать корень и два поддерева опять же с минимальным числом вершин. Следовательно, эти деревья также относятся к классу  $T$ -деревьев. Одно поддерево, очевидно, должно быть высотой  $h - 1$ , а другому позволяет иметь высоту на единицу меньшее, т. е.  $h - 2$ . Ниже представлены деревья высотой 2, 3 и 4.



Поскольку принцип их построения очень напоминает определение чисел Фибоначчи, то мы будем называть такие деревья *деревьями Фибоначчи*. Их определение более, чем стандартно:

1. Пустое дерево есть дерево Фибоначчи высоты 0.
2. Единственная вершина есть дерево Фибоначчи высоты 1.
3. Если  $T_{h-1}$  и  $T_{h-2}$  — деревья Фибоначчи высотой  $h-1$  и  $h-2$ , то  $T_h = \langle T_{h-1}, x, T_{h-2} \rangle$  также дерево Фибоначчи высотой  $h$ .

4. Никакие другие деревья деревьями Фибоначчи не являются.

Число вершин в  $T_h$  определяется из такого простого рекуррентного отношения:

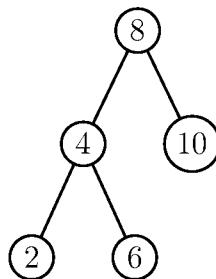
$$N_0 = 0, N_1 = 1 \\ N_h = N_{h-1} + 1 + N_{h-2}$$

Число  $N_i$  как раз и будет числом вершин для самых худших случаев (верхний предел  $h$ ).

### 5.12.3.1 Включение в сбалансированное дерево

Проанализируем операцию включения в сбалансированное дерево новой вершины. Мы надеемся, что цена этой операции не ухудшит замечательные свойства AVL-деревьев. Если наше дерево состоит из корня  $t$  и левого и правого поддеревьев  $L$  и  $R$  соответственно, то либо оно идеально сбалансировано и вставка новой вершины углубит одно из поддеревьев на единицу. Либо вставляемая вершина попадет в более низкое поддерево и улучшит сбалансированность. Наконец, при попадании нового узла в более высокое поддерево, AVL-сбалансированность дерева нарушится и возникнет необходимость балансировки дерева.

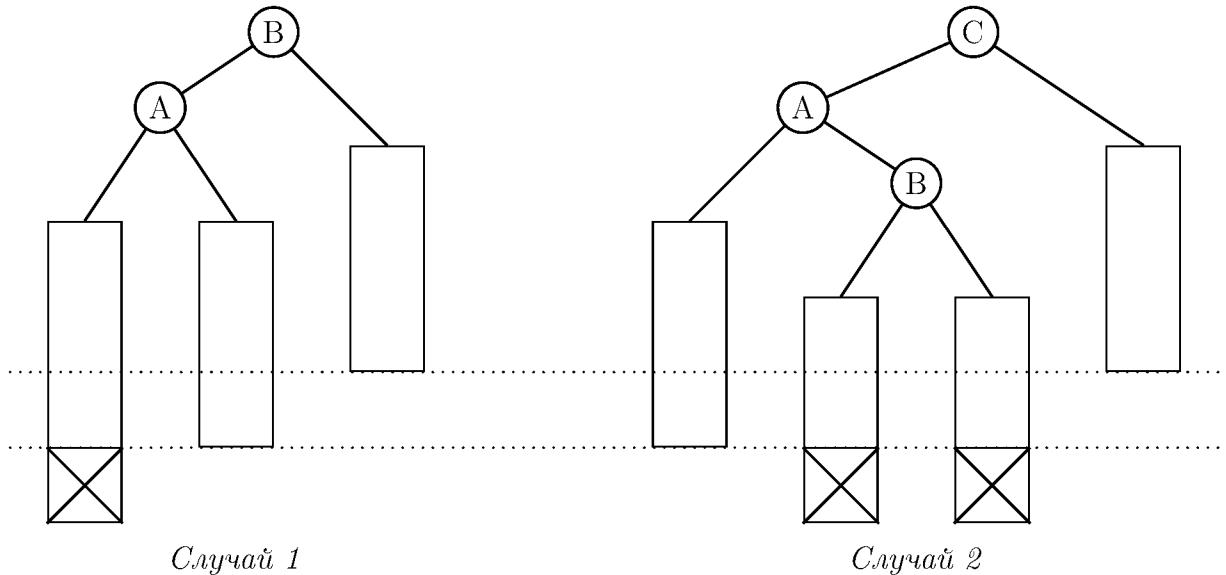
Рассмотрим дерево



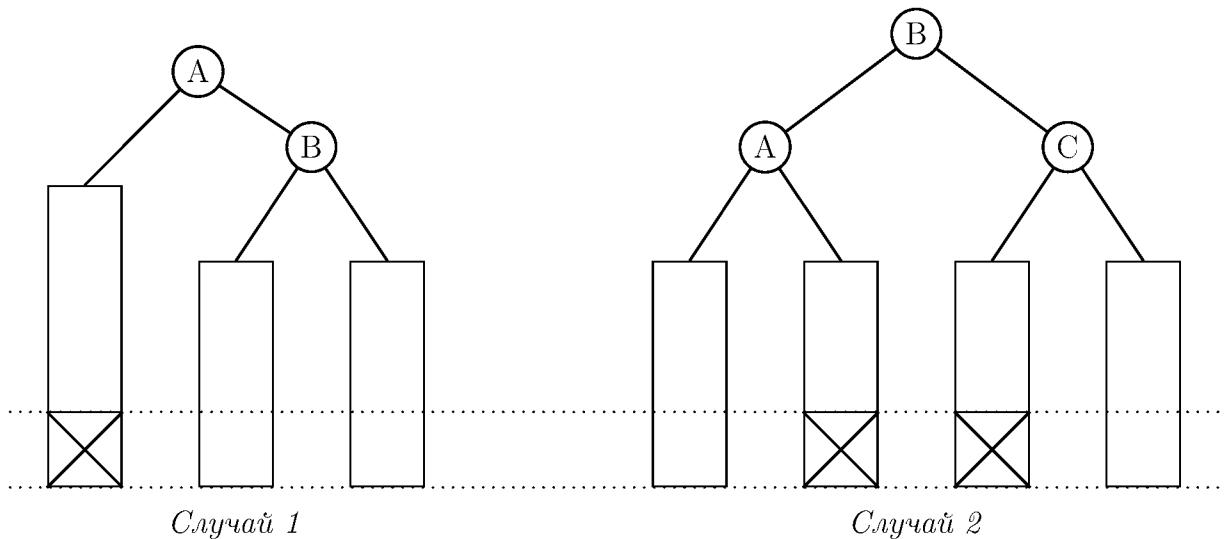
Вершины с ключами 9 и 11 можно вставить, не нарушая сбалансированности дерева. Если бы в дерево поиска сначала пришло бы число 10, то дерево стало бы односторонним. Дерево с корнем 8 будет лишь лучше сбалансированным. Включение вершин 1, 3, 5, 7 разбалансирует дерево.

С точностью до симметрии существует всего лишь две различные возможности несбалансированности, возникшей из-за включения. Они отличаются местом возникновения перевеса: с края или в середине дерева.

Первый случай — вставка вершины 1 или 3, второй — 5 или 7:



Процесс балансировки заключается в вертикальном «подтягивании» одного или двух поддеревьев путем манипуляций ссылками на потомков в вышестоящих вершинах, релевантных для восстановления баланса. Среди этих манипуляций такие: подчинение корня левому поддереву и переподчинение внука, выравнивающее баланс, и подъем на два уровня внутреннего под дерева, включение в которое привело к несбалансированности. При этом относительное горизонтальное расположение переставляемых вершин остается без изменений. Мы просто передергиваем ветки поддеревьев как нитки к куклам-марионеткам, «не перекрещивая пальцы».



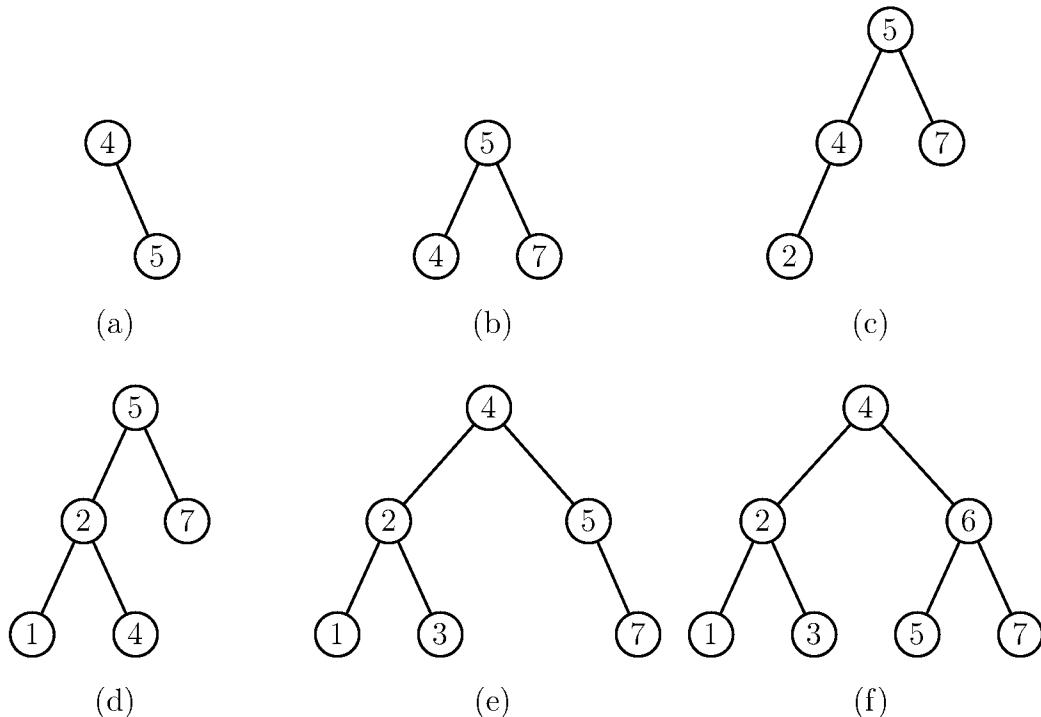
Алгоритм включения и балансировки существенно использует информацию о сбалансированности дерева. Проверять балансировку по дереву всякий раз при включении в него слишком дорого:  $O(N)$ , где  $N$  — количество вершин дерева. На другом полюсе — хранение в каждой вершине показателя сбалансированности — разности высоты правого и левого поддерева, принимающего значения в диапазоне  $[-1; +1]$  — в эти же два бита можно упаковать и три перечислимых балансировочных значения.

Схема алгоритма включения в сбалансированное дерево такова:

1. поиск элемента в дереве (неудачный!);
2. включение новой вершины и определение результирующего показателя сбалансированности;
3. отход по пути поиска с проверкой показателя сбалансированности для каждой проходимой вершины, балансируя в необходимых случаях соответствующие поддеревья.

Программа включения в сбалансированное дерево приведена в [54].

Проиллюстрируем принцип работы алгоритма:



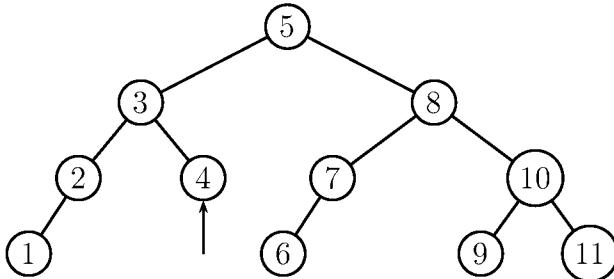
Когда в дереве две вершины (а), включение вершины 7 вначале приводит к дисбалансу — правостороннему линейному списку. Действуя вышерассмотренным случаем 1 (несбалансированность с края), восстановим баланс (б). Последующие включения вершин 2 и 1 снова приводят к случаю 1 — несбалансированному справа поддереву с корнем 4 (с). Балансируем, получаем (д). Последующая вставка элемента с ключом 3 нарушает баланс в корневой вершине со значением 5. Балансировка должна проходить по второму сценарию (случай 2). Результат — дерево (е). При любом следующем включении баланс может быть нарушен лишь в вершине 5. Добавление вершины 6 опять приводит к случаю 2 и к дереву (ф).

Авторами идеи была получена оценка ожидаемой высоты сбалансированного дерева  $h = \log_2 n + c$ , где константа  $c \approx 0.25$ , а значит и времени поиска. AVL-сбалансированные деревья ведут себя также, как сбалансированные, однако затраты на их поддержку намного меньше. Экспериментальные данные подтверждают, что на два включения приходится одна балансировка.

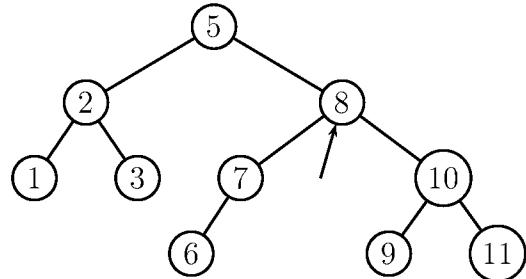
Сложность балансировки оправдана тогда, когда поиск данных происходит значительно чаще, чем вставка.

### 5.12.3.2 Исключение из сбалансированного дерева

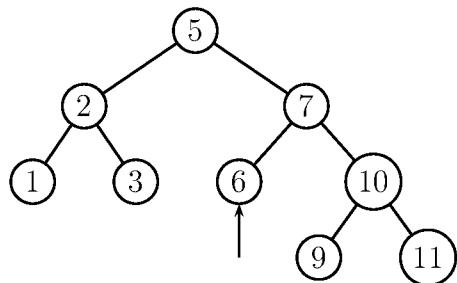
Вспоминая, что исключение из дерева сложнее включения, следовало бы того же ожидать и от AVL-дерева. Процедура следует схеме включения элемента. Удаление терминальной или однодетской вершины — задача в одно действие. Двудетные вершины обрабатываются так же, как и раньше: они заменяются на самую правую вершину ее левого поддерева. Мы ограничимся лишь иллюстрацией процесса удаления.



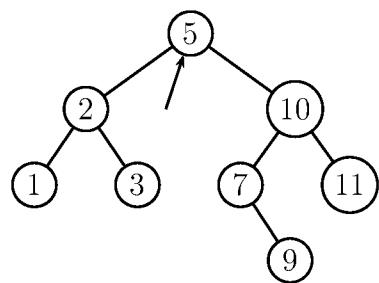
(a)



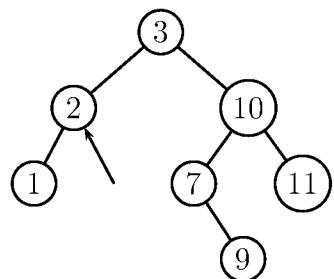
(b)



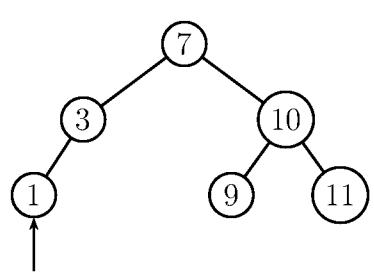
(c)



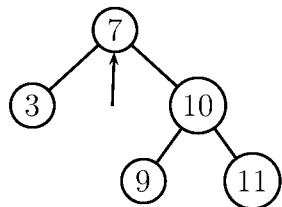
(d)



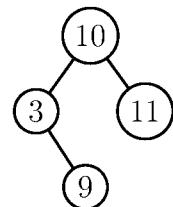
(e)



(f)



(g)



(h)

Исключение любого элемента из сбалансированного дерева оценивается  $O(\log N)$ .

### 5.12.3.3 Деревья оптимального поиска

Заканчивая рассмотрение деревьев поиска, нельзя не упомянуть еще об одном способе их усовершенствования. Если *a priori* или из статистики известны вероятности появления аргументов поиска, то можно сообразно им реорганизовывать деревья поиска, поднимая к корню более вероятные ключи. Цена вопроса  $O(N^2)$ . Ху и Таккером, Уолкером и Готлибом [54, 63] были получены более качественные алгоритмы: затраты памяти  $O(N)$  вместо  $O(N^2)$  и затраты на оптимизацию структуры всего лишь  $O(N \log N)$ .

### 5.12.4 Физическое представление. Отображение на массив.

Сплошное представление деревьев удобно пояснить на примере турнирного дерева с фиксированной структурой. Считая, что участники турнира образуют полные пары нужной кратности, введём жёсткое размещение элементов дерева в массиве. В первом элементе массива разместим корень дерева, во 2-ом и 3-ем — его левого и правого потомков. Далее поместим пары потомков потомков и т. д. Сыновья элемента дерева с индексом  $i$  хранятся в элементах массива с индексами  $2i$  и  $2i + 1$ . Согласно данной схеме размещения,  $j$ -ый элемент  $i$ -ого уровня имеет индекс  $2^{i-1} + j - 1$ . Этот метод весьма экономичен по памяти, пропадают только слова, отведённые отсутствующим вершинам дерева; надо только суметь пометить их как неиспользуемые. Ввиду систематической и легко вычислимой структуры дерева, память на связывание элементов дерева не расходуется. Сложностные оценки для операций поиска, вставки и удаления элемента в турнирное дерево сплошного представления пропорциональны  $O(N)$ .

Заметим, что сплошное турнирное представление дерева даёт нам ещё один обход — поуровневый, но он не имеет замечательных свойств, которыми обладают основные методы.

Основным неудобством сплошного представления дерева является высокая цена вставки и удаления элементов. Не мал и перерасход памяти на пустые элементы [44]. Как всегда, за ликвидацией этих недостатков мы обратимся к динамическим структурам. Для деревьев можно использовать рекурсивные ссылочные представления, которые были разработаны для списков, но с той лишь разницей, что указатели вперёд и назад по линейной структуре теперь направляются к левому и к правому поддеревьям соответственно. Тип для вершины дерева, конечно же, фиксирован, но имеет двойное рекурсивное разветвление [72]:

```
type p = ^node;
type node = record
    op : char;
    l, r : p
end;
```

Цепное представление рассмотренного нами дерева выражения было рассмотрено выше.

#### 5.12.4.1 Прошивка деревьев

В представлении бинарного дерева содержатся два указателя — на левое и правое поддеревья (обозначим их  $l$  и  $r$ , соответственно). У листьев дерева оба эти указателя пустые. Поскольку в бинарном дереве, как правило, около половины узлов являются листьями (если не рассматривать вырожденные случаи), то такое представление оказывается неэкономичным с точки зрения расхода памяти [44].

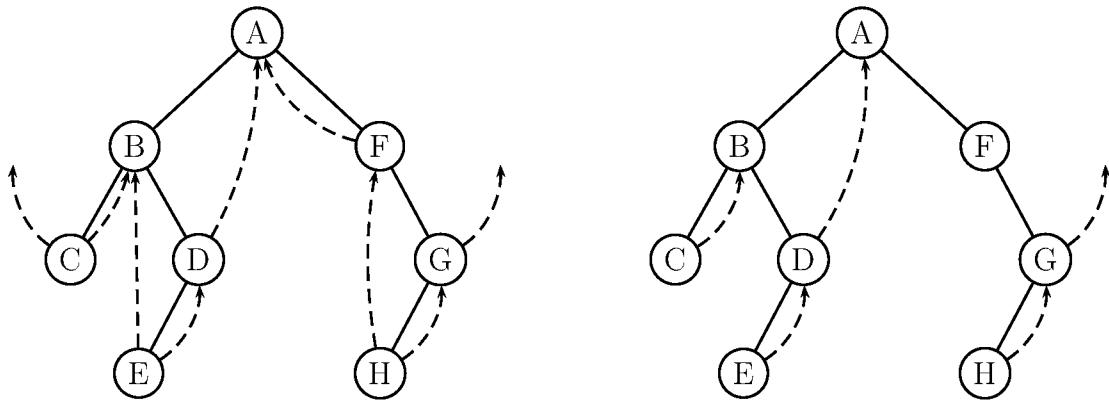
В прошитых бинарных деревьях вместо пустых указателей используются специальные связи-нити и каждый указатель в узле дерева дополняется однобитовым признаком *ltag* и *rtag*, соответственно [63]. Признак определяет, содержит ли в соответствующем указателе обычная ссылка на поддерево или в нем содержится связь-нить.

Связь-нить в поле  $l$  указывает на узел — предшественник в *обратном* порядке обхода (*inorder*), а связь-нить в поле  $r$  указывает на узел — преемник данного узла в обратном порядке обхода.

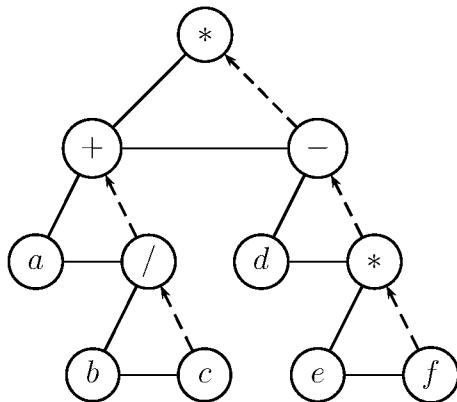
Введение признаков *ltag* и *rtag* не приводит к сколько-нибудь значительному увеличению затрат памяти, зато упрощает алгоритм обхода деревьев, так как для прошитых деревьев можно выполнить *нерекурсивный* обход *без использования стека*. Заметим, что у самого левого при концевом обходе узла левая связь-нить пустая; аналогично пуста правая связь-нить у самого правого при концевом обходе узла. Таким образом, прошивка деревьев — это ещё один (нестековый!) метод борьбы с рекурсией, оставшийся с тех времён, когда она считалась чересчур расточительной. Фактически, прошитое дерево линеаризовано и может быть помещено в очередь или список. Хотя древовидная структура прошитого дерева сохраняется, процедуры его рекурсивного или стекового обхода должны быть модифицированы, поскольку обычный обход существенно опирается на *nil*'ы, терминирующие ветви дерева.

В настоящее время эта концепция, оставшаяся с тех времен, может быть полезна для итеративного обхода дерева, например, хранящего элементы множества, с целью совершения над всеми элементами множества некоторой операции. Техника итераторов в этом случае очень удобна, в частности, можно использовать те же алгоритмы, что и для обработки списка, поскольку функциональность итераторов совпадает.

Среди прошитых деревьев важный класс составляют правопрошитые деревья, т. е. прошитые бинарные деревья, у которых используется только правая связь-нить, а в поле  $l$  содержится либо обычный указатель, либо пустой указатель. Поле *ltag* в таком случае не используется. На рисунке, приведенном ниже, обычные ссылки нарисованы сплошной линией, связи-нити — штриховой линией с соответствующими сторонами узлов. Пустые связи-нити для самого левого и самого правого узлов изображены штриховыми линиями, не ведущими ни к какому из узлов дерева. Они символизируют начало и конец обратного обхода линеаризованного прошивкой дерева.



Такой способ представления дерева удобен для арифметических выражений. Однако в силу того, что оба операнда арифметических операций «равноправны» и подчинены не один другому, а операции, бинарное дерево для выражений не разворачивается на  $45^\circ$ , поэтому все братья изображаются на одном уровне. Правопрошитое дерево для выражения показано ниже. Отметим, что связь-нить от младшего из братьев ведет к отцу. Правопрошитое дерево такого вида легко позволяет представлять и  $n$ -арные операции в обычных деревьях. Еще раз подчеркнем, что на рисунке для всех узлов корни их правых поддеревьев изображены справа на одном уровне с узлами.



Прошивку деревьев придумали Перлис и Торnton в 1960 г. Читателям, интересующимся алгоритмами обхода деревьев, рекомендуем монографию [63].

## 5.13 Графы

Графом называют некоторое подмножество декартова произведения двух множеств. Часто берутся не разные множества, а одно и то же множество, т. е. граф рассматривается как подмножество декартова произведения множества на себя. В математическом смысле граф означает то же, что и отношение. На бумаге (топологически) граф изображается как множество точек (называемых *вершинами*), соединенных линиями (называемыми *ребрами*). Каждая пара вершин соединяется не более чем одним ребром.

Две вершины называются смежными, если в графе есть ребро, соединяющее эти вершины.

Говорят, что в графе существует путь из вершины  $A$  в вершину  $B$ , если найдется такая последовательность вершин  $A_1, A_2, \dots, A_n$  такая, что  $A_1 = A$ ,  $A_n = B$  и  $\forall i = 1, \dots, n - 1$  вершины  $A_i, A_{i+1}$  являются смежными (т. е.  $A_i$  и  $A_{i+1}$  соединены ребром).

Путь называется *простым*, если на нем все промежуточные вершины  $A_2, \dots, A_{n-1}$  попарно различны.

Граф называется *связным*, если имеется путь между любыми двумя вершинами графа.

*Циклом* называется простой путь длины не менее 3 от какой-либо вершины до нее самой (т. е. путь, более сложный, чем движение по некоторой дуге и возврат обратно по этой же дуге).

Ориентированным графом называется граф, каждому ребру которого приписано определенное направление (ребро в ориентированном графе называется дугой).

Говорят, что в ориентированном графе существует путь из вершины  $A$  в вершину  $B$ , если найдется такая последовательность вершин  $A_1, A_2, \dots, A_n$ , причем  $A_1 = A, A_n = B$  и  $\forall i = 1, \dots, n - 1$  существует дуга из вершины  $A_i$  в вершину  $A_{i+1}$  (т. е. путь по дугам с учетом их направления).

Подробно теория графов рассматривается в курсе дискретной математики. Здесь же отметим, что в программировании задачи с использованием теории графов возникают достаточно часто, например, в задачах проектирования электрических схем, в системах искусственного интеллекта.

Одна из основных проблем, возникающих при использовании графов, связана с поиском путей между вершинами. Рассмотрим способы представления графов в памяти ЭВМ, обеспечивающие достаточно удобные возможности решения этой проблемы.

Граф представляется матрицей смежности. Пусть граф содержит  $k$  вершин. Тогда матрицей смежности называется квадратная булевская матрица  $M$  размерности  $k \times k$ , в которой значения элементов определены так:  $M_{ij} = \text{false}$ , если нет ребра, соединяющего  $i$ -тую вершину с  $j$ -той, и  $M_{ij} = \text{true}$  в противном случае. Для неориентированных графов матрица смежности симметрична.

Для ориентированного графа элемент  $M_{ij}$  матрицы смежности равен **true** тогда и только тогда, когда граф содержит дугу, начинающуюся в  $i$ -той вершине и оканчивающуюся в  $j$ -той вершине.

Для нахождения путей в графе используют операции над матрицами смежности. Произведение и сумма булевых матриц определяются так же, как и для обычных матриц, только вместо умножения и сложения чисел используются булевые операции  $\&$  («и») и  $\vee$  («или»), соответственно. По аналогии с обычными числовыми матрицами определяется степень булевой матрицы.

Справедливо утверждение, что в графе с  $k$  вершинами и матрицей смежности  $M$  существует путь из вершины номер  $i$  к вершине номер  $j$  тогда и только тогда, когда в матрице  $M^+$  элемент  $M_{ij}^+$  истинен. Матрица определяется как

$$M^+ = \bigvee_{i=1}^k M^i$$

Признаком цикла для вершины номер  $i$  в ориентированном графе служит истинное значение элемента  $M^+(i, i)$ .

Для получения значения матрицы  $M^+$  необходимо вычислить булевский матрочлен за  $O(N^4)$  шагов. Эффективный алгоритм вычисления матрицы  $M^+$  (алгоритм Уоршалла) описан в книгах [70, 77, 36]. Его сложностная оценка есть  $O(N^3)$ .

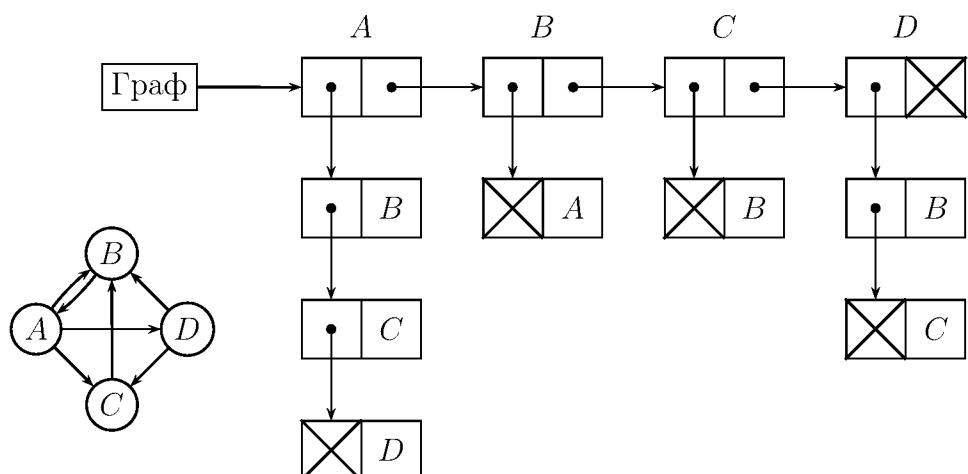
Граф с  $k$  вершинами можно представить с помощью  $k$  списков, каждый из которых соответствует вершине графа и содержит номера ее смежных вершин. Указатели этих списков размещаются в одном массиве.

Для представления графа можно воспользоваться одномерным массивом целых чисел. Обозначим этот массив идентификатором  $A$ . Первые  $k$  элементов массива  $A$  содержат индексы элементов массива, с которых начинаются последовательности номеров вершин, смежных с данной, а начиная с  $k + 1$  элемента массива  $A$  подряд размещаются сами последовательности номеров вершин. В начале каждой последовательности указывается число смежных вершин, а потом перечисляются их номера. Так для  $i$ -той вершины  $A[i]$  — индекс описания вершины номер  $i$  в массиве  $A$ . Обозначим  $P = A[i]$ . Тогда  $A(P)$  — число вершин, смежных с  $i$ -той, а элементы  $A(P + 1), \dots, A(P + A(P))$  задают их номера.

Граф с  $k$  вершинами можно представить целочисленным массивом из  $k$  столбцов и  $m$  строк, где  $m$  — максимальное число вершин, смежных с произвольной вершиной графа. Элементы столбца матрицы содержат номера смежных вершин. Если у вершины меньше, чем  $m$  ребер, то последние элементы столбца заполняются нулями.

Этот способ удобен в случаях, когда в графе у большинства вершин одинаковое или почти одинаковое число смежных вершин, равное или близкое к  $m$ , и число  $m$  существенно меньше  $k$ .

Если график меняется в процессе обработки, т. е. добавляются и удаляются вершины и дуги, то удобно использовать списки. Граф описывается с помощью основного списка вершин и списков дуг. В основной список включены узлы для каждой вершины графа. С каждым узлом основного списка связан свой список дуг. Каждый узел списка дуг содержит указатель на узел основного списка, в который входит соответствующая дуга. На рисунке показан график и его представление с помощью списков. Для того, чтобы избавиться от многочисленных пересечений стрелок, на этом рисунке в представлении узлов для дуг вместо стрелок, ведущих к вершинам, поставлены имена вершин.



Слева график, справа — его представление: над узлами основного списка проставлены названия вершин, а в узлах списков дуг вместо стрелок записаны имена вершин [44].

## 5.13.1 Алгоритмы на графах

### 5.13.1.1 Поиск в глубину

Реализация поиска в графе в глубину отличается от поиска в дереве тем, что при всякой попытке перейти в новую вершину осуществляется проверка, была ли посещена уже эта вершина. Это связано с тем, что в графе, в отличие от дерева, могут быть циклы и петли, в силу чего «наивный» алгоритм просто зациклится.

```
program DepthSearchDemo(input, output);

{ Граф представляется в виде матрицы смежности }
type Graph = array [1..5, 1..5] of boolean;

{ Путь в графе – последовательность (список) вершин }
type Path = List { of integer };

procedure Create(var l : Path);
function First(var l : Path) : Iterator ;
function Last(var l : Path) : Iterator ;
function Insert(var l : Path; var i : Iterator ; i : integer) : Iterator ;
function Delete(var l : Path, var i : Iterator ) : Iterator ;
procedure Destroy(var l : Path);

function NotEqual(var lhs, rhs : Iterator) : boolean;
{ Описание оператора слегка изменено для удобства: функция, а не процедура }
function Next(var i : Iterator) : Iterator; { Аналогично operator++ в STL для std::
list<T>::iterator }

function Prev(var i : Iterator) : Iterator ;
function Fetch(var i : Iterator) : integer;

{ Объявление find заимствовано из STL: ISO/IEC 14882:1998(E), p. 546 }
function find(first , last : Iterator ; var x : integer) : Iterator ;
var found : boolean;
begin
  found := false;
  while not found and NotEqual(first, last) do begin
    if Fetch( first ) = x then
      found := true;
    end;
    find := first ;
  end;

{ Печать списка с последнего элемента до первого }
procedure print(first, last : Iterator );
begin
  while NotEqual(first, last) do begin
    Prev(last);
    
```

```

    write(Fetch(last), ' ')
end;
writeln;
end;

{ DepthSearch ищет все пути в графе между init и goal и выводит их на печать }
procedure DepthSearch(init, goal : integer; var G : Graph);
var P : Path;

procedure search;
var i, k : integer;
begin
    k := Fetch(First(L));
    if k = goal then
        print(First(P), Last(P))
    else
        for i := 1 to 5 do
            if G[k, i] and NotEqual(find(First(P), Last(P), i), Last(P)) then begin
                insert(P, First(P), i); { Вершины пути хранятся в обратном порядке
                                         для удобства программирования }
                search;
                delete(P, First(P));
            end;
    end;
end;

begin
Create(P);
insert(P, first(P), init );
search;
Destroy(P);
end;

var G : Graph;
    i, j : integer;
begin { Создание графа без дуг }
    for i := 1 to 5 do
        for j := 1 to 5 do
            G[i, j] := false;
    { Добавим несколько дуг }
    G[1, 2] := true; G[1, 3] := true;
    G[2, 4] := true;
    G[3, 2] := true; G[3, 5] := true;
    G[4, 3] := true; G[4, 5] := true;
    DepthSearch(1, 5, G);
end.

```

### 5.13.1.2 Поиск в ширину

```
#include <iostream>
#include <list>
#include <queue>
#include <algorithm>

const int MAX = 5;

std::ostream& operator<<(std::ostream& s, std::list<int> p)
{
    for(std::list<int>::iterator i = p.begin(); i != p.end(); i++)
        s << *i << " ";
    return (s << std::endl);
}

// Источники: [57, 58]
void breadth_search(bool graph[MAX][MAX], int init, int goal)
{
    std::list<int> p(1, init);
    std::queue<std::list<int>> q;
    q.push(p);
    while(!q.empty())
    {
        p = q.front();
        q.pop();
        int v = p.back();
        if(v == goal)
            std::cout << p;
        else
            for(int i = 0; i < MAX; i++)
                if(graph[v][i] && (std::find(p.begin(), p.end(), i) == p.end()))
                {
                    p.push_back(i);
                    q.push(p);
                    p.pop_back();
                }
    }
}

int main(void)
{
    bool g[MAX][MAX] = {{false, true, true, false, false},
                        {false, false, false, true, false},
                        {false, true, false, false, true},
                        {false, false, true, false, true},
```

```
{false, false, false, false, false});  
breadth_search(g, 0, 4);  
return 0;  
}
```

# Глава 6

## Сортировка и поиск

### 6.1 Алгоритмы поиска

Одно из наиболее часто встречающихся в программировании действий — поиск [65]. Эффективность поиска часто является основным критерием качества различных структур данных [54]. К классическим задачам поиска слова в тексте или отыскания данных в базах данных в последние годы добавились такие актуальные задачи, как поиск в интернете, антивирусное сканирование программного кода и антиспамовая фильтрация почтовых сообщений. Быстрый многокритериальный семантический поиск необходим для очистки и анализа данных (data mining) и в оперативно-розыскной деятельности спецслужб. Уроки деревьев поиска наводят на мысль, что прямые методы поиска могут быть чудовищно неэффективными.

Для рассмотрения алгоритмов поиска предположим, что множество данных, в котором выполняется поиск, фиксировано, резидентно и допускает прямой доступ к каждому элементу:

```
var a : array[0..N - 1] of item;
```

где *item*, как всегда, комбинированная структура с ключевым полем *key*. В результате поиска значения *x* в массиве *a* устанавливается индекс *i*, удовлетворяющий условию *a[i].key = x*. Для простоты далее будем отождествлять ключ и искомый элемент.

#### 6.1.1 Линейный поиск

Линейный поиск заключается в последовательном просмотре массива как списка с той же линейной сложностью:  $O(N)$ . Просмотр выполняется в цикле, условие завершения которого таково:

- элемент найден:  $\exists i : (0 \leq i < N) \& (a_i = x)$ ;
- обход массива завершен, но искомый элемент не найден.

```
i := 0; { Выполнять поиск, пока индекс массива в пределах диапазона и пока
          искомое значение x не совпадет с очередным элементом a[i] }
while (i < N) and (a[i] <> x) do
    i := i + 1;
```

Стоимость выполнения такого цикла — 4 у. е. за итерацию: 3 операции в предусловии и 1 операция в теле цикла.

Ввиду возможности досрочного окончания цикла мы не используем в этой программе цикл с параметром **for**.

Проанализируем условие продолжения цикла. Инвариант цикла, т. е. условие, выполняющееся перед каждым увеличением индекса  $i$  выглядит так:

$$(0 \leq i < N) \& (\forall k : 0 \leq k < i : a_k \neq x)$$

Вторая часть инварианта постулирует, что ранее, для всех  $k < i$ , совпадения также не было. Нарушая этот инвариант всеми способами по законам де Моргана и др., получаем условие прекращения цикла:

$$((i = N) \vee (a_i = x)) \& (\forall k : 0 \leq k < i : a_k \neq x)$$

Это условие не только специфицирует результат поиска, но и определяет именно *первое* вхождение искомого элемента с минимально возможным индексом. Поскольку элементы нумеруются нами с 0, равенство  $i = N$  свидетельствует, что при обходе массива совпадения не было. Легко доказать и завершность цикла: на каждом шаге значение  $i$  увеличивается на конечную величину 1, и, следовательно, за конечное число шагов достигнет предельного значения  $N$ .

Ускорить поиск поможет барьерный элемент, который надо поместить в  $N + 1$ -ый элемент массива, имеющий индекс  $N$ :

```
var a: array [0..N] of integer;
```

и из предусловия цикла будут удалены две из трёх проверок:

```
a[N] := x;  
i := 0;  
while a[i] <> x do  
    i := i + 1;
```

Постусловие цикла также упростится:

$$(a_i = x) \& (\forall k : 0 \leq k < i : a_k \neq x)$$

Здесь мы, как и в случае со списком, перешли от цикла с параметром — дискретным временем к циклу по сплошному участку массива. Издержки поиска с барьерным элементом — это дополнительный элемент массива, инструкция присваивания для засылки барьерного элемента и несколько более сложная интерпретация кода.

Осуществляя линейный поиск, мы предполагали равные шансы отыскания элемента на любом из  $N$  возможных мест. Кроме того, очевидно, что этот метод поиска единственным возможным для структур данных со строго последовательным доступом, в частности, основанных на поступательном или вращательном электромеханическом движении магнитного или оптического носителя.

### 6.1.2 Двоичный поиск

Хорошо известно, что поиск данных в упорядоченном множестве с произвольным доступом к элементам более эффективен. В телефонных справочниках, словарях и энциклопедиях данные лексикографически упорядочены. Этот порядок, в свою очередь, базируется на

алфавитном порядке букв<sup>1</sup> и даёт простой и быстрый метод поиска, известный как метод половинного деления или просто, клишированный с английского, двоичный (бинарный) поиск. Последнее название также неудачно, как и двоичное (бинарное) дерево.

Мы помним, что быстрый поиск данных в дереве был предопределён упорядоченной структурой дерева и существенным сокращением перебора элементов. При каждом сравнении отбрасывалась половина из ещё нерассмотренных элементов. В упорядоченных множествах прямого доступа мы можем добиться такой же эффективности.

Рассмотрим массив  $a$ , на котором введено следующее отношение порядка

$$\forall k : 1 \leq k < N : a_{k-1} \leq a_k$$

Для того, чтобы отбросить при поиске половину элементов, и продолжить поиск половинным делением, надо выяснить, в какой половине находится искомый элемент. В силу данного отношения порядка, если выбрать некоторое *промежуточное* значение индекса  $m$ , то отыскиваемый элемент  $x$  либо равен  $a_m$ , либо больше его, либо меньше. В первом случае поиск успешно заканчивается, во втором все элементы с индексами  $i \leq m$  можно исключить из дальнейшего поиска, и L его тем же методом для оставшихся элементов с индексами в диапазоне от  $m + 1$  до  $N$ . Именно в этом диапазоне обязан быть искомый элемент, если он вообще есть. В последнем случае, когда промежуточный элемент  $a_m$  меньше искомого, отбрасывается другая половина рассматриваемого множества. Приведём программу поиска методом половинного деления.

```

var L, R : integer; { Границы отрезка массива a, в котором может находиться
    элемент x и должен осуществляться поиск }
found: Boolean;
begin
    L := 0;
    R := N - 1;
    found := false;
    while ( L <= R ) and not found do begin
        m := { любое промежуточное значение между L и R };
        if a[m] = x then
            found := true
        else if a[m] < x then
            L := m + 1
        else R := m - 1
    end;
end;

```

Инвариант цикла, т. е. условие, выполняющееся перед каждым шагом:

$$(L \leq R) \& (\forall k : 0 \leq k < L : a_k < x) \& (\forall k : R < k < N : a_k > x)$$

Постусловие цикла (при поиске отброшены отрезки, заведомо не содержащие искомого элемента):

$$found \vee ((L > R) \& (\forall k : 0 \leq k < L : a_k < x) \& (\forall k : R < k < N : a_k > x))$$

---

<sup>1</sup> «Благодарим алфавит за любезно предоставленные им упорядоченные буквы» — [www.lib.ru](http://www.lib.ru)

откуда следует, что элемент найден или его нет в просматриваемой последовательности:

$$(a_m = x) \vee (\forall k : 0 \leq k < N : a_k \neq x)$$

Значение  $m$  в диапазоне  $1 \dots N - 2$  может быть любым. Но, вспоминая сбалансированные деревья, потребуем середины, чтобы при всяком сравнении отбрасывать половину бесперспективных элементов. При выборе среднего значения  $m$  максимальное число сравнений равно  $\lceil \log_2 N \rceil + 1$ . То есть, поиск в упорядоченном множестве с прямым доступом к элементам также эффективен, как и в дереве поиска, и намного быстрее линейного поиска с числом сравнений  $N/2$ . Однако вставка и удаление элемента в упорядоченную таблицу дороже, чем в дереве поиска.

Эффективность поиска можно несколько повысить, отложив условный оператор проверки на равенство, вероятность успеха которого равна  $1/N!$  Кроме того, упростим условие окончания цикла. Оказывается, наивное желание закончить поиск при совпадении  $a_m$  с  $x$   $found = true$  в заголовке цикла избыточно и пересекается с условием сходимости границ поиска  $L$  и  $R$ . Минимизированный инвариант цикла

$$(\forall k : 0 \leq k < L : a_k < x) \& (\forall k : R \leq k < N : a_k \geq x)$$

видоизменяет критерий поиска до тех пор, пока интервал  $[L, R]$  не пуст. Поскольку поиск длится не более  $\log_2 N$  шагов, упрощённое предусловие экономит  $2\log_2 N$  операций: отрицания  $found$  и конъюнкции его результата с отношением  $L \leq R$ . (Другой способ несколько ускорить поиск — вместо установки флага завершения цикла  $found$  окончить итерации установкой некорректных границ  $L$  и  $R$  очередного интервала поиска. В результате такого трюка предусловие цикла также упрощается.)

```

L := 0;
R := N; { Барьерный индекс > максимального N - 1. Если R в результате поиска
          остается равным N, то поиск неудачен }
while L < R do begin
    m := (L + R) div 2;
    if a[m] < x then
        L := m + 1
    else
        R := m
end;

```

Несмотря на сокращение проверок, сложность этого поиска все равно логарифмическая. Другой особенностью этого поиска является позднее обнаружение совпадения серединных элементов искомых отрезков ввиду отсутствия их проверки на равенство.

Докажем завершимость цикла с модифицированным предусловием. Отрицая предусловие  $L < R$ , получим условие окончания  $L \geq R$ . Оно обязательно выполнится, если разность  $R - L$  на каждом шаге поиска убывает. Действительно, сначала  $L < R$ . Затем при выборе среднеарифметического  $m$  это соотношение сохраняется: на каждом шаге поиска либо  $L$  увеличивается до  $m + 1$ , либо  $R$  уменьшается до  $m$  и всегда  $L \leq m < R$ . При  $L = R$  повторение цикла заканчивается. Но для завершения поиска необходимо проделать ещё несколько операций. Во-первых, если  $R = N$ , то поиск неудачен, ведь последнее неуспешное сравнение касается интервала  $[R, R]$ , внутри которого вообще нет никаких элементов! В других же случаях, когда правая граница интервала поиска смешалась

влево, следует предусмотреть её проверку на равенство  $a_R = x$ . Эти издержки, конечно же невелики и, как и в случае вставки в дерево поиска, окупаются отысканием элемента с наименьшим среди равнозначных значений индексом: предыдущая версия программы поиска делением пополам вынесенной в начало цикла проверкой жадно хватала первый попавшийся элемент со значением  $x$ !

### 6.1.3 Поиск в таблице

Отличие поиска в таблице от поиска в массиве заключается в том, что в таблицах ключ обычно является составным и имеет регулярную структуру, т. е. сам является массивом, чаще всего словом или строкой. Удобно определить строковый тип в Паскале так:

```
type string = array [0..M - 1] of char;
```

Тогда основополагающее отношение лексикографического порядка на множестве строк задаётся следующим образом

$$(x = y) \Leftrightarrow (\forall j : 0 \leq j < M : x_j = y_j)$$

$$(x \prec y) \Leftrightarrow (\exists i : 0 \leq i < M : ((\forall j : 0 \leq j < i : x_j = y_j) \& (x_i \prec y_i))$$

Проверка совпадения строк осуществляется их полтерным сравнением до первого расхождения. То есть ищется неравенство двух букв с одинаковыми индексами. В случае неуспеха строки равны. Для дальнейших рассуждений необходимо сначала рассмотреть случай коротких строк, для которых приемлем линейный поиск расхождений или имеется аппаратная поддержка.

В большинстве практических приложений удобно иметь дело со строками переменной длины. Каждая такая строка содержит свой размер, обычно не превосходящий максимального  $M$ . Например, на ЭВМ VAX-11 была реализована аппаратная поддержка строк для  $M = 255$ . Фактически это строки не переменной, а регулируемой длины. Это достаточно гибкая и эффективная схема, позволяющая реализовать основные практические потребности, не прибегая к динамическому распределению памяти. Наиболее распространены следующие представления строк.

1. Как уже это делалось при отображении списков, размер строки явно не указывается. Сама строка простирается от начала, заданного адресом или указателем, и до конца, определяемого терминирующим элементом. В качестве ограничителя строк обычно используют непечатный элемент кода с минимальным порядковым номером  $chr(0)$ , предшествующий обычным видимым буквам алфавита. Именно так реализованы строки в Си и в Модуле-2, аналогично их можно запрограммировать в стандартном Паскале.
2. Размер строки агрегируется с её телом и хранится в её первом (нулевом) элементе как сверхкороткое целое. В этом случае легко извлечь длину строки из её структуры и не тратить ресурсы на подсчёт. Недостатками этого способа являются: ограничение длины строки мощностью алфавита, усложнение интерпретации строки и возможные ошибки ввиду полной несовместимости с первым способом.

Далее мы будем отдавать предпочтение первому способу представления строк. Он позволяет организовать их сравнение и копирование более быстрым барьерным методом:

```

i := 0; { Цикл выполняется, пока соответствующие буквы строк совпадают или
пока одна из строк не закончится раньше. Проверка для второй строки –
излишняя, поскольку если вторая строка короче первой, возникнет
несовпадение x[i] <> chr(0) и y[i] = chr(0) }
while (x[i] = y[i]) and (x[i] <> chr(0)) { and (y[i] <> chr(0)) } do
    i := i + 1;

```

В качестве барьера для прекращения цикла выступает терминирующий элемент строки *chr(0)*. Предусловие (инвариант) цикла

$$\forall j : 0 \leq j < i : x_j = y_j \neq \text{chr}(0)$$

отрицая которое получим условие завершения поиска

$$((x_i \neq y_i) \vee (x_i = \text{chr}(0))) \& (\forall j : 0 \leq j < i : x_j = y_j \neq \text{chr}(0))$$

Т. е. при условии, что  $x_i = y_i$  для всех  $i$  строки  $x$  и  $y$  совпадают, а если хотя бы для одного  $i$   $x_i \prec y_i$ , то строка  $x$  лексикографически предшествует строке  $y$ :  $x \prec y$ .

Вернёмся к задаче поиска в таблицах. Особенность поиска по составному ключу в том, что поиск искомого ключа среди других ключей таблицы в свою очередь требует последовательных сравнений компонент ключа. Рассмотрим турнирную таблицу Microsoft Imagine Cup. Здесь ключами являются названия вузов, а данными — названия проектов и города, где назначен следующий тур конкурса. Чтобы узнать, куда едет команда МГТУ, надо отыскать в таблице элемент с соответствующим ключом, побуквенно проверяя компоненты ключа на М, Г, Т и У. На скалярной ЭВМ это потребует цикла с параметром при фиксированной длине ключа или цикла с предусловием при использовании терминирующего элемента строки.

### Microsoft Imagine Cup 2003-2005 CIS

M	A	I		S	a	l	o	n	i	k	i		MAORIE
M	A	I		I	o	k	o	g	a	m	a		FIBRA
M	Г	Т	У	B	a	r	c	e	l	o	n	a	BABYLON
				...								...	
M	Ф	Т	И	C	a	n	-	P	a	у	л	у	Inspiration
M	Ф	Т	И	I	o	k	o	g	a	m	a		omniMusic

Комбинированный тип данных для таблицы может быть таким

```

type T = array [0..N - 1] of string;
var x : string;

```

Допустим, что  $N$  достаточно велико, а таблица  $T$  лексикографически упорядочена. Методом деления пополам, используя ранее составленные наброски программ двоичного поиска и сравнения, получим фрагмент программы поиска

```

L := 0;
R := N;
while L < R do begin
    m := (L + R) div 2;
    i := 0;

```

```

while(T[m, i] = x[i]) and ( x[i] <> chr(0)) do
    i := i + 1;
    if(T[m, i] < x[i]) then
        L := m + 1
    else
        R := m
    end;
{ Проверка ключа правогранничного элемента последнего интервала, не
  учитываемого при модифицированном условии поиска a[R] = x? }
if R < N then begin
    i := 0;
    while(T[R, i] = x[i]) and (x[i] <> chr(0)) do
        i := i + 1
    end;

```

{ условие совпадения: (R < N) and (T[R, i] = x[i]) }

А теперь приведем ту же самую программу на языке Си. Использование указателей позволяет записать фактически *алгоритм* поиска, поскольку эта функция применима к любым массивам и подмассивам. Заметим, что и в библиотеке STL C++ присутствует подобный алгоритм *find()* и предикат двоичного поиска *binary\_search()*.

```

/* Функция ищет вхождение образца what в интервале [l; r), в случае неудачи
   возвращая маркер конца последовательности r */
T* bin_search(T* l, T* r, T what) /* Для типа T должны быть определены
   operator< и operator== */
{
    int* end = r; // Описание терминатора и установка его значения
    for (;;) // Бесконечный цикл, завершается возвратом из функции
    {
        ptr_diff d = r - l; /* Примененный к указателям r и l operator-()
           возвращает разность индексов, как если r и l были бы индексами в массиве
           */
        if(d < 0) // Перехлест границ — неудача при поиске
            return end;
        T* m = l + d / 2; /* Указатель инкрементируется на результат целого
           деления */
        if(*m == what) /* Покомпонентное сравнение выполняет автоматически
           operator==, если он определен. В C++ это верно для std::string */
            return m;
        if(*m < what) /* И здесь мы опираемся на встроенный или переопределенный
           для типа T operator< */
            l = m + 1;
        else
            r = m - 1;
    }
}

```

В стандартной библиотеке Си имеется аналогичная функция *bsearch()*.

Если  $T$  — это **int**, то применение этой функции для поиска в упорядоченном массиве выглядит так:

```
int array[] = {1, 2, 3, 4, 5};  
  
int* last = &array[5]; /* Этого элемента в массиве нет! Но он следует за  
последним и можно взять его адрес для терминирования */  
  
bin_search(&array[0], last, 4); // Вернет указатель со значением &array[3]  
bin_search(&array[1], last, 1); // Вернет указатель со значением last
```

#### 6.1.4 Поиск по образцу

Другой часто встречающейся задачей поиска является поиск вхождения подстроки в некоторой последовательности знаков. Поскольку длина этой последовательности может быть очень большой (поиск в глобальных сетях интернет, в текстовых и двоичных файлах, содержащих программы или данные), то весьма актуально получение сложностных оценок и ускорения такого поиска. Монография Дэна Гасфилда [87] посвящена новому приложению алгоритмов поиска по образцу — расшифровке молекулярных структур для задач биологии и генетики.

Пусть в массиве  $s$  задана последовательность из  $N$  элементов, а в массив  $p$  помещён более короткий образец для поиска длиной  $M$ :  $0 < M \leq N$

```
var s : array [0..N - 1] of item;  
p : array [0..M - 1] of item;
```

Поиск подстроки обнаруживает первое вхождение  $p$  в  $s$ . Рассмотрим прямолинейный метод поиска подстроки. Результатом поиска будем считать индекс  $i$  начала первого вхождения образца  $p$  в последовательность  $s$ . С целью точного формулирования условий поиска введём *предикат частичного совпадения*  $P(i, j)$ :

$$P(i, j) = \forall k : 0 \leq k < j : s_{i+k} = p_k$$

который констатирует, что первые  $j$  букв образца совпадают с  $j$  буквами последовательности, начиная с  $i$ -ой буквы. Результирующий индекс поиска должен удовлетворять этому предикату в точке  $(i, M)$ , т. е.  $P(i, M)$  при успешном окончании поиска становится истинным. Кроме того, раз мы ищем первое вхождение образца,  $P(k, M)$  должно быть ложным и для всех  $k < i$ ! Обозначив соответствующий *предикат первоходжденя* через  $Q(i)$ , выражим это условие через предикат  $P$ :

$$Q(i) = \forall k : 0 \leq k < i : \neg P(k, M)$$

т. е.  $P(k, M)$  для всех предыдущих  $k$  ложно. Алгоритмически  $Q$  может быть определено для каждого  $i$  как

```
Q := true;  
k := 0;  
while Q and (k < i) do begin
```

```

Q := Q and not P(k, M);
k := k + 1;
end;

```

Ввиду зависимости этих предикатов поиск имеет смысл реализовать как повторяющееся сравнение:

```

i := -1;
repeat
    i := i + 1; {вычисление Q(i)}
    found := P(i, M)
until found or (i = (N - M));

```

Вычисление предиката  $P$  вновь производится полтерным сравнением фрагмента последовательности с образцом. Применяя к формуле для  $P$  закон де Моргана, получим:

$$P(i, j) = (\forall k : 0 \leq k < j : s_{i+k} = p_k) = (\#k : 0 \leq k < j : s_{i+k} \neq p_k)$$

т. е. ищется несовпадение с образцом, причём закон де Моргана превращает истинность совпадения для всех литер в ложность несовпадения хотя бы одной. Поиск по образцу также, как и поиск по составному ключу, реализуется вложенными циклами, а предикаты  $P$  и  $Q$  — инварианты этих циклов — включаются в Паскаль-программу, увы, лишь как комментарии:

```

i := -1;
repeat
    i := i + 1;
    j := 0; {Q(i)}
    while (j < M) and (s[i + j] = p[j]) do
        {вычисление P(i, j + 1)}
        j := j + 1
        {Q(i)  $\&$  P(i, j)  $\&$  ((j = M)  $\vee$  (s[i + j]  $\neq$  p[j]))}
    until (j = M) or (i = (N - M));

```

В условии окончания цикла множитель  $j = M$  эквивалентен  $found$ , т. к. из него следует  $P(i, M)$ , а  $i = N - M$  влечёт  $Q(N - M)$  — отсутствие совпадений с образцом во всей последовательности (совпадения не было и уже не будет — слишком мало букв остается в последовательности). Если поиск продолжается при  $j < M$ , то он не должен прекращаться при несовпадении  $s_{i+j} \neq p_j$ . По определению предиката  $P$  отсюда следует ложность  $P(i, j)$ , а по определению  $Q$  — истинность  $Q(i + 1)$ , что и подтверждает истинность  $Q(i)$  после очередного увеличения  $i$ .

Проанализируем алгоритм прямого поиска образца. Его недостатком является то, что всякий раз в случае неуспеха приходится возвращаться назад на всю совпавшую часть образца, чтобы продолжить сравнение с образцом со следующего за началом рассмотренного фрагмента элементом. Эти возвраты, ввиду конструктивных особенностей устройств внешней памяти, на которых нередко размещается сканируемая последовательность, еще больше замедляют прямой поиск.

Этот алгоритм работает достаточно эффективно, если встреча префикса образца в последовательности маловероятна и несовпадение наступает почти сразу после нескольких сравнений букв фрагмента и образца. Сложностная оценка прямого поиска по образцу

находится в диапазоне от  $O(M)$  или  $O(N)$ , когда образец встречается сразу или его первая буква появляется в конце последовательности либо не появляется вообще, и до  $O(N \cdot M)$ , когда и образец, и последовательность содержат общий повторяющийся участок. В последнем случае в цепочке, например, из  $N - 1$  букв  $a$  с одной буквой  $b$  в конце вхождение подобного же образца из  $M - 1$  букв  $a$  также с одной буквой  $b$  на конце потребует максимального количества сравнений. Однако, усовершенствованные методы способны существенно улучшить поиск в таких неблагоприятных случаях.

### 6.1.5 Алгоритм Кнута-Мориса-Пратта

В 1977 году Д. Кнут, Д. Морис и В. Пратт изобрели алгоритм, фактически требующий только  $N$  сравнений даже в самом плохом случае вместо  $N \cdot M$ . Новый алгоритм основывается на том соображении, что, начиная каждый раз сравнение образца с самого начала, мы действуем вслепую и забываем ценную информацию о неудачном поиске. После частичного совпадения начальной части образца с соответствующими литерами последовательности мы фактически знаем удачно пройденную (совпавшую!) часть строки и можем вычислить некоторые сведения на основе самого образца (и только него!), с помощью которых потом быстро продвинемся по тексту, если префиксно-периодическая структура образца делает заведомо неуспешными ближайшие сравнения контекста поиска. Как же можно осветить сканируемую последовательность для более быстрого и дальнего продвижения после неудачи? Для иллюстрации идеи алгоритма приведем пример поиска слова «Hooligan». Знаки, подвергшиеся сравнению, подчеркнуты. При каждом несовпадении двух знаков образец сдвигается по последовательности вперед на все пройденное расстояние, поскольку при анализе образца установлено, что меньшие сдвиги не могут привести к полному совпадению.

Hoola-Hoola girls like Hooligans.

Hooligan

-----  
Hooligan

-  
Hooligan

-  
Hooligan

-----  
Hooligan

-  
...

Hooligan

-----

Псевдокод КМП-алгоритма в терминах предиката частичного совпадения с образцом  $P$  и предиката первого вхождения  $Q$  записывается так:

```
i := 0; { Индекс сканирования последовательности }
j := 0; { Индекс сравнения в образце }
{ Образец не исчерпан, и строка не закончилась }
while(j < M) and (i < N) do begin
```

```

{ Истинна конъюнкция  $Q(i - j) \& P(i - j, j)$  }
{  $i$  заменено на  $i - j$ , т. к. когда была начата попытка сопоставления с
образцом  $j$  букв тому назад, полного совпадения на этом отрезке не было, а
частичное совпадение имеет место, т. е. условие применяется к
подстроке, начинающейся ранее, в силу ее частичного совпадения с образцом
, и первый аргумент предикатов смещается назад к началу участка
совпадения. }

while( $j >= 0$ ) and ( $s[i] <> p[j]$ ) do
    {  $j := D(j)$  прыжок вперед,  $D(j)$  — некая функция скачкообразного смещения
по гарантированно нерелевантному контексту }
     $i := i + 1$ ; { Эволюционное движение по строке с минимальным шагом }
     $j := j + 1$ 
end;

```

Функция сдвига  $D$  будет нами установлена позднее, а сейчас займемся обоснованием вложенного цикла поиска. Условия  $Q(i - j)$  и  $P(i - j, j)$  являются глобальными инвариантами циклов так же, как и отношения  $0 \leq i < N$  и  $0 \leq j < M$ . Напомним, что  $Q(i - j)$  есть условие первого вхождения  $j$  букв тому назад относительно текущего индекса сканирования  $i$ , а  $P(i - j, j)$  — предикат частичного совпадения первых  $j$  букв образца, начиная с  $i - j$  буквы последовательности. Это позволяет нам отказаться от слежения за текущим указателем  $i$  первой литерой образца в последовательности и перейти к относительному расположению образца  $i - j$  в этой последовательности.

Если алгоритм заканчивает работу по исчерпанию образца  $j = M$ , то составляющая предусловия внешнего цикла  $P(i - j, j)$  обратится в  $P(i - M, M)$ , т. е. по определению предиката частичного совпадения  $P$  оно начинается с позиции  $i - M$ . Если же выполнение закончено по концу последовательности из-за  $i = N$ , то, поскольку при этом  $j < M$ , из инварианта  $Q(i)$  следует, что совпадения вообще нет.

Теперь надо доказать, что алгоритм делает то, что надо, т. е. никогда не нарушает инварианта. Легко видеть, что вначале  $i = j = 0$  и инвариант истинен. Сначала исследуем эффект от двух операторов, *синхронно* увеличивающих  $i$  и  $j$  на единицу. Они, очевидно, не сдвигают образец вправо по строке и не делают ложным  $Q(i - j)$ , поскольку разность остается неизменной. Но, может быть, они делают ложным  $P(i - j, j)$  — вторую конъюнктивную составляющую инварианта? Обращаем внимание, что по определению цикла **while** в этой точке истинно отрицание выражения в заголовке внутреннего цикла, т. е. либо  $j < 0$ , либо  $s_i = p_j$ . Последнее условие увеличивает частичное совпадение и устанавливает истинность  $P(i - j, j + 1)$ , а первое — постулирует (свидетельствует об) истинность  $P(i - j, j + 1)$ . Следовательно, одновременная единичная инкрементация  $i$  и  $j$  не нарушает ни тот, ни другой инвариант. Осталось главное — доказать, что  $j := D(j)$  также сохраняет истинность инварианта. Надеясь отыскать такое семимильное и адекватное  $D$ , примем пока это на веру.

Чтобы составить функцию вычисления  $D$  мы должны сначала понять смысл осуществляемого ею действия. При условии, что  $D < j$ , присваивание соответствует сдвигу образца вправо на  $j - D$  позиций ( $j > D!$ ). Чтобы сдвиг был настолько большим, насколько это возможно,  $D$  должно быть как можно меньше. В частности,  $D = -1$  означает максимальный сдвиг на весь образец.

Если инвариант  $P(i - j, j) \& Q(i - j)$  после присвоения  $j$  значения  $D$  истинен, то перед ним должно быть истинно  $P(i - D, D) \& Q(i - D)$ . Это предусловие и будет ориентиром

при поиске подходящего выражения для  $D$ . Основное соображение: благодаря истинности  $P(i - j, j)$  мы знаем, что

$$s_{i-j} \dots s_{i-1} = p_0 \dots p_{j-1},$$

то есть было частичное совпадение с образцом  $j$  букв тому назад относительно текущего указателя сканирования  $i$ . Поэтому условие частичного совпадения образца и фрагмента последовательности перед прыжком  $P(i - D, D)$  с  $D \leq j$ , развернутое в

$$p_0 \dots p_{D-1} = s_{i-D} \dots s_{i-1}$$

превращается в условие совпадения начала образца с его скачкообразной преодоленной частью

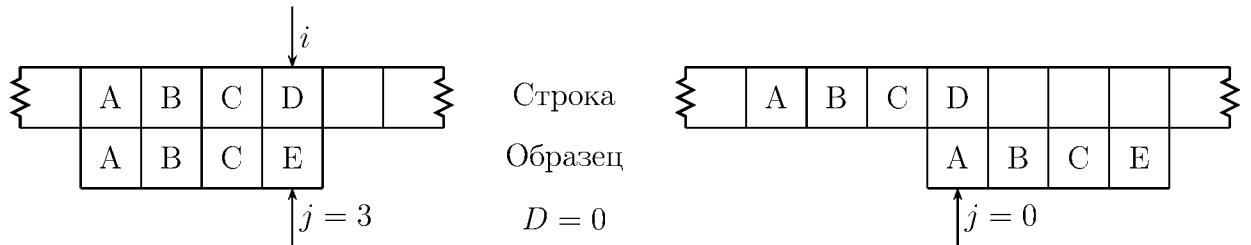
$$p_0 \dots p_{D-1} = p_{j-D} \dots p_{j-1}$$

и предикат  $\neg P(i - k, M)$  для  $k = 1 \dots j - D$  превращается в

$$p_0 \dots p_{k-1} \neq p_{j-k} \dots p_{j-1} \quad \forall k = 1 \dots j - D,$$

что подтверждает истинность одного из условий инварианта цикла  $Q(i - D)$ .

Эти соотношения позволяют сделать важный вывод: значение  $D$  определяется только отыскиваемым образцом, точнее, его самоподобием (фрактальностью) и не зависит от сканируемой последовательности. Кроме того, из этих условий следует, что для определения  $D$  мы должны для каждого смещения в образце  $j$  найти *наименьшее*  $D$ , т. е. *самую длинную* последовательность литер образца, непосредственно предшествующих позиции  $j$ , которая полностью совпадает с началом образца. Для каждого  $j$  такое  $D$  будем обозначать  $d_j$ . Так как эти значения зависят только от образца, то перед началом поиска по какому-либо образцу следует вычислить таблицу  $d_i$  для этого образца, причём согласно вышеупомянутым соотношениям эти вычисления представляют собой ни что иное, как быстрое поисковое исследование самого образца этим же методом. К тому же затраты на построение таблицы  $d_j$  никак не больше  $M^2$ , а размер образца  $M$ , конечно же, много меньше длины последовательности  $N$ . Поэтому однократное вычисление  $d_i$  (претрансляция образца [54]) не окажет заметного влияния на время поиска. Для повторного (и многократного!) поиска данного образца эту таблицу вычислять не потребуется. Конкретные примеры функции  $d_j$ :



Присваивание  $j := B$  сдвигает образец на  $j - D$  позиций.

						$i, j$		
Строка		A	A	A	A	A	C	
Образец		A	A	A	A	A	B	
Сдвинутый образец		A	A	A	A	A	B	

$$j = 5, d[5] = 4, (\max \text{ shift} = j - d[j] = 1)$$

$$p[0] \dots p[3] = p[1] \dots p[4]$$

						$i, j$		
Строка		A	B	C	A	B	D	
Образец		A	B	C	A	B	C	
Сдвинутый образец					A	B	C	A B C

$$j = 5, d[5] = 2, (\max \text{ shift} = j - d[j] = 3)$$

$$p[0] \dots p[1] = p[3] \dots p[4]$$

$$p[0] \dots p[2] \neq p[2] \dots p[4]$$

$$p[0] \dots p[3] \neq p[1] \dots p[4]$$

						$i, j$		
Строка		A	B	C	D	E	A	
Образец		A	B	C	D	E	F	
Сдвинутый образец						A B C		

$$j = 5, d[5] = 0, (\max \text{ shift} = j - d[j] = 5)$$

$$p[0] \dots p[0] \neq p[4] \dots p[4]$$

$$p[0] \dots p[1] \neq p[3] \dots p[4]$$

$$p[0] \dots p[2] \neq p[2] \dots p[4]$$

$$p[0] \dots p[3] \neq p[1] \dots p[4]$$

Частичное совпадение с образцом и вычисление  $d_j$

							↓	
Строка		A	B	C	D	E	F	
Образец		A	B	C	D	E	A	
Сдвинутый образец							A B	

$$j = 5, d[5] = -1, (\text{shift} = j - d[j] = 6)$$

Строка		A	B	C	D	E	F		
Образец		A	B	C	D	E	G		
Сдвинутый образец							A	B	

$$j = 5, d[5] = -1, (\text{shift} = j - d[j] = 6)$$

Образец сдвигается за его последнюю литеру

Последний пример на предыдущем рисунке наводит на мысль увеличения сдвига: так как  $p_j$  равно  $A$  вместо  $F$ , то соответствующая литера последовательности не может быть  $A$  из-за того, что условие  $s_i \neq p_j$  заканчивает цикл. Следовательно, сдвиг на 5 не приведёт к последующему совпадению, и поэтому можно увеличить сдвиг до шести. Учитывая это, мы переопределим вычисление  $d_j$  как поиск самой длинной *совпадающей* последовательности в самом образце

$$p_0 \dots p_{d_j-1} \neq p_{j-d_j} \dots p_{j-1}$$

с дополнительным ограничением  $p_{d_j} \neq p_j$ . Если никаких совпадений нет, то мы считаем  $d_j = -1$ , что даёт сдвиг аж на целый образец относительно его текущей позиции (рис. 1.12, внизу).

Ясно, что вычисление  $d_j$  само представляет собой поиск в строке и мы для этого тоже можем использовать сам КМП-алгоритм. Первая часть предлагаемой программы *KMP* как раз и содержит такое самоприменение КМП-алгоритма к вводимому образцу. Далее, после построения таблицы, осуществляется быстрый поиск образца в ранее введённой последовательности. Как всегда, ускорение поиска обусловлено сокращением перебора и неудачных попыток сопоставления с образцом путём больших сдвигов на дальние расстояния, величина которых гарантирована по построению таблицы  $d_j$ .

```

program KMP(input, output);
const Mmax = 100;
Nmax = 10000;

var i, j, k, k0, M, N : integer;
c : char;
p : array [0..Mmax - 1] of char; { Образец для поиска }
s : array [0..Nmax - 1] of char; { Последовательность }
d : array [0..Mmax - 1] of integer; { Таблица сдвигов }

begin
N := 0;
while not eofn do begin { Чтение последовательности }
  read(c);
  s[N] := c;
  N := N + 1;
end;
readln;
while not eof do begin { Поиск в последовательности s всех образцов, заданных
  в файле input }

```

```

repeat { Чтение образца... }
  writeln;
  write('>_');
  M := 0;
  while not eoln do begin { Образец — строка входного текстового файла }
    read(c);
    p[M] := c;
    M := M + 1;
  end;
  readln;
  writeln;
until M > 0; { ... до тех пор, пока не будет введен образец ненулевой длины }
j := 0;
k := -1;
d[0] := -1; { Не было совпадений }
while j < (M - 1) do begin { Построение таблицы — тот же поиск }
  while(k >= 0) and (p[j] <> p[k]) do { В первый раз k = -1 и цикл не
    выполняется, d[0] задано вручную (-1) }
    k := d[k]; { Поискем-ка сначала в образце }
    j := j + 1;
    k := k + 1;
  if p[j] = p[k] then
    d[j] := d[k]
  else
    d[j] := k;
  end; { Таблица построена тем же методом КМП }
{ Начало поиска }
i := 0;
j := 0;
k := 0;
while(j < M) and (i < N)do begin
  while(k <= i) do begin
    write(s[k]);
    k := k + 1;
  end;
  while(j >= 0) and (s[i] <> p[j]) do
    j := d[j];
    i := i + 1;
    j := j + 1;
  end;
  if j = M then write('!_Найдено')
end;
end.

```

### 6.1.5.1 Анализ КМП-поиска

Точный анализ КМП-поиска, как и сам алгоритм, весьма сложен. Его авторами получена сложностная оценка числа сравнений  $O(M+N)$ , существенно лучшая, чем оценка прямого поиска  $O(N * M)$ . Достоинством КМП-метода является его однопроходность, ведь указатель сканирования  $i$  никогда не возвращается назад, и электромеханическим устройствам не придётся делать реверс, преодолевая инерцию. Это также удобно для поиска на удалённых (сетевых) дисках. В начале третьего тысячелетия А. Л. Калининым [] было предложено обобщение КМП-алгоритма на случай одновременного поиска нескольких образцов. Алгоритм Кнута-Мориса-Пратта-Калинина позволил выполнить эффективную реализацию антиспамового фильтра почтовых сообщений Kaspersky Antispam и явился одним из результатов его кандидатской диссертации.

### 6.1.6 Алгоритм Бойера-Мура

Остроумная схема КМП-поиска даёт подлинный выигрыш только тогда, когда неудаче предшествовало некоторое число совпадений. Лишь в этом случае образец сдвигается более чем на единицу. К несчастью, это скорее исключение, чем правило: совпадения встречаются реже, чем несовпадения. Поэтому реальный выигрыш от использования КМП-стратегии в большинстве случаев поиска весьма незначителен. В 1977 году Р. Бойер и Д. Мур предложили другой метод поиска, который не только улучшает обработку самого плохого случая (когда и в образце, и в последовательности многократно повторяется небольшое число букв), но и даёт выигрыш в промежуточных ситуациях.

БМ- поиск основан на необычном для большинства культур письменности соображении — сравнение литер проводится не с начала образца, слева направо, а наоборот, с конца образца справа налево. Чтение в обратном направлении характерно для семитских языков и нотаций. Как и в случае КМП-поиска, сначала по образцу генерируется таблица сдвигов, в которой задаются максимально допустимые в каждой ситуации прыжки вперёд. Пусть для каждого знака алфавита  $x$   $d_x$  означает его расстояние от самого правого (семитское чтение!) вхождения в образец до его конца. Если в процессе поиска обнаружено расхождение образца и строки, то образец сразу же можно сдвинуть вправо на  $d_{P_{M-1}}$  позиций, т. е. на число позиций, скорее всего, большее единицы. Если  $P_{M-1}$  в образце вообще не встречается, то сдвиг можно сделать сразу на всю длину образца. Вот пример, иллюстрирующий этот процесс:

Hoola-Hoola girls like Hooligans.

Hooligan

Hooligan

Hooligan

Hooligan

Мы видим, что здесь количество шагов заметно меньше, чем в КМП-алгоритме для этого же примера (нам даже не пришлось использовать многоточие!)

Поскольку сравнение литер теперь идёт *справа налево*, будет удобно модифицировать предикаты частичного совпадения  $P$  и первого вхождения  $Q$ : заменив в первом из них 0 на  $j$ ,  $i$  на  $M$  и  $i$  на  $i - j$ , а во втором  $M$  на 0 и  $i$  на  $k$  в соответствии с вводимой БМ-алгоритмом инверсией порядка сравнения

$$P'(i, j) = \forall k : j \leq k < M : s_{i-j+k} = p_k$$

$$Q'(i) = \forall k : 0 \leq k < i : \neg P'(k, 0)$$

Предикат  $P'$  указывает, что в последовательности, начиная с позиции  $i$ , не совпало  $j$  букв *хвоста (суффикса)* образца, т. е. фактически  $i + M - j - 1$ -ая ее литера сравнивается с  $M - 1$ -ой литературой образца,  $i + M - j - 2$ -ая — с  $M - 2$ -ой, и т. д.  $i$ -тая литера последовательности, как нетрудно видеть, совпадает с  $j$ -той литературой образца. Полное совпадение достигается тогда, когда не совпало 0 букв, поэтому предикат  $P(i, j)$  становится истинным при  $j = 0$ :  $P(i, 0) = \text{true}$ .

В отличие от  $P'$   $Q'$  действует все равно в соответствии с направлением просмотра последовательности слева направо и утверждает, что в любой ее позиции левее  $i$ -той ( $0 \leq k < i$ ) полных совпадений ( $P'(k, 0)!$ ) не было.

Сформулируем с помощью этих предикатов инварианты поиска с реверсным сравнением в упрощённой версии алгоритма Бойера-Мура

```

i := M;
j := M;
while (j > 0) and (i < N) do begin
  {Q(i - M)}
  j := M;
  k := i;
  while (j > 0) and (s[k - 1] = p[j - 1]) do begin
    {P(k - j, j) & (k - j = i - M)}
    k := k - 1;
    j := j - 1
  end;
  i := i + d[s[i - 1]]
end;
```

Индексы  $i$ ,  $j$  и  $k$  удовлетворяют условиям  $0 \leq j \leq M$  и  $0 \leq i, k \leq N$ . Поэтому из окончания цикла при  $j = 0$  вместе с  $P(k - j, j)$  следует совпадение в  $k$ -й позиции  $P(k, 0)$ .

При окончании цикла с  $j > 0$  необходимо, чтобы  $i$  достигло значения  $N$ , следовательно, истинность  $Q(i - M)$  влечёт истинность  $Q(N - M)$ , что означает отсутствие совпадений. Убедимся, что  $Q(i - M)$  и  $P(k - j, j)$  действительно инварианты двух циклов. В начале повторений они, естественно, истинны как  $Q(0)$  так и  $P(0, M)$ , и ни полного, ни частичного совпадения нет.

Далее необходимо разобраться в эффекте двух операторов, *синхронно* уменьшающих  $k$  и  $j$ . На  $Q(i - M)$  они действуют, и так как было выяснено, что  $s_{k-1} = p_{j-1}$ , то справедливость предусловия  $P(k - j, j - 1)$  гарантирует истинность  $P(k - j, j)$  в качестве постусловия. Если внутренний цикл заканчивается при  $j > 0$ , то из  $s_{k-1} \neq p_{j-1}$  следует

ложность  $P(k - j, 0)$ , так как

$$\neg P(i, 0) = \exists k : 0 \leq k < M : s_{i+k} \neq p_k$$

Более того, из  $k - j = M - i$  следует  $Q(i - M) \& \neg P(k - j, 0) = Q(i - M + 1)$ , фиксирующее несовпадение в позиции  $i - M + 1$ .

Теперь нам нужно показать, что оператор  $i := i + d_{s_i}$  никогда не делает инвариант ложным и цикл делает то, что надо. Гарантирано, что перед этим оператором  $Q(i + d_{s_{i-1}} - M)$  истинно. Так как мы знаем, что  $Q(i + 1 - M)$  истинно, то достаточно установить истинность отрицания  $\neg P(i + h - M, 0)$  для  $h = 2, 3, \dots, d_{s_{j-1}}$ . Вспомним, что  $d_x$  определено нами как расстояние от самого правого вхождения знака  $x$  в образец до его конца. Формально это может быть записано так:

$$\forall k : M - d_x \leq k < M - 1 : p_k \neq x$$

т. е. далее в образце несовпадение. Подставляя  $s_i$  вместо  $x$ , получаем

$$\begin{aligned} \forall h : M - d_{s_{i-1}} \leq h < M - 1 : s_{i-1} \neq p_h &- \text{несовпадение} \\ \forall h : 1 < h \leq d_{s_{i-1}} : s_{i-1} \neq p_{h-M} \\ \forall h : 1 < h \leq d_{s_{i-1}} : \neg P(i + h - M, 0) \end{aligned}$$

что и требовалось доказать.

Ниже приводится программа упрощённого алгоритма Бойера-Мура, построение которой аналогично предыдущей программе. Упрощение заключается в том, что используется только правило плохой литеры без правила хорошего суффикса. Об этом правиле можно прочитать в [87].

```

program BM(input, output);
const Mmax = 100;
      Nmax = 1000;

var i, j, k, i0, M, N : integer;
    c : char;
    p : array[0..Mmax - 1] of char; { образец }
    s : array[0..Nmax - 1] of char; { строка }
    d : array[char] of integer;

begin
    N := 0; { Ввод текста, в котором будет осуществляться поиск }
    while not eoln do begin
        read(c);
        s[N] := c;
        N := N + 1;
    end;
    readln;
    repeat
        writeln;

```

```

write('>_');
M := 0; { Считывание образца }
while not eoln do begin
  read(c);
  p[M] := c;
  M := M + 1;
end;
readln;
if M <> 0 then begin { Если образец пуст, поиск не выполняется }
  { Заполнение таблицы сдвигов d }
  for c := chr(0) to chr(255) do
    d[c] := M; { Пока все буквы в образце отсутствуют }
  for j := 0 to M - 2 do
    d[p[j]] := M - j - 1; { В массиве d остается самое правое (последнее!)
      вхождение буквы в образец (ближайшее к концу при чтении задом
      наперед!) }

{ Поиск подстроки }
i := M;
i0 := 0;
repeat
  writeln;
  if i0 > 0 then
    write('_ : i0);
    while i0 < i do begin
      write(s[i0]);
      i0 := i0 + 1;
    end;
    j := M;
    k := i;
    repeat
      k := k - 1;
      j := j - 1;
    until (j < 0) or (p[j] <> s[k]);
    i := i + d[s[i - 1]];
    until(j < 0) or (i > N);
    if j < 0 then
      write('!_Найден');
    end;
  until eof;
end.

```

#### 6.1.6.1 Анализ алгоритма Бойера-Мура

При публикации алгоритма авторы привели детальный анализ его производительности. Замечательные свойства этого алгоритма в том, что почти всегда, кроме специально

построенных примеров (типа «убийцы КМП» задом наперед), он требует значительно меньше  $N$  сравнений, проходя саженью образца по рулетке последовательности. В самых благоприятных обстоятельствах, когда последняя литера образца всегда не совпадает с соответствующими литерами текста, проходимого аршином длины  $M$ , число сравнений равно  $N/M$ . Худшая оценка алгоритма  $NM$  маловероятна.

Публикую свою работу после статьи Кнута, Мориса и Пратта, Бойер и Мур приводят соображения по поводу дальнейших усовершенствований алгоритма. Одно из них — объединить стратегию Бойера-Мура, обеспечивающую большие сдвиги в случаях *несовпадения*, со стратегией Кнута-Мориса-Пратта, дающей существенные сдвиги при частичном совпадении. Комбинированная стратегия базируется на двух таблицах: КМП и БМ. В каждой ситуации поиска из двух сдвигов выбирается больший, поскольку оба алгоритма гарантируют, что никакой меньший сдвиг не может привести к совпадению. Анализируя эту изощренную стратегию, проф. Н. Вирт пессимистически иронизирует, что такое усложнение генерации таблиц и самого поиска вряд ли приведет к существенному выигрышу[54].

### 6.1.7 Алгоритм Рабина-Карпа

Рабин и Карп изобрели еще один алгоритм поиска подстрок, который эффективен на практике и к тому же обобщается на многомерный случай (например, поиск на двумерной решетке). Хотя в худшем случае время работы алгоритма Рабина-Карпа  $O(M(N-M+1))$ , в среднем он работает достаточно быстро.

Без ограничения общности предположим, что алфавит последовательности и образца — цифровой и строки имеют простую числовую интерпретацию. Если  $p[0 \dots M-1]$  — образец, то через  $\mathcal{P}$  обозначим число, десятичной записью которого он является. В последовательности  $s[0 \dots N-1] \forall j = 0, 1, \dots, N-M-1$  обозначим через  $\mathcal{S}_j$  число, десятичным изображением которого является подстрока  $s[j \dots j+M-1]$ . Очевидно,  $j$  является допустимым сдвигом при ускоренном поиске тогда и только тогда, когда соответствующая подстрока  $s_j$  совпадает с образцом  $p$ . Если бы мы могли вычислить  $\mathcal{P}$  за время  $O(M)$ , и все  $\mathcal{S}_i$  за время  $O(N)$ , то мы смогли бы за это же время найти все допустимые сдвиги, сравнивая  $\mathcal{P}$  со всеми  $\mathcal{S}_j$ . Однако для длинных образцов это может привести к слишком большим числам даже для 64-битной целочисленной и адресной арифметики, которая до сих пор остается уделом профессиональных вычислений.

Вычислить  $\mathcal{P}$  за линейное время  $O(M)$  можно по схеме Горнера:

$$\mathcal{P} = p_{M-1} + 10(p_{M-2} + \dots + 10(p_1 + 10p_0) \dots)$$

Точно также за время  $O(M)$  можно вычислить  $\mathcal{S}_0$ .

Чтобы вычислить  $\mathcal{S}_1, \mathcal{S}_2, \dots, \mathcal{S}_{N-M-1}$  за время  $O(N-M)$  заметим, что при известном  $\mathcal{S}_j$  можно вычислить  $\mathcal{S}_{j+1}$  за постоянное (а, значит, очень малое) время:

$$\mathcal{S}_{j+1} = 10(\mathcal{S}_j - 10^{M-1}s_j) + s_{j+M}$$

Чтобы получить строку  $s[j+1 \dots j+M]$  из  $s[j \dots j+M-1]$  надо удалить первую цифру старой строки (это аналогично вычитанию  $10^{M-1}s_j$ ) и приписать к ней справа следующую цифру из последовательности (текстовый аналог сложения с цифрой  $s_{j+M}$ ). Заметим, что константу  $10^{M-1}$  можно вычислить заранее, причем за время, пропорциональное  $O(\log M)$

(этот способ основан на двоичном разложении числа  $M$  [36]). Таким образом, все числа  $\mathcal{P}$ ,  $\mathcal{S}_0, \mathcal{S}_1, \dots, \mathcal{S}_{N-M-1}$  могут быть найдены за время  $O(N + M)$ , за это же время могут быть найдены все вхождения образца  $p$  в последовательность  $s$ .

До сих пор мы не касались вопроса о величине чисел  $\mathcal{P}$  и  $\mathcal{S}_j$ . При достаточно длинном образце эта характеристика может оказывать существенное влияние на время работы алгоритма, поскольку затраты на непосредственное вычисление и хранение чисел, длиннее машинного слова, линейны, а не постоянны, как в случае аппаратной реализации. К счастью, эту трудность можно обойти таким образом: надо проводить вычисления  $\mathcal{P}$  и  $\mathcal{S}_j$  по модулю  $q$ . Тогда все числа не будут превосходить  $q$  и можно считать, что каждое из чисел  $\mathcal{P}$  и  $\mathcal{S}_j$  действительно будет вычислено за постоянное время. В общем случае для алфавита  $\{0, 1, \dots, d\}$  выбирают такое *простое* число  $q$ , что  $dq$  помещается в машинное слово. Бессмысленно выбирать не простые числа, поскольку вероятность холостого срабатывания возрастает пропорционально квадрату  $\text{НОД}(q, \mathcal{S}_j, \mathcal{P})$ . Соотношение для вычисления  $\mathcal{S}_{j+1}$  принимает вид:

$$\mathcal{S}_{j+1} = (d(\mathcal{S}_j - s_j h) + s_{j+M}) \bmod q, \quad h \equiv d^{M-1} \pmod{q}.$$

Вычисления по модулю  $q$  всем хороши, кроме одного: из  $\mathcal{S}_j = \mathcal{P} \pmod{q}$  еще не следует  $\mathcal{S}_j = \mathcal{P}$ . Из неравенства  $\mathcal{S}_j \neq \mathcal{P} \pmod{q}$  следует, что образец не входит в последовательность, начиная с позиции  $j$ , а вот в случае  $\mathcal{S}_j = \mathcal{P} \pmod{q}$  необходимо проверить, совпадают ли строки  $p[0 \dots M-1]$  и  $s[j \dots j+M-1]$ . Если они совпадают, то вхождение найдено, в противном случае имеет место *холостое срабатывание* этого оценочного поискового алгоритма. Если простое число  $q$  достаточно велико, то вероятность холостых срабатываний будет очень мала.

Оценим максимальную длину образца, гарантированно отыскиваемого без холостых срабатываний для аппаратно реализованной 64-битной арифметики при использовании вполне достаточного для практики 8-битного алфавита из 256 знаков. Как было отмечено выше, в качестве множителя  $q$  необходимо выбрать простое число, такое что  $dq < 2^{64}$ . Отсюда следует, что  $q < 2^{56}$ , и существуют два таких образца  $p_1$  и  $p_2$  длиной по 7 знаков каждый, что  $p_1 = p_2 \pmod{q}$ . Поскольку длина образца измеряется сугубо целым числом байтов, то придется уступить еще один байт и перейти в следующий логарифмический поддиапазон  $[2^{48}, 2^{56})$ , в котором найдется простое число  $q$ :  $2^{48} < q < 2^{56}$ , и чем ближе к правой границе, тем лучше. Это уменьшит вероятность холостых срабатываний.

```

program RK(input,output);
const Mmax = 100; { Максимальная длина образца }
Nmax = 10000; { Максимальная длина текста }
d = 256; { Основание системы счисления для интерпретации образца и
подстрок последовательности – достаточно для 8–битных кодировок }
q = 27021597764222939; { Наибольшее простое (?) число от 2 до MaxInt для 64–
битной арифметики }
{q = 8388593}; { Для кухонных и бытовых (32 бит) процессоров }

var N, M : integer; { Фактическая длина текста и образца }
p : array [0..Mmax - 1] of char; { Образец }
s : array [0..Nmax - 1] of char; { Текст }
p0, s0 : integer; { Значения характеристических чисел образца и фрагмента
последовательности (так называемая бегущая сумма) }
```

```

h : integer; { Множитель для вычисления бегущей суммы равен значению
    функции  $h = d^{m-1} \pmod{q}$  }
i, k : integer;
begin
    writeln; { Считывание текста }
    N := 0;
    while not eoln do begin
        read(s[N]);
        N := N + 1;
    end;
    readln;
    repeat
        writeln;
        write('>');
        M := 0; { Считывание образца }
        while not eoln do begin
            read(p[M]);
            M := M + 1;
        end;
        readln;
        writeln;
        if M <> 0 then begin
            h := 1;
            for i := 1 to M - 1 do
                h := h * d mod q; {  $h = d^{M-1} \pmod{q}$  }
            p0 := 0;
            s0 := 0;
            for i := 0 to m - 1 do begin { Вычисление характеристических чисел }
                p0 := (d * p0 + ord(p[i])) mod q;
                s0 := (d * s0 + ord(s[i])) mod q;
            end;
            for i := 0 to N - M - 1 do begin
                if p0 = s0 then begin { Если характеристические числа совпали }
                    k := 0; { Прямое сравнение подстроки с образцом }
                    while (k <= M) and (p[k] = s[i + k]) do
                        k := k + 1;
                    if k >= M then { Совпадение с образцом }
                        writeln(i);
                end;
                { Несовпадение, вычисление следующего характеристического числа  $S_j$  }
                s0 := ((s0 - (h * ord(s[i]))) * d + ord(s[i + M])) mod q;
            { продвижение по образцу — позиционное вычитание старой буквы из конца
                числа (старший разряд) и добавление новой в начало числа (младший разряд)
            }
            if s0 < 0 then
                s0 := s0 + q
        end;
    end;
end.

```

```

    end
end
until m = 0;
{ Вот зачем нам понадобилась 256-ричная система счисления из I семестра! }
end.

```

## 6.2 Алгоритмы сортировки

### 6.2.1 Введение

Сортировка представляет собой хороший пример задачи, для решения которой существует множество различных алгоритмов. Каждый из них имеет свои достоинства и свои недостатки, и выбор нужного алгоритма зависит от многих конкретных условий [52, 65].

Сортировку следует понимать как процесс перестановки заданного множества объектов в некотором порядке. Цель сортировки — облегчить последующий поиск элементов в таком отсортированном множестве. Сортировка — это один из универсальных основополагающих видов обработки данных. Телефонные книги, словари, оглавления библиотек, прайс-листы — вот примеры отсортированных для поиска множеств хранимых объектов [54].

Сортировка является непременным разделом любого фундаментального курса обучения программированию [36]. При построении алгоритмов сортировки используется множество классических универсальных приемов. Трудно найти какую-либо другую задачу, для которой было бы изобретено такое огромное разнообразие алгоритмов. Алгоритмы сортировки идеально подходят для сложностного анализа. С другой стороны, сортировки, как и поиск, показывают, что за быстроту работы и экономию памяти приходится расплачиваться существенным усложнением алгоритмов и структур данных.

Выбор алгоритма всегда зависит от структуры обрабатываемых данных. В случае сортировки эта зависимость столь глубока, что соответствующие методы распадаются на два почти непересекающихся класса — *сортировку массивов* и *сортировку файлов* (*последовательностей*). Иногда их называют *внутренними* и *внешними* сортировками, поскольку массивы хранятся в основной (оперативной, внутренней) памяти машины с произвольным доступом, а файлы обычно размещаются в медленной, но более емкой, дешевой и долговременной внешней памяти, на электромеханических устройствах, основанных на поступательном или вращательном движении носителя. Уже на примере сортировки колоды игральных карт для пасьянсов или фокусов ясно, что удобнее разложить все карты на большом столе в прямом доступе так, чтобы все они были одновременно видны. Если же сортируемые карты по каким-либо причинам должны оставаться в колоде, то сортировка идет вслепую, поскольку видна только одна карта. Так же приходится сортировать и очень большие колоды, для которых не хватит никакого стола. Заметим, что раскладка пасьянса является своеобразным недетерминированным процессом сортировки, в то время как гадание скорее является поиском с существенно неформальной семантической интерпретацией.

Для постановки задачи сортировки введем некоторые понятия и обозначения. Сортируемые элементы будем обозначать  $a_1, a_2, \dots, a_n$ . Сортировка есть некоторая перестановка этих элементов  $a_{k_1}, a_{k_2}, \dots, a_{k_n}$ , где  $(k_1, k_2, \dots, k_n)$  — перестановка последовательности

$1, 2, \dots, n$ ), для которой выполнено некоторое отношение порядка  $f$ :

$$f(a_{k_1}) \leq f(a_{k_2}) \leq \dots \leq f(a_{k_n}).$$

В случае, когда элементы  $a_1, a_2, \dots, a_n$  непосредственно сравнимы, отношение порядка  $f$  вычислять не требуется, а достаточно хранить сами сравниваемые компоненты в соответствующих полях элементов  $a_i$ . Так же, как и в случае поиска, сравнимые поля элементов сортировки называются *ключами*. Типы других компонент элементов, хранимых с ключами, могут быть самыми разнообразными, и поэтому элементы обычно имеют комбинированный тип:

```
type Item = record
    key : integer;
    { описание других компонент }
end;
```

Итак, ключ идентифицирует каждый элемент множества и определяет его итоговое местоположение в упорядочиваемой последовательности. Для исследования алгоритмов сортировки существенен только ключ. Для простоты мы будем полагать ключ скалярным и, без ограничения общности, целочисленным. Как обрабатывать составные ключи, мы уже знаем на примере поиска в таблицах.

Метод сортировки называется *устойчивым* (*стабильным*), если в процессе сортировки относительное расположение элементов с равными ключами не изменяется. Устойчивость сортировки полезна, когда предполагается последующее упорядочивание по вторичным ключам среди равнозначных элементов.

Метод сортировки называется *естественным*, если в процессе сортировки учитывается частичная или полная упорядоченность элементов сортируемого множества.

Методы сортировки также могут быть классифицированы по использованию сравнений сортируемых элементов.

Помимо упомянутых классических книг Д. Кнута, Н. Вирта и Г. Лорина сортировки достаточно подробно рассматриваются в недавно изданных учебниках Массачусетского Технологического Института [85] и Принстонского университета [88]. Хорошая книга Ахо, Хопкрофта и Ульмана отличается примерами на стандартном Паскале [90].

### 6.2.2 Сортировка массивов

Основное ограничение внутренней сортировки традиционно было связано с дефицитной оперативной памятью. Это означает, что перестановки элементов в процессе сортировки должны выполняться на месте, без использования вспомогательных массивов. Другим критерием экономичности алгоритма сортировки является временная сложность, которую мы будем выражать в виде функции  $f(C(n), M(n))$ , где  $n$  — число сортируемых элементов,  $C$  — число сравнений ключей и  $M$  — число перестановок (пересылок) элементов. Сравнения и пересылки рассматриваются отдельно, поскольку их времена могут отличаться существенно.

Рассмотрение алгоритмов сортировки начнем с т. н. *прямых* методов. С этих простых и неэффективных методов началась история и эволюция сортировок.

### 6.2.2.1 Сортировка вставкой

Этот метод широко используется при игре в карты: игрок обычно располагает имеющиеся карты в левой руке веером в порядке возрастания ранга, а сдаваемая карта вставляется сообразно ее рангу после последней карты того же ранга. Сортировка *вставкой* (она же — сортировка *прямым включением*) представляет собой вариант этой карточной идеи. Сортируемые элементы мысленно делятся на уже готовую (отсортированную) последовательность  $a_1, \dots, a_{i-1}$  и исходную (неупорядоченную) последовательность  $a_i, \dots, a_n$ . Вначале первая последовательность должна быть пуста, но мы включим в нее первый элемент множества, поскольку последовательность из одного элемента всегда упорядочена! Этот элемент не обязательно минимальный, и впоследствии он может быть переставлен на подобающее место. Схема алгоритма сортировки вставкой такова:

```
{ Неупорядоченная последовательность простирается от i-ого до n-ого
    элементов }
for i := 2 to n do begin
    x := a[i];
    { Вставка элемента x в последовательность a[1..i] }
end;
```

Ниже приведен пример работы алгоритма прямой вставки для последовательности 44 55 12 42 94 18 06 67 (вставленные элементы выделены полужирным шрифтом, вставляемые — курсивом):

Исходные ключи	44	55	12	42	94	18	06	67
$i = 2$	44	<b>55</b>	12	42	94	18	06	67
$i = 3$	<b>12</b>	44	55	<b>42</b>	94	18	06	67
$i = 4$	12	<b>42</b>	44	55	<b>94</b>	18	06	67
$i = 5$	12	42	44	55	<b>94</b>	<b>18</b>	06	67
$i = 6$	12	<b>18</b>	42	44	55	94	<b>06</b>	67
$i = 7$	<b>06</b>	12	18	42	44	55	94	<b>67</b>
$i = 8$	06	12	18	42	44	55	<b>67</b>	94

В схеме алгоритма мы намеренно не уточнили процесс поиска подходящего места для вставки, поскольку он может быть организован по-разному. Рассмотрим сначала пример, когда поиск идет с начала отсортированной подпоследовательности, т. е. *слева направо*.

```

procedure StrSelEasySort(var a : seq);
{ Сортировка прямой вставкой – упрощенный алгоритм }
var i, j, k : index;
    x : item;
begin
    for i := 2 to n do begin
        x := a[i]; { i-тый элемент исходной последовательности копируется в x }
        j := 1;
        { Поиск места для вставки идет слева направо }
        while (a[j] <= x) and (j < i) do
            j := j + 1;
        { Место для вставки найдено, j-тый элемент – первый, больший x }
        { Сдвигаем хвост упорядоченной последовательности – цена сдвига O(n) }
        for k := i downto j do
            a[k] := a[k - 1];
        a[j] := x; { Вставляем извлеченный элемент на освободившееся от сдвига
                    место. Сдвиг продолжается вплоть до элемента, где раньше был x }
    end;
end;

```

Умножая общее число сдвигов —  $n - 1$  на цену сдвига  $n/2$  получим квадратичную сложностную оценку  $O(n^2)$ . Такая высокая цена вызвана частыми групповыми операциями (сдвигами) и обращением с массивом как с последовательностью.

На самом деле поиск места для вставки удобно осуществлять *справа налево*, чередуя сравнения и движения по упорядоченной последовательности, как бы просеивая вставляемый элемент  $x = a_i$  через разноячеистое сите упорядоченной подпоследовательности (движение справа налево позволяет выполнять необходимые сдвиги в процессе поиска, а не после него; иногда сдвигов не бывает вообще, когда элемент головы неупорядоченной подпоследовательности включается в хвост упорядоченной без каких-либо перемещений). То есть  $x$  сравнивается с очередным элементом  $a_j$ , а затем либо помещается на место  $a_{j+1}$ , если  $a_j \leq x$ , либо элемент  $a_j$  помещается на место  $j + 1$  (сдвигается). Место  $j + 1$  свободно, поскольку либо  $j + 1 = i$  (на первом шаге), а элемент массива  $a_i$  уже продублирован в переменной  $x$ , вследствие чего запись на  $i$ -тую позицию не приведет к потере, либо, на последующих шагах, элемент на этой позиции уже записан на позицию  $j + 2$  (в силу первого шага).

Итак, просмотр упорядоченной подпоследовательности происходит справа налево и заканчивается по одной из следующих причин:

1. найден элемент  $a_j$  с ключом, меньшим ключа  $x$ ;
2. достигнут левый конец последовательности.

Доказательство корректности и завершности алгоритма вставки может быть выполнено индукцией по длине упорядоченной подпоследовательности. База индукции: последовательность из одного элемента упорядочена. Гипотеза: вставка превращает упорядоченную последовательность длины  $k - 1$  в упорядоченную последовательность длины  $k$ : в силу определения вставки на каждом шаге происходит добавление в последовательность только одного элемента, причем он записывается сразу за последним элементом  $a_j \leq x$ , быть может, в конец упорядоченной подпоследовательности, а  $a_j < a_{j+1}$  вследствие способа выбора места вставки, т. е. устойчивая упорядоченность элементов сохраняется. Таким образом, по исчерпании неупорядоченных элементов вся последовательность будет отсортирована.

```
procedure StrInsSort(var a : seq);
{ Сортировка прямой вставкой – стандартный алгоритм }
{ Основной принцип: полагая, что первые  $i - 1$  элементов отсортированы, берем
   $i$ -й элемент и вставляем его на нужное место }
var i, j : index;
  x : item;
begin
  for i := 2 to n do begin
    x := a[i]; {  $i$ -ты элемент исходной последовательности копируется в x }
    j := i; { Фиксируем стартовую позицию для поиска справа налево }
    { Сдвигаем элементы вправо }
    while (a[j - 1] > x) and (j > 1) do begin
      a[j] := a[j - 1];
      j := j - 1;
    end;
    a[j] := x; { Вставляем на нужное место }
  end;
end;

procedure StrSelSortB(var a : seq);
{ Сортировка прямой вставкой – алгоритм с барьером }
{ Основной принцип: барьер упрощает сравнение и ускоряет сортировку }
{ Недостаток: приходится наращивать структуру данных на 1 элемент }
var i, j : index;
  x : item;
begin
  for i := 2 to n do begin
    x := a[i];
    a[0] := x; { a[0] – барьер }
    j := i;
```

```

{ Копирование со сдвигом }
while (a[j - 1] > x) do begin
    a[j] := a[j - 1];
    j := j - 1;
end;
    a[j] := x; { Вставляем на освободившееся место }
end;
end;

```

Проанализируем метод вставки. Число сравнений ключей  $C_i$  на  $i$ -том этапе находится в диапазоне от 1 до  $i - 1$ , в среднем —  $i/2$ . Число перестановок элементов  $M_i$  равно  $C_i + 2$  (включая барьер). Поэтому общее число сравнений и перестановок таково:

Оценка	Сравнения	Перестановки
Минимальная	$n - 1$	$3(n - 1)$
Средняя	$\frac{n^2 + n - 2}{4}$	$\frac{n^2 + 9n - 10}{4}$
Максимальная	$\frac{n^2 + n - 4}{4}$	$\frac{n^2 + 3n - 4}{2}$

Минимальные оценки метода вставки достигаются в случае уже упорядоченной исходной последовательности. Поскольку поиск идет справа налево, наихудшая производительность имеет место, когда сортируемые элементы располагаются в обратном порядке, что влечет сдвиги максимальной величины.

Сортировка вставкой демонстрирует некоторое естественное постепенное упорядочение, когда в каждый момент времени у нас имеется частично отсортированная последовательность с полностью отсортированной подпоследовательностью в начале. Кроме того, приведенный алгоритм сортировки вставкой *устойчив*.

Алгоритм с прямой вставкой можно легко улучшить, зная об упорядоченности выходной подпоследовательности. В этом случае поиск места для вставки может быть существенно ускорен за счет применения бинарного поиска в упорядоченной части последовательности. Такой поиск возможен, поскольку она располагается в памяти прямого доступа. Соответствующая модификация алгоритма называется *методом двоичной вставки*. Приведем программу сортировки этим методом:

```

procedure BinInsSort(var a : seq);
{ сортировка двоичной вставкой }
var i, j, L, R, m : index;
    x : item;
begin
    for i := 2 to n do begin
        x := a[i];
        L := 1;
        R := i;
        { Двоичный поиск, цена сравнений снижается с n до log2 n }
        while L < R do begin { При L ≥ R место вставки обнаружено }
            m := (L + R) div 2;
            if a[m] <= x then
                L := m + 1

```

```

else
    R := m;
end;
{ Сдвиги остаются, сложность  $O(n^2)$  }
for j := i downto R + 1 do
    a[j] := a[j - 1];
    a[R] := x;
end;
end;

```

Анализ алгоритма двоичной вставки. Место включения обнаружено, если  $L = R$ . Таким образом, в конце поиска интервал должен иметь единичную длину; деление пополам проводится  $i \log_2 i$  раз, и число сравнений будет

$$C = \sum_{i=1}^n \lceil \log_2 i \rceil,$$

где верхние полускобки  $\lceil x \rceil$  означают минимальное целое, большее  $x$  [76]. При увеличении числа слагаемых  $n$  сумму можно аппроксимировать интегралом:

$$\int_1^n \log_2 x \, dx = n(\log_2 n - c) + c,$$

где  $c = \log_2 e = \frac{1}{\ln 2} = 1,4426$ , т. е. среднее число сравнений не зависит от начального порядка элементов. Если в исходном состоянии элементы упорядочены, то, из-за отбрасывания дробной части при половинном делении, двоичный поиск дает чуть большее число сравнений, а если упорядочены в обратную сторону, то чуть меньше. При отсортированном массиве мы выполняем только  $2(n - 1)$  встречное копирование в переменную  $x$  и обратно. Сдвиг не нужен и сортировка работает за линейное время.

К сожалению, частичное ускорение с  $O(n)$  до  $O(\log n)$ , даваемое двоичным поиском, касается лишь части трудоемкости сортировки вставкой — сравнений. Главный член сложностной оценки —  $M$  — продолжает оставаться квадратичным.

Сортировка двоичной вставкой может быть полезна только в том случае, когда сравнение занимает гораздо больше времени, чем копирование элементов. Например, если необходимо отсортировать множество строк, заданное массивом указателей типа **char\*** (в языке Си), то сравнение элементов (просмотр двух строк до первого несовпадения) выполняется гораздо дольше, чем их копирование, т. к. время сравнения двух строк есть  $O(n)$ , а время перестановки строк, выполняемое копированием указателей, —  $O(1)$ .

Таким образом, низкая производительность метода вставки — следствие группового сдвига всех отсортированных элементов при вставке каждого нового — сохраняется и при двоичной вставке. Этот метод, так же, как и простая вставка, не рекомендуется к практическому применению на сколько-нибудь больших массивах.

### 6.2.2.2 Сортировка выборкой

Алгоритм основан на простой идее:

- отыскивается элемент с наименьшим ключом;
- он меняется местами с первым элементом  $a_1$ ;
- процесс повторяется для оставшихся  $n - 1$ -ого,  $n - 2$ -х, … элементов до тех пор, пока не останется один, самый большой элемент.

Схема алгоритма сортировки выборкой такова

```

for i := 1 to n – 1 do begin
    { присвоить k индекс наименьшего из a[i..n] }
    { поменять местами a[k] и a[i] }
end;

```

Метод выборки в некотором смысле противоположен методу вставки. При прямой вставке на каждом шаге рассматривается только *один* очередной элемент исходной последовательности и *все* элементы готовой последовательности, среди которых отыскивается место вставки. При прямом выборе для поиска *одного* элемента с наименьшим ключом просматриваются *все* элементы исходной последовательности, и найденный минимальный элемент помещается в выходную последовательность как очередной элемент. Поскольку сортировка выборкой также выполняется на месте и обмен возможен ввиду прямого доступа к элементам массива, то вместо удаления минимального элемента из входной последовательности на его место помещается очередной рассматриваемый элемент  $a_i$ , который будет впоследствии обязательно рассмотрен среди оставшихся элементов.

Ниже приведен пример работы сортировки выборкой для последовательности 44 55 12 42 94 18 06 67 (переставленные элементы выделены полужирным шрифтом, курсивом набраны минимальные элементы в неотсортированных подпоследовательностях):

Исходные ключи	44	55	12	42	94	18	<i>06</i>	67
$i = 1$	<b>06</b>	55	12	42	94	18	<b>44</b>	67
$i = 2$	06	<b>12</b>	<b>55</b>	42	94	18	44	67
$i = 3$	06	12	<b>18</b>	<b>42</b>	94	<b>55</b>	44	67
$i = 4$	06	12	18	<b>42</b>	94	55	44	67
$i = 5$	06	12	18	42	<b>44</b>	55	<b>94</b>	67
$i = 6$	06	12	18	42	44	<b>55</b>	94	67
$i = 7$	06	12	18	42	44	55	<b>67</b>	<b>94</b>

Заметим, что на 4-ом и 6-ом шагах элементы уже оказались на своих местах, но они все-таки были дважды обменены сами с собой.

```

procedure StrSelSort(var a : seq);
{ Сортировка прямым выбором – стандартный алгоритм }
{ Основной принцип: берем элемент с наименьшим ключом и ставим его на
первое место, меняя с a[1]. Затем процесс повторяем для элементов a[2]..a[n
– 1] (т. к. a[1] – уже минимальный из рассмотренных) }
var i, j, k : index;
x : item; { Здесь x – для перестановки, а не для сдвига }
begin
for i := 1 to n – 1 do begin
    x := a[i]; { Берем очередной элемент... }

```

```

k := i; { ... и запоминаем его индекс }
{ Ищем минимальный среди оставшихся }
for j := i + 1 to n do
    { Встретился меньший выбранного — запоминаем его значение и индекс и
    продолжаем поиск }
    if a[j] < x then begin
        k := j;
        x := a[j];
    end;
    { Перестановка текущего элемента с наименьшим в подпоследовательности
    }
    a[k] := a[i];
    a[i] := x;
end;
end;

```

Доказательство корректности этого алгоритма несложно и осуществляется по индукции. Шаг индукции основан на том, что по определению минимального элемента множества минимальный среди оставшихся не меньше максимального среди ранее извлеченных минимальных элементов.

Проанализируем алгоритм выборки. Число сравнений ключей

$$C = \frac{n^2 - n}{2}$$

не зависит от их начального порядка. То есть этот метод не обладает естественной отзывчивостью на предварительную упорядоченность элементов в части их сравнения. Число перестановок минимально в случае уже упорядоченных ключей и равно  $3(n - 1)$ .

Если изначально ключи были расположены в обратном порядке, то число перестановок максимально:

$$M_{max} = \frac{n^2}{4} + 3(n - 1)$$

Для того, чтобы определить  $M_{avg}$ , необходимо рассуждать так: алгоритм просматривает массив, сравнивая каждый элемент с только что обнаруженной минимальной величиной; если он меньше первого, то выполняется некоторое присваивание. Вероятность того, что третий элемент окажется меньше двух первых, равна  $1/3$ , вероятность оказаться минимальным для четвертого —  $1/4$  и т. д. Поэтому общее число ожидаемых пересылок равно  $\frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} = H_n - 1$ , где  $H_n$  —  $n$ -ое гармоническое число. Это число можно выразить еще таким способом:

$$H_n = \ln n + g + \frac{1}{2n} - \frac{1}{12n^2} + \dots$$

где  $g = 0,577216\dots$  — константа Эйлера. Для достаточно больших  $n$  можно игнорировать дробные составляющие, поэтому среднее число присваиваний на  $i$ -том просмотре аппроксимируется выражением

$$F_i = \ln i + g + 1.$$

Среднее число пересылок  $M_{avg}$  в сортировке выборкой есть сумма  $F_i$ :

$$M_{avg} = n(g + 1) + \sum_{i=1}^n \ln i.$$

Вновь аппроксимируя эту сумму дискретных членов интегралом

$$\int_1^n \ln x \, dx = x(\ln x - 1) = n \ln n - n + 1$$

получаем, наконец, приблизительное значение

$$M_{avg} = n(\ln n + g).$$

Отсюда можно сделать заключение, что, как правило, алгоритм сортировки выборкой предпочтительнее простой вставки. Однако, если ключи вначале упорядочены или почти упорядочены, алгоритм вставки будет оставаться несколько более быстрым.

### 6.2.2.3 Обменные сортировки

Смешать, но не взбалтывать.

Джеймс Бонд

В данном разделе мы опишем простой метод сортировки, в котором обмен местами двух элементов производится не с целью вставки или выборки, а непосредственно для упорядочения. Так называемый алгоритм прямого обмена, основывается на сравнении и перестановке пар соседних элементов до тех пор, пока таким образом не будут упорядочены все элементы.

Как и при сортировке выборкой мы осуществляем проходы по массиву, постепенно сдвигая наименьший элемент оставшейся последовательности к левому краю массива. Если мы откажемся от горизонтальной ориентации сортируемых последовательностей в пользу вертикальной, то упорядочение этим методом можно интерпретировать как всплытие (погружение) пузырьков в чане с водой (в стеклянной колонне с пивом) сообразно их весу, точнее, плотности. Этот гидрогазодинамический смысл обменной сортировки и дал ей более популярное название *пузырьковой* сортировки.

Сразу внесем в нашу переборную идею такое улучшение: поскольку легчайший пузырек всплывает наверх всего за один проход, то при последующих просмотрах можно исключать из рассмотрения по одному такому элементу и тем самым несколько ускорить пузырьковую сортировку. Вспомните сортировку выборкой, когда минимум отыскивался не во всей последовательности, а в ее нерассмотренном остатке!

Пузырьковая сортировка *столбца* с ранее рассматриваемыми данными:

$i = 1$	$i = 2$	$i = 3$	$i = 4$	$i = 5$	$i = 6$	$i = 7$	$i = 8$
44	06	06	06	06	06	06	06
55	44	12	12	12	12	12	12
12	55	44	18	18	18	18	18
42	12	55	44	42	42	42	42
94	42	18	55	44	44	44	44
18	94	42	42	55	55	55	55
06	18	94	67	67	67	67	67
67	67	67	94	94	94	94	94

```
procedure BubbleSort(var a : seq);
```

{ Сортировка прямым обменом – пузырьковая, обмен здесь характернейшая особенность процесса сортировки }

{ Основной принцип: на каждом проходе сравниваются два соседних элемента и обмениваются местами. Продолжается до тех пор, пока элементы не будут упорядочены. Недостатки: (1) на последних проходах элементы могут уже быть упорядочены. (2) Пузырек вслыхивает быстро (за 1 проход до верху), а тонет медленно (за 1 проход на 1 позицию)

«Применяется: НИКОГДА!» Д. В. Сошиников }

```
var i, j, k : index;
```

```
    x : item;
```

```
begin
```

```
    for i := 2 to n do
```

```
        for j := n downto i do begin
```

```
            if a[j - 1] > a[j] then begin {  $a_i \leftrightarrow a_j$  }
```

```
                x := a[j - 1];
```

```
                a[j - 1] := a[j];
```

```
                a[j] := x;
```

```
            end;
```

```
        end;
```

```
    end;
```

Из нашего примера видно, что три последних прохода не влияют на порядок элементов из-за того, что все они уже отсортированы. Эти лишние проходы запланированы на случай самой плохой входной последовательности, в которой на первом месте расположен ее максимальный элемент. Однако, если на каждом проходе вести подсчет числа состоявшихся обменов (или фиксировать сам факт обменов), то можно улучшить пузырьковую сортировку:

```
procedure BubbleEnhSort(var a : seq);
```

{ Сортировка прямым обменом – пузырьковая улучшенная }

{ Основной принцип: если элементы уже упорядочены, сортировка прекращается }

```
var i, j, k : index;
```

```
    x : item;
```

{ swapped : boolean; – факт обмена }

```
begin
```

```
    i := 2;
```

```

repeat { Невозможность досрочного окончания цикла с параметром в
стандарте Паскаля приводит к циклу с постусловием }
{ for i := 2 to n do begin }
  k := 0; { Количество обменов данного прохода }
  { swapped := false; — обменов не было }
  { i пузырьков уже всплыло, их просматривать не нужно }
  for j := n downto i do begin
    if a[j - 1] > a[j] then begin {  $a_i \leftrightarrow a_j$  }
      u := x[j - 1];
      a[j-1] := a[j];
      a[j] := u;
      k := k + 1;
      { swapped := true; }
    end;
  end;
  until k = 0 or i > n { or not swapped };
end;

```

Еще одно улучшение — не только фиксировать наличие обмена, но и запоминать индекс последнего состоявшегося обмена  $k$ . Поскольку все пары соседних элементов выше индекса  $k$  уже упорядочены, просмотры можно заканчивать на этом индексе, большем нижнего предела параметра цикла  $i$ .

Говоря о быстром всплытии легкого пузырька, мы одновременно можем констатировать медленное погружение тяжелого. Как сказал проф. Д. Б. Подшивалов, всплывает пузырек сразу, за один проход, а тонет медленно, только на один уровень за проход. Еще одна цитата в тему: «Один плохо расположенный пузырек на тяжелом конце в массиве с обратным порядком быстро перемещаются на нужное место, а также неудачно размещенный элемент на легком конце медленно просачивается к своему окончательному положению» [54]. Такая асимметрия пузырьковой сортировки вызвана особенностью происходящих парных обменов и направлением просмотра, при которых меньший элемент снова участвует в сравнении в качестве члена следующей пары и имеет шанс таким образом быстро «проскочить в дамки». В то же время большие элементы остаются на месте без дополнительного внимания до следующего прохода, где у них, в случае проигрыша, снова не будет возможности быстрого продвижения к поверхности.

Например, массив

12 18 42 44 55 67 94 06

с помощью усовершенствованной пузырьковой будет упорядочен за 2 просмотра, а для сортировки массива

94 06 12 18 42 44 55 67

потребуется 8 проходов. Это наводит на мысль чередовать направления просмотра с целью дальнейшего ускорения сортировки. Технически чередование выписывается явно или могло бы быть реализовано чередованием прямого и обратного итераторов обхода массива, для которых по-разному определено направление просмотра «вперед». В честь барменского приспособления для встрихивания (с переворачиванием!) коктейлей, модернизированная пузырьковая сортировка названа *шайкерной*.

Шайкерная сортировка столбца даст следующие результаты:

<i>L</i>	2	3	3	4	4
<i>R</i>	8	8	7	7	4
<i>dir</i>	↑	↓	↑	↓	↑
	44	06	06	06	06
	55	44	44	12	12
	12	55	12	44	18
	42	12	42	18	42
	94	42	55	42	44
	18	94	18	55	55
	06	18	67	67	67
	67	67	94	94	94

Программа шейкерной сортировки итерирует линейную композицию из прямого и обратного прохода:

```

procedure ShakerSort(var a : seq);
{ Сортировка прямым обменом – шейкерная }
{ Основной принцип: на каждом проходе сравниваются два соседних элемента и
обмениваются местами. Обмен идет в двух направлениях. Применяется: если
известно, что элементы почти упорядочены }

var i, j, k, L, R : index;
x : item;

begin
  L := 2;
  R := n;
  k := n;
  repeat
    for j := R downto L do
      if a[j - 1] > a[j] then begin
        x := a[j - 1];
        a[j - 1] := a[j];
        a[j] := x;
        k := j { Усовершенствованный вариант: запоминается последний обмен }
      end;
    L := k + 1;
    for j := L to R do
      if a[j - 1] > a[j] then begin
        x := a[j - 1];
        a[j - 1] := a[j];
        a[j] := x;
        k := j { Усовершенствованный вариант: запоминается последний обмен }
      end;
    R := k - 1;
  until L > R;
end;

```

Проанализируем обменные сортировки. Число сравнений в простейшем варианте

$$C = \frac{n^2 - n}{2}$$

а минимальное, среднее и максимальное число перестановок элементов равно соответственно

$$\begin{aligned} M_{min} &= 0, \\ M_{avg} &= \frac{3(n^2 - n)}{4}, \\ M_{max} &= \frac{3(n^2 - n)}{2}. \end{aligned}$$

Анализ улучшенных обменных методов, особенно шейкерного, более сложен. Минимальное число сравнений  $C_{min} = n - 1$ . Д. Кнут считает [65], что среднее число проходов улучшенной пузырьковой сортировки  $O(n - k_1\sqrt{n})$ , а среднее число сравнений —  $O(\frac{n^2 - n(k_2 + \ln n)}{2})$ . Но, главное, все перечисленные усовершенствования не влияют на число перестановок. Они лишь сокращают количество избыточных двойных проверок. А ведь обмен местами двух элементов, как правило, дороже сравнения ключей. Кроме того, все эти усовершенствования не снижают квадратичного порядка алгоритмической сложности обменных сортировок: все улучшения являются константными или аддитивными термами квадратичной доминанты.

Фактически, обменная сортировка и её усовершенствования занимают некоторую среднюю позицию между простыми и ресурсоёмкими сортировками вставкой и выборкой. Только шейкерная сортировка даёт выигрыш в случае почти упорядоченного массива, например, когда осуществляются редкие вставки в отсортированный массив.

Можно показать, что среднее расстояние, на которое должен продвигаться каждый из  $n$  элементов во время любой сортировки, равно  $n/3$  позиций [65]. Поэтому простые методы сортировки, продвигающие элемент на одну позицию за шаг, квадратичны и неудовлетворительны. Серьезного улучшения скорости сортировки можно достичь только перемещая сортируемые элементы на большие расстояния в каждом такте. Перейдем к рассмотрению существенных улучшений изученных нами простых методов сортировки.

#### 6.2.2.4 Сортировка Шелла

В 1959 г. Д. Шеллом было предложено усовершенствование сортировки вставкой. Для ускорения он предложил выделять в сортируемой последовательности периодические подпоследовательности регулярного шага, в каждой из которых отдельно выполняется обычная (или двоичная) сортировка вставкой. Эти подпоследовательности пронизывают крупными стежками всю сортируемую последовательность и, по построению, дают большие перемещения элементов. Ввиду регулярного шага (периода) и прямого доступа к элементам массива (индексированного, конечно же, целым типом) прохождение этих подпоследовательностей реализуется циклом с параметром **for**. После каждого прохода шаг подпоследовательностей уменьшается и сортировка повторяется с новыми прыжками, причем переход к меньшему шагу не только не требует затрат на организацию новых подпоследовательностей, но и не ухудшает сортированность упорядочиваемого массива (теорема К п. 5.2.1 в книге [65] и лемма 6.7 в книге [88]). Последней выполняется сортировка с шагом 1, подчищающая огехи предыдущих проходов. При кратном уменьшении шага, например, 8 4 2 1 мы в конце концов получим две автономные подпоследовательности и на завершающем проходе нам придётся «много вставлять». Избежать этого можно, по

рецепту Д. Кнута, используя последовательность

$$1 \ 4 \ 13 \ 40 \ 121 \ 364 \ 1093 \ 3280 \ 9841 \dots 3n + 1 \dots$$

Эта последовательность приводит к хорошему «перемешиванию» организуемых подпоследовательностей и сводит к минимуму их изолированность, и на последней стадии сериализации вставка будет идти почти без ресурсоемких сдвигов. Наверное, сортировка Шелла может базироваться и на других простых методах, но метод вставки из них не только относительно быстрый, но и обладает позитивной реакцией на частичную упорядоченность последовательности, которая имеет место в результате предыдущих проходов. Но установлено, что оптимальные расстояния не должны быть множителями друг друга, чтобы не образовывать автономно сортируемых непересекающихся цепочек. Более того, взаимопроникновение различных цепочек должно быть максимизировано.

Приведем пример сортировки Шелла:

Исходные ключи	44	55	12	42	94	18	06	67
Сортировка с шагом 4	44	18	06	42	94	55	12	67
Сортировка с шагом 2	06	18	12	42	44	55	94	67
Сортировка с шагом 1	06	12	18	42	44	55	67	94

Выигрыш сортировки Шелла происходит за счет того, что на каждом этапе либо сортируется относительно мало элементов, либо эти элементы уже довольно хорошо упорядочены и происходит сравнительно немного перестановок, к тому же и на расстояния  $\gg 1$ .

```

procedure ShellSort(var a : seq);
{ Сортировка Шелла }
{ Основной принцип: сортировка вставкой для элементов, отстоящих на h[p]
позиций. Проходы для h[p] = 9, 5, 3, 1 }

const l = 4;

var i, j, k : index;
    m : idx;
    x : item;
    h : array[1..l] of integer;
begin
    h[1] := 9;
    h[2] := 5;
    h[3] := 3;
    h[4] := 1;
    for m := 1 to t do begin
        k := h[m];
        { Сортируем вставкой субсериализированный массив }
        for i := k + 1 to n do begin
            x := a[i];
            j := i - k;
            while (x < a[j]) and (j > 1) do begin

```

```

    a[j + k] := a[j];
j := j - k;
end;
    a[j + k] := x;
end;
end;
end;

```

В книге Н. Вирта [54] приведен барьерный вариант такой программы, упрощающий условие окончания места для вставки, причем для каждой из подпоследовательностей нужен свой барьер, пристраиваемый слева от края массива. Число барьера равно максимальному шагу.

Проанализируем метод сортировки Шелла. Недостатком сортировки Шелла является ее неустойчивость. Кроме того, сортировка Шелла может привести к резонансному явлению (трешингу) в системах с виртуальной памятью, когда члены последовательности попадают на разные страницы виртуальной памяти. Математический анализ сортировки Шелла исключительно сложен. Абсолютно оптимальные последовательности уменьшающегося шага до сих пор не построены. Но в современной монографии Седжвика [88] приведено несколько очень хороших результатов:

Последовательность шагов	Оценка
1 4 13 40 121 364 1093 3280 9841 ...	$O(n^{3/2})$
1 8 23 77 281 1073 4193 16577 ...	$O(n^{4/3})$
1 2 3 4 6 9 8 12 18 27 16 24 36 54 81 ...	$O(n \log^2 n)$

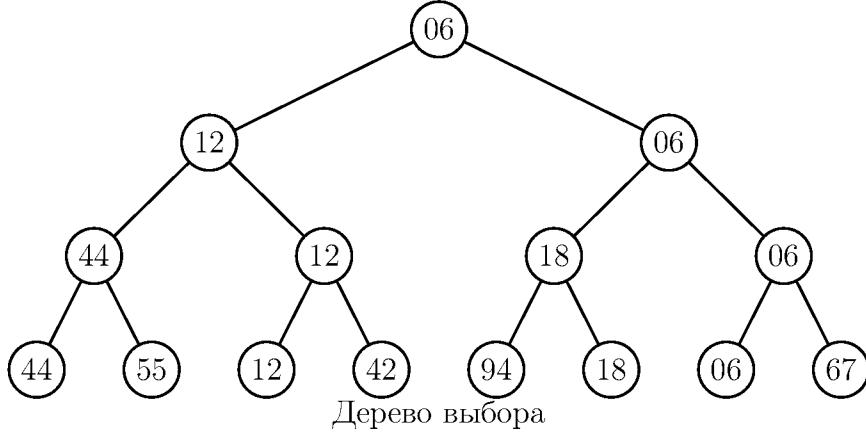
Общая степенная сложностная оценка сортировок Шелла  $O(n^{1+\delta})$ , где  $0 < \delta < 1$ , даже при малых  $\delta$  (скажем,  $10^{-6}$ ) асимптотически хуже линеарифмической  $n \log n$  и  $n \log^2 n$ . Но одновременно сортировка Шелла существенно быстрее простых сортировок с их квадратичной сложностью.

### 6.2.2.5 Пирамидальная сортировка

Сортировка выборкой была основана на повторяющемся поиске наименьшего ключа в постепенно сокращающейся последовательности элементов. Несмотря на то, что число сравнений в последующих проходах уменьшалось ( $n-1, n-2, \dots$ ), оно все равно остается квадратичным  $\left(\frac{n^2 - n}{2}\right)$ . Причина такой высокой трудоемкости сортировки выборкой в том, что при поиске минимального элемента на каждом проходе теряется ценная информация о других почти минимальных элементах. Дело в том, что поиск **линейный**, и его результат — единственное скалярное значение, никаких пометок, закладок, флагков, тегов в просматриваемой последовательности не ставится, она остается линейной и не приспособленной к сокращению перебора структурой.

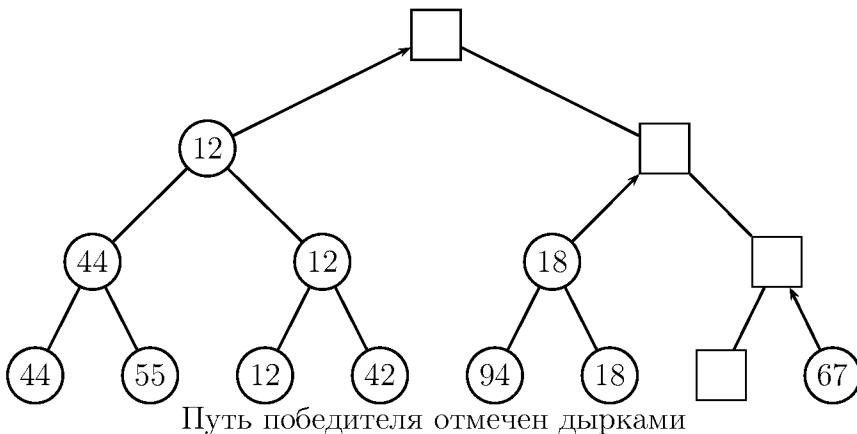
Существует модификация сортировки выборкой, оставляющая после каждого прохода гораздо больше порядковой информации, чем просто минимальное значение. В сортируемом множестве можно сравнить пары соседних элементов. В результате  $n/2$  сравнений мы получим такое же число победителей — элементов с меньшими ключами. Если среди победителей провести такое же сравнение, мы получим  $n/4$  меньших элементов. Продолжая процесс  $\lceil \log_2 n \rceil + 1$  раз, и проделав  $n - 1$  сравнение, мы получим дерево

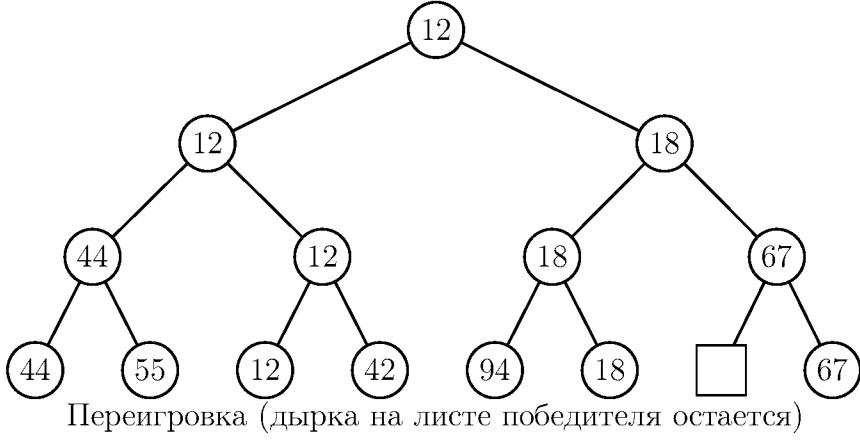
выбора минимального элемента. Кстати, дерево выбора эффективно реализует очередь с приоритетами, с той лишь разницей, что наибольшему приоритету соответствует наименьший ключ элемента, помещаемого в вершину дерева.



Раньше мы довольствовались только минимальным элементом — корнем дерева, теперь древовидная структура сохраняет турнирную историю этой своеобразной игры в поддавки и может быть использована для сокращения перебора сортируемых элементов. Итак, мы не только нашли минимальный элемент, но и сохранили иерархию субминимальных (локально минимальных) элементов, теряющую при сортировке выборкой.

Чтобы воспользоваться этой структурой, надо от победителя отправиться к призерам, осуществив спуск вдоль пути, отмеченного наименьшим элементом. Все вершины его пути надо исключить из дерева, заменив на пустой элемент (квадратную дырку). Далее, поднимаясь назад по дыркам этого же пути, надо выполнить переигровку турнира для определения нового победителя в отсутствие прежнего. При этом элемент, соревнующийся с пустым местом ( $+\infty$ , поскольку  $\min(a, +\infty) = a$ ), автоматически проходит в следующий тур. В результате этого процесса в корень переместится наименьший из оставшихся элементов. Его также можно исключить из дерева, поместив в очередь отсортированных элементов. Повторяя этот процесс до полной выемки непустых элементов из дерева, получим отсортированную последовательность.





Дадим сложностную оценку турнирной сортировки. На каждом из  $n$  шагов требуется только  $\lfloor \log_2 n \rfloor + 1$  сравнений (высота дерева!). Таким образом, общее число сравнений составит  $n \log_2 n$ . Оценим теперь стоимость построения дерева. Сложность построения дерева оценивается в  $n$  шагов [88]: для построения дерева из 127 элементов необходимо построить 32 дерева размером 3, 16 деревьев размером 7, 8 деревьев размером 15, 4 дерева размером 31, 2 дерева размером 63 и одно дерево размером 127. В худшем случае это требует  $32 \cdot 1 + 16 \cdot 2 + 8 \cdot 3 + 4 \cdot 4 + 2 \cdot 5 + 1 \cdot 6 = 120$  продвижений. Для  $n = 2^k - 1$  верхняя граница числа элементарных перемещений равна

$$\sum_{1 \leq i \leq n} i 2^{k-i-1} = 2^k - k - 1 < n.$$

Это же доказательство справедливо и для  $n \neq 2^k - 1$ .

Таким образом, общая оценка пирамидальной остается линеарифмической, что является весьма существенным улучшением не только простых квадратичных методов, но и сублинейных (малополиномиальных) шелловских  $n^{1+\delta}$ .

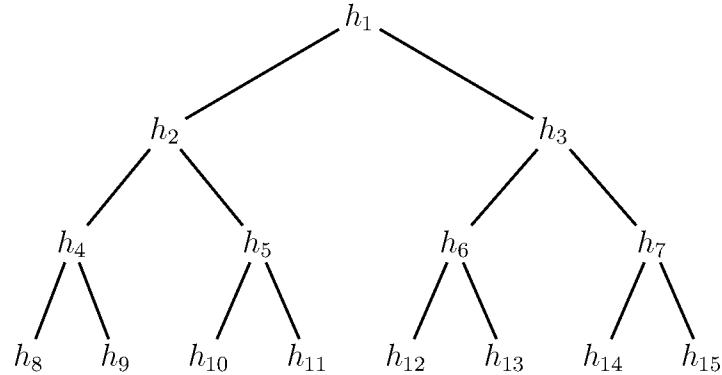
Для практической реализации сортировки с помощью дерева выбора осталось лишь указать способ эффективного построения и использования этого дерева. Если вспомнить сплошное поуровневое представление турнирного дерева, то возникает простой способ его размещения, правда, требующий двойного расхода памяти: сначала размещаются участники турнира, потом — победители пар, за ними — победители победителей и т. д. Их число —  $n + n/2 + n/4 + \dots = 2n - 1$ . Если бы не двойной расход памяти, эта простая и быстрая схема широко бы применялась. Победитель пары первого круга  $(a_i, a_{i+1})$  находится по легко вычислимому адресу  $n + i \text{div} 2$ . Чтобы заполнить второй уровень, необходимо запомнить индекс последнего записанного в первом круге элемента  $n_j + i \text{div} 2$  и т. д. Для полного турнирного дерева без неявок и отказов (т. е. с  $2^k$  участниками) заполнение дерева совсем тривиально: в качестве базового адреса для размещения уровня дерева необходимо выбирать  $2^{k-j-1}$ , где  $j$  — номер уровня. Итак, из-за дублирования в дереве выбора победивших элементов пространственная сложность алгоритма возрастает вдвое.

Для того чтобы сократить объем памяти дерева выбора с  $2^{n-1}$  элемента до  $n$  необходимо применить пирамидальное улучшение традиционных древовидных сортировок. Пирамида определяется как последовательность ключей,  $h_L, h_{L+1}, \dots, h_R$ , такая что

$$h_i \leq h_{2i} \text{ и } h_i \leq h_{2i+1} \text{ для } i = L, \dots, R/2.$$

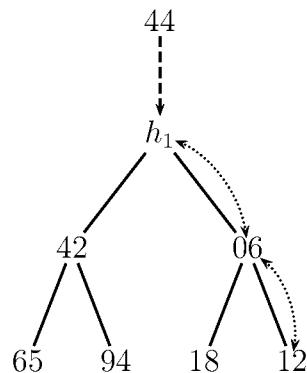
Пирамида отличается от турнирного дерева отсутствием дубликатов элементов сортируемой последовательности.

Если любое двоичное дерево отображать в массив поуровневой схемой

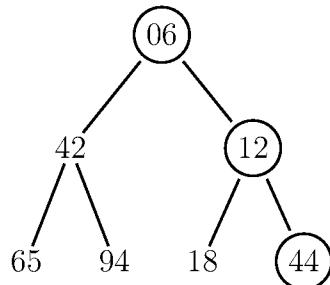


то деревья сортировки становятся пирамидами, и в верхушку попадает наименьший элемент  $h_1 = \min(h_1, h_2, \dots, h_n)$ .

Рассмотрим построение на базе некоторой пирамиды с заданными элементами  $h_{L+1}, \dots, h_R$  для некоторых значений  $L$  и  $R$  расширенной пирамиды  $h_L, \dots, h_R$  с добавленным элементом  $x$ . Возьмём, например, пирамиду  $h_1, h_2, \dots, h_7$



и добавим к ней слева элемент  $h_1 = 44$ . Вносимое в пирамиду новое значение всегда ставится в ее вершину и начинает играть в поддавки с примыкающими элементами, спускаясь вниз по направлению к наименьшему из двух подчиненных элементов, и пропуская наверх этот наименьший элемент. Элементы с индексами  $i$  и  $j$ , где  $j = 2i$  или  $j = 2i + 1$ , так обмениваемые при перестроении дерева выбора, сохраняют неизменными условия пирамиальности. В нашем примере значение 44 сначала меняется местами с 06, затем с 12, и в результате образуется дерево



Сформулируем алгоритм вставки элемента в пирамиду. Как уже говорилось, первоначально элемент помещается в ее вершину,  $i = 1$ . Рассмотрим тройку элементов с индексами  $i, 2i, 2i + 1$ . Во-первых, по построению пирамиды, элементы  $2i$  и  $2i + 1$  являются потомками  $i$ -того элемента. Выберем наименьший из этих трех элементов. Если он находится на  $i$ -той позиции, то вставка элемента завершается. Иначе обозначим позицию наименьшего элемента буквой  $j$ , где  $j = 2i$  или  $j = 2i + 1$ . Поменяем местами элементы  $i$  и  $j$ , положим  $i := j$  и продолжим процесс сравнений и обменов с  $i$ -той позиции, куда в результате итерации был помещен вставляемый элемент.

Процесс вставки в пирамиду можно проиллюстрировать на таком примере. Согласно известной шутке, в каждой иерархии любой ее член занимает место согласно уровню своей компетентности (некомпетентности). Новичок, попадая в эту структуру, сначала примеряет к себе высшую должность, а потом постепенно спускается по иерархии вниз до тех пор, пока не будет руководить теми, кто компетентнее его. Ввиду иерархичности структуры дерева выбора, скромный вариант (движение снизу вверх), в принципе возможен, но никогда не применяется, поскольку требует линейного, а не логарифмического времени.

В 1963 году Р. Флойдом был предложен лаконичный способ построения пирамиды на месте. Верхняя половина  $h_m \dots h_n$  организуемого в пирамиду массива  $h_1 \dots h_n$ , где  $m = (n \text{div} 2) + 1$  становится нижним уровнем двоичного дерева выбора. Далее, в оставшейся части массива надо построить вышестоящие уровни дерева на основе отношения порядка: элементы с индексами  $k, 2k, 2k + 1$  упорядочиваются так, чтобы больший из них находился в  $k$ -ой ячейке массива. Пирамида постепенно расширяется влево путём добавления новых элементов, переставляемых на легко вычислимую надлежащую позицию. Процесс формирования пирамиды из  $n$  элементов  $h_1 \dots h_n$  на месте описывается так:

```
L := (n div 2) + 1;
while L > 1 do begin
    L := L - 1;
    sift (L, n);
end;
```

В этом коде  $L$  присваивается значение  $n \text{div} 2 + 1$ , поскольку все элементы с большими индексами лежат в нижнем слое пирамиды и не имеют потомков. В первой итерации алгоритм упорядочивает микропирамиду, состоящую из  $L$ -того,  $2L$ -того и  $2L + 1$ -ого элементов. Этот процесс продолжается до исчерпания таких пирамид. В результате первой итерации получим  $n/4$  трехэлементных двухэтажных пирамид. В результате дальнейшей деятельности возникнут семиэлементные трехэтажные пирамиды (*семирамиды?*), в которых произойдут, если необходимо, одно- и двузвенные цепочки обменов. После обработки  $n/8$  пирамид из 7 элементов, будут получены пирамиды из 15, 31 и т. д. Этот процесс завершится, поскольку в массиве содержится конечное число элементов. Кроме того, этот алгоритм действительно строит пирамиду: в силу описанного алгоритма вставки в пирамиду и того факта, что один элемент тоже является пирамидой, по индукции следует, что описанная трехэлементная конструкция также является пирамидой. Индукция осуществляется по высоте пирамиды: при помощи вышеописанного алгоритма вставки из одного элемента и двух пирамид высоты  $k$  (каждая из которых содержит не более  $2^k - 1$  элементов, поскольку некоторые из пирамид могут быть неполными!) можно создать пирамиду высоты  $k + 1$  (которая содержит не более  $2^{k+1} - 1$  элементов).

Процесс построения пирамиды описанным алгоритмом выглядит так (чертёж отделяет

пирамидальную часть массива и еще необработанные элементы; пирамидальная структура расположена справа от черты):

44	55	12	42		94	18	06	67
44	55	12	42	94	18	06	67	
44	55	06	42	94	18	12	67	
44	42	06	55	94	18	12	67	
06	42	12	55	94	18	44	67	

Чтобы приспособить эту пирамиду для целей сортировки, необходимо проделать  $n$  шагов выталкивания наверх наименьшего элемента, и направить полученные элементы в некую выходную очередь. Однако, поскольку при всяком выталкивании число элементов в пирамиде уменьшается на 1, то можно организовать эту очередь в том же массиве, просто поменяв местами первый и последний элементы, и повторить процесс вставки новой вершины в пирамиду, основанную на массиве длины  $n - 1$  (в этом и состоит идея Флойда, предложившего минимальную по памяти турнирную сортировку). В результате наверх придет второй по величине элемент. Его следует поменять с элементом на позиции  $n - 1$  и осуществить спуск нового элемента в пирамиде из  $n - 2$  элементов и т. д. Алгоритм сортировки готовой пирамиды выглядит так:

```
R := n;
while R > 1 do begin
    x := a[1];
    a[1] := a[R];
    a[R] := x;
    R := R - 1;
    sift (1, R);
end;
```

Примененный к готовой пирамиде алгоритм сортировки работает так:

06	42	12	55	94	18	44	67	
12	42	18	55	94	67	44	06	
18	42	44	55	94	67	12	06	
42	55	44	67	94	18	12	06	
44	55	94	67	42	18	12	06	
55	67	94	44	42	18	12	06	
67	94	55	44	42	18	12	06	
94	67	55	44	42	18	12	06	

Поскольку этот алгоритм на последнюю позицию помещает минимальный элемент, на предпоследнюю — второй по величине и т. д., а на первой позиции оказывается максимальный из элементов, то элементы оказываются упорядоченными по убыванию. Этот недостаток легко исправить, поменяв отношение порядка в процедуре *sift()* на противоположное. «Правильная» функция *Heapsort()* выглядит так:

```
procedure HeapSort(var a : seq);
{ Сортировка HeapSort – просеиванием }
var L, R : index;
```

```

x : item;

procedure Sift(L, R : index);
var i, j : index;
    x, z : item;
begin
    i := L;
    j := 2 * L;
    x := a[L];
    if (j < R) and (a[j] < a[j + 1]) then
        j := j + 1;
    while (j <= R) and (x < a[j]) do begin
        z := a[i];
        a[i] := a[j];
        a[j] := z;
        i := j;
        j := 2 * j;
        if (j < R) and (a[j] < a[j + 1]) then
            j := j + 1;
    end;
end; {Sift}

begin {HeapSort}
    L := (n div 2) + 1;
    while L > 1 do begin
        L := L - 1;
        sift (L, n);
    end;
    R := n;
    while R > 1 do begin
        x := a[1];
        a[1] := a[R];
        a[R] := x;
        R := R - 1;
        sift (1, R);
    end;
end; {HeapSort}

```

Проанализируем пирамидальную сортировку. В худшем случае нужно  $n/2$  перемещений элементов на логарифмические расстояния  $\log_2 n/2, \log_2(n/2 - 1), \dots$ . Следовательно, фаза сортировки потребует  $n - 1$  вставку в пирамиду с самое большее  $\log_2(n - 1), \log_2(n - 2), \dots, 1$  перемещениями. Кроме того, нужно еще  $n - 1$  перемещение для вставки в пирамиду после извлечения ее верхушки. Каждая вставка занимает не более  $\log_2 n$  шагов, что дает общую оценку  $n \log n$ . Для больших  $n$  это даже лучше сортировки Шелла с ее  $n^{1+\delta}$ . Наилучшая производительность пирамидальной сортировки достигается при обратной упорядоченности исходных элементов. При этом фаза порождения пирамиды вообще не

требует перемещений. Среднее число перемещений равно  $\frac{n \log n}{2}$ , причем отклонения от этого значения невелики.

В 1981 году проф. Э. В. Дейкстра предложил гладкий вариант пирамидальной сортировки, дающий линейное время для почти отсортированного массива. Однако эта сортировка не получила широкого распространения из-за чересчур сложного алгоритма.

#### 6.2.2.6 Гладкая сортировка

В 1981 году Э. В. Дейкстра предложил алгоритм гладкой сортировки, имеющий время работы порядка  $n \log n$  в наихудшем случае и порядка  $n$  в случае почти отсортированной последовательности (название алгоритма указывает на то, что время его работы плавно изменяется от  $O(n \log n)$  до  $O(n)$  в зависимости от того, насколько хорошо отсортирована исходная последовательность).

Предположим, что сортируемые элементы хранятся в массиве с индексами  $0, \dots, n - 1$ . Подобно пирамидальной, плавная сортировка обрабатывает массив в два прохода. Результатом первого прохода является построение из элементов массива нескольких частично упорядоченных деревьев (пирамид). При втором проходе элементы, начиная с наибольшего, занимают предназначенные им места.

Введем в рассмотрение последовательность чисел Леонардо  $L_k: 1, 1, , 3, 5, 9, 15, \dots$  Здесь  $L_0 = L_1 = 1, L_{k+2} = L_{k+1} + L_k + 1$ . Любое натуральное число может быть представлено в леонардовской системе счисления:  $\overline{p_k p_{k-1} \dots p_1 p_0} = \sum_{i=0}^k p_i L_i, p_i \in \{0, 1\}$ , причем две единицы могут идти подряд только в том случае, если справа от них нет других единиц. Доказательство Э. В. Дейкстра оставляется читателю в качестве упражнения.

Нам будет удобно хранить леонардово представление натурального числа  $m$  в виде пары  $(p, L)$ , где  $p = \overline{p_k p_{k-1} \dots p_1 p_0}, L = (L_n, L_{n-1}), m = \sum_{i=0}^k p_i L_{n+i}$ , т. е.  $p$  является натуральным числом, двоичные цифры которого указывают, входит ли соответствующее число Леонардо в представление  $m$ . Хранение одновременно  $L_n$  и  $L_{n-1}$  позволяет последовательно вычислять  $L_{n+1}, L_{n+2}, \dots, L_{n+k}$ . Функции *up* и *down* реализуют переход к следующей и предыдущей паре:

```

typedef struct {
    long cur, prev;
} lpair;

void up(lpair *L)
{
    long tmp = L->cur;
    L->cur = L->cur + L->prev + 1;
    L->prev = tmp;
};

void down(lpair *L)
{
    long tmp = L->prev;
    L->prev = L->cur - L->prev - 1;
}

```

```

L->cur = tmp;
}

```

Следующие два фрагмента кода строят по леонардову представлению  $m$  представления чисел  $m + 1$  и  $m - 1$ :

```

{ Увеличение  $m$ }

if ((p & 3) == 3) {
    p = (p >> 2) + 1;;
    lup(&L), lup(&L);
}
else {
    ldown(&L);
    p <<= 1;
    while (L.cur > 1) {
        p <<= 1;
        ldown(&L);
    }
    ++p;
}
{ Уменьшение  $m$ }

if (L.cur == 1) {
    --p;
    while ((p & 1) == 0)
        p >>= 1, lup(&L);
}
else {
    p = (p << 2) - 1;
    ldown(&L), ldown(&L);
}

```

*Примечание.* Одно и то же число может быть представлено разными тройками. Так, преобразование  $p <<= 1$ ;  $down(\&L)$  приводит к другому представлению того же самого числа. В алгоритме используется представление с нечетным  $p$ .

В дальнейшем будем называть отрезком подпоследовательность идущих подряд элементов, длина которой является числом Леонардо, а стандартным разбиением последовательности длины  $m$  — такое ее разбиение на отрезки, что первый отрезок имеет максимально возможную длину  $l$ , а за ним следует стандартное разбиение последовательности длины  $m - l$ .

Всякий отрезок можно рассматривать как дерево. Если обозначить отрезок длины  $L_k$  через  $\langle L_k \rangle$ , то

$$\langle L_k \rangle = \langle L_{k-1} \rangle \langle L_{k-2} \rangle \langle R \rangle,$$

где  $\langle R \rangle$  — корень дерева. Сыновьями корня считаются корни отрезков  $\langle L_{k-1} \rangle$  и  $\langle L_{k-2} \rangle$ . Отрезки  $\langle L_1 \rangle$  и  $\langle L_0 \rangle$  состоят из корня без сыновей.

Введем понятия допустимых и надежных отрезков. Отрезок длины 1 является допустимым и надежным. Отрезок  $\langle L_n \rangle$  является допустимым, если  $\langle L_{n-1} \rangle$  и  $\langle L_{n-2} \rangle$

являются надежными; отрезок  $< L_n >$  является надежным, если он допустимый и значение корня не меньше значений его сыновей.

Далее под  $i$  понимается номер текущего элемента, а под префиксом последовательности — первые  $i$  ее элементов. Потребуем, чтобы при первом проходе после обработки  $i$ -го элемента выполнялись условия:

$P_1$ : самый правый отрезок стандартного разбиения префикса — допустимый, все остальные — надежные.

$P_2$ : значения корней надежных отрезков стандартного разбиения префикса, которые также являются отрезками стандартного разбиения всей последовательности, не убывают слева направо.

При втором проходе потребуем большего:

$P'_1$ : все отрезки стандартного разбиения префикса — надежные.

$P'_2$ : значения корней отрезков стандартного разбиения префикса не убывают слева направо.

Выполнение условия  $P'_1 \& P'_2$  гарантирует, что  $i$ -й элемент является максимальным среди элементов префикса и, следовательно, стоит на своем месте. Однако после уменьшения  $i$  это условие может нарушиться, и потребуются дополнительные перестановки, чтобы вновь ему удовлетворить.

Теперь мы готовы к тому, чтобы в деталях рассмотреть оба прохода алгоритма.

Первый проход. Если оба младших бита  $p$  установлены ( $p \& 3 == 3$ ), то после обработки нового элемента из него и двух последних отрезков образуется новый отрезок. Следовательно, последний отрезок из допустимого должен стать надежным. Для этого к его корню применяется функция *sift* из алгоритма пирамидальной сортировки: если наибольший сын *child* корня *top* превосходит его, то они меняются местами и *sift* применяется к позиции, где раньше стоял *child*; в листе *sift* ничего не делает. Если же  $p \& 3 == 1$ , то новый элемент сам станет отрезком длины 1, поэтому последний отрезок также должен быть сделан надежным. Если  $i + L.prev < n$ , то достаточно применения процедуры *sift*, а иначе отрезок длины  $L.cur$  входит в стандартное разбиение всей последовательности, поэтому применения *sift* недостаточно для выполнения  $P_2$ . Вместо нее используется аналогичная процедура *trinkle*, которая тоже выполняет просеивание, но работает как будто с троичным деревом — третьим потомком корня считается корень предыдущего отрезка.

После первого прохода выполняется условие  $P_1 \& P_2$ . Чтобы удовлетворить более сильному условию  $P'_1 \& P'_2$ , применяется *trinkle* к последнему элементу.

Второй проход. Если  $L.cur == 1$ , то никаких дополнительный мер принимать не надо. В противном случае после исключения корня последний отрезок распадается на два надежных. Чтобы удовлетворить условию  $P'_2$ , к корням этих отрезков применяется процедура *semitrinkle* (вариант *trinkle*, учитывающий тот факт, что отрезок является надежным).

Вот код процедуры гладкой сортировки:

```
typedef int elem;
typedef unsigned long ulong;

typedef struct {
    long cur, prev;
} lpair;
```

```

void up(lpair *L)
{
    long tmp = L->cur;
    L->cur = L->cur + L->prev + 1;
    L->prev = tmp;
}

void down(lpair *L)
{
    long tmp = L->prev;
    L->prev = L->cur - L->prev - 1;
    L->cur = tmp;
}

/*
// нормализация представления размера пирамиды в леонардовой системе счисления
//
// (в нормализованном представлении младший бит должен быть установлен) ||
// p - леонардово представление (возможно, ненормализованное) ||
// L.cur - число леонардо, соответствующее младшему биту
*/
void normalize(ulong *p, lpair *L)
{
    while ((*p & 1) == 0)
        up(L), *p >>= 1;
}

void swap(elem *x, elem *y)
{
    elem tmp = *x;
    *x = *y;
    *y = tmp;
}

/*
// просеивание ||
// top указатель на вершину, L.cur - размер пирамиды */
void sift(elem *top, lpair L)
{
    while (L.cur > 1) {
        elem *child = top - (L.cur - L.prev); /* // вершина первой дочерней пирамиды */
        down(&L); /* // L.cur --- размер первой дочерней пирамиды */
        if (*child < *(top - 1)) {
            child = top - 1; /* // вершина второй дочерней пирамиды */
            down(&L); /* // L.cur --- размер второй дочерней пирамиды */
        }
    }
}

```

```

    }
    /* // child указывает на максимальную вершину дочерних пирамид */
    if (*child <= *top) /* // порядок верный — выходим */
        break;
    swap(top, child); /* // иначе восстанавливаем порядок */
    top = child;      /* // и переходим к дочерней пирамиде */
}
}

/*
// просеивание в троичной пирамиде
// третьим потомком выступает следующая по размеру пирамида
*/
void trinkle(elem *top, ulong p, lpair L)
{
    while (p > 0) {
        normalize(&p, &L);
        elem *ntop = top - L.cur; /* // указатель на вершину следующей пирамиды */
        if (p == 1 || *ntop <= *top) { /* // осталась только одна пирамида или
            вершины упорядочены */
            sift (top, L);
            break;
        }
        /* // пирамид больше одной и вершины не упорядочены */
        --p;
        if (L.cur == 1) { /* // пирамида из одного элемента --- выполняем обмен,
            переходим к следующей */
            swap(ntop, top);
            top = ntop;
        }
        else {
            /* // в пирамиде больше одного элемента — выполняем троичное просеивание */

            /* // третьим потомком считаем вершину следующей по размеру пирамиды
            — ntop */
            elem *child = ntop + L.prev; /* // указатель на вершину первой дочерней
                пирамиды */
            lpair Lc = L; /* // временная пара для отслеживания размера дочерних
                пирамид */
            down(&Lc); /* // Lc.cur — размер первой дочерней пирамиды */
            if (*child < *(top - 1)) { /* // сравниваем вершины дочерних пирамид */
                child = top - 1; /* // переходим к вершине второй дочерней пирамиды */
                down(&Lc); /* // Lc.cur — размер второй дочерней пирамиды */
            }
            /* // child указывает на максимальную из вершин дочерних пирамид */
            if (*ntop > *child) { /* // вершина следующей пирамиды больше */

```

```

        swap(ntop, top);
        top = ntop;
    }
    else {
        /* // вершины текущей и следующей пирамид упорядочены --- ||
         // выполняем обычное просеивание */
        swap(child, top);
        sift (child , Lc);
        break;
    }
}

/*
// троичное полупросеивание ||
// по указателю top корректная пирамида размером L.cur
*/
void semitrinkle(elem *top, ulong p, lpair L)
{
    elem *ntop = top - L.cur;
    if (p > 1 && *ntop > *top) { /* // вершины соседних пирамид неупорядочены */
        swap(ntop, top);
        /* // после обмена порядок элементов в следующей пирамиде нарушен, поэтому ||
         // для нее выполняем полное троичное просеивание */
        trinkle(ntop, p - 1, L);
    }
}

void smoothsort(elem *m, long n)
{
    long i;
    ulong p = 1;
    lpair L;
    /* // смысл манипуляций p и L подробно объясняются в лекции */
    L.cur = 1;
    L.prev = 1;
    /* // прямой проход --- построение пирамид */
    for (i = 1; i < n; ++i) {
        if ((p & 3) == 3) {
            sift (m + i - 1, L);
            p = (p >> 2) + 1;
            up(&L), up(&L);
        }
        else {

```

```

if (i + L.prev < n) /* // если последняя пирамида будет обединена в
    большую,
    sift (m + i - 1, L); // не нужно следить за упорядоченностью вершин */
else
    trinkle (m + i - 1, p, L); /* // в общем же случае необходимо троичное
        просеивание */
    down(&L), p <<= 1;
    while (L.cur > 1)
        down(&L), p <<= 1;
        ++p;
    }
}
trinkle (m + n - 1, p, L); /* // перед обратным проходом последнее троичное
    просеивание */
while (i > 1) { /* // обратный проход – сортировка */
    --i;
    if (L.cur == 1) {
        --p;
        normalize(&p, &L);
    }
    else { /* // удаление элементы приводит к разбиению пирамиды
        на две */
        --p;
        down(&L), p = (p << 1) + 1;
        semitrinkle (m + (i - L.prev) - 1, p, L); /* // троичное полупросеивание для
            левого потомка */
        down(&L), p = (p << 1) + 1;
        semitrinkle (m + i - 1, p, L); /* // ... и для правого */
    }
}
}

```

Попробуем оценить время работы плавной сортировки. Для чисел Леонардо существует формула

$$L_k = \frac{\sqrt{5} + 1}{\sqrt{5}} \left( \frac{1 + \sqrt{5}}{2} \right)^k + \frac{\sqrt{5} - 1}{2} \left( \frac{1 - \sqrt{5}}{2} \right)^k - 1$$

(эта последовательность имеет номер A001595 в онлайн-справочнике целочисленных последовательностей <http://www.research.att.com/njas/sequences/Seis.html>). Следовательно, количество чисел Леонардо, не превосходящих  $n$ , есть  $O(\log n)$ , т.е. стандартное разбиение длины  $n$  состоит из  $O(\log n)$  отрезков, каждый из которых является деревом высоты  $O(\log n)$ . Значит, каждая из процедур *Sift*, *Trinkle*, *Semitrinkle* выполняется за время  $O(\log n)$ , а так как эти функции вызываются из *smoothsort* фиксированное число раз на каждом шаге, общее время их работы будет  $O(n \log n)$ . Осталось проанализировать вложенные циклы в процедуре *smoothsort*. В первом из них число вызовов процедуры *down* ограничивается числом предшествовавших вызовов процедуры *Up*, а она, в свою очередь, вызывается не более  $2n$  раз. Рассуждения для второго цикла аналогичны. Итак,

в наихудшем случае плавная сортировка работает за время  $O(n \log n)$ .

Если последовательность изначально упорядочена, элементы вообще не переставляются.

#### 6.2.2.7 Быстрая сортировка

Займемся теперь улучшением обменной сортировки — наихудшей из простых. Забегая вперед, скажем, что она будет кардинально улучшена, и оправдает полученное в свое время от ее автора Чарльза Энтони Ричарда Хоара многообещающее название *быстрая сортировка*.

Мы знаем, что быстрыми являются те сортировки, в которых осуществляются перестановки на большие расстояния. Отвлечемся от обычного для фон Неймановской машины исполнения вслепую, не глядя на данные, и отсортируем  $n$  элементов, уже упорядоченных в обратном порядке. Меняя местами первый с последним всего за 3 операции, тогда как черепашьим пузырьковым одношаговым движением это же расстояние будет пройдено за  $6n - 9 \gg 3$  операций! Далее процесс повторяется для 2 и  $n - 1$  и т. п. пар, равноотстоящих от середины элементов. Этот пример, вероятность которого очень мала и равна  $\frac{1}{n!}$ , сортируется за линейное время. Видимо, подобная идея дальних симметричных обменов навела автора на следующий алгоритм сортировки: выберем наугад какой-нибудь промежуточный (*разделяющий*) элемент  $x$  (*зерно*) и будем просматривать сортируемый массив слева, пока не обнаружим элемент  $a_i > x$ , затем, проходя этот же массив справа, будем искать элемент  $a_j < x$ . Чтобы ликвидировать обнаруженную инверсию, поменяем местами эти два элемента (они в силу поиска с концов к середине находятся на довольно большом расстоянии) и продолжим процесс просмотра и перестановки, пока оба прохода не встретятся где-то в середине массива. Этот процесс классифицирует сортируемые элементы относительно *зерна сортировки*  $x$ : в нужной ли половине массива они находятся. (Здесь уже можно начать подозревать, что ненужные половины будут впоследствии отброшены как это уже имело место в деревьях поиска или при дихотомии в упорядоченных таблицах.) В результате массив окажется разбитым на левую часть, с ключами, меньшими (или равными)  $x$ , и правую, с ключами большими (или равными)  $x$ . Этот процесс представим в виде процедуры, причем отношения  $<$  и  $>$  заменяются на  $\leq$  и  $\geq$ , а в заголовке цикла **while** используется их отрицания  $>$  и  $<$  соответственно.

```
procedure partition;
var w, x : item;
begin
    i := 1;
    j := n;
    { Взять промежуточный элемент x (зерно) }
repeat
    while a[i] < x do
        i := i + 1;
    while x < a[j] do
        j := j - 1;
    if (i <= j) then begin
        w := a[i];
        a[i] := a[j];
        a[j] := w;
    end;
end;
```

```

a[j] := w;
i := i + 1;
j := j - 1;
end;
until i > j;
end;

```

При этом зерновой элемент  $x$  выступает в качестве барьера для обоих просмотров. Если в последовательности 44 55 12 42 94 06 18 67 взять зерновой элемент  $x = 42$ , то при ее разделении на подпоследовательности произойдут два обмена:  $18 \leftrightarrow 44$  и  $06 \leftrightarrow 55$ , последний из которых меняет третий элемент с пятым. Ключи  $a_1, \dots, a_{i-1}$  не больше ключа  $x = 42$ . Ключи  $a_{j+1}, \dots, a_n$  не меньше  $x$ . Следовательно, массив разделен на две части:

$$\begin{aligned} \forall k: 1 \leq k < i: a_k \leq x, \\ \forall k: j < k \leq n: x \leq a_k. \end{aligned}$$

Этот алгоритм прост и эффективен в памяти прямого доступа, однако, в случае идентичных ключей разделение все равно потребует  $n/2$  обменов. Этих необязательных обменов можно избежать, если добавить равенство в условия просматривающих циклов.

```

while(a[i] <= x) do
    i := i + 1;
while(x <= a[j]) do
    j := j - 1;

```

При этом мы вынуждены будем отказаться от барьерно-элементного способа организации циклов ради маловероятного случая. Чтобы разделение упорядочивало, надо продолжить его автономное применение к каждой из получившихся частей, пока разбиение не приведет к одноэлементным частям. Естественной реализацией этого алгоритма является рекурсивная процедура *QuickSort()*

```

procedure QuickSort(var a : seq);
{ Быстрая сортировка Хоара }
{ Основной принцип: производим разделение массива в границах [L..R] так, чтобы
левая половина содержала элементы, меньшие некоторого x, правая —
большие }

procedure RecSort(L, R : index);
var i, j : index;
x, w : item;
begin
    x := a[(L + R) div 2]; { выбираем зерно — средний элемент }
    i := L;
    j := R;
{ Разделение массива }
repeat
{ Два совмещенных просмотра во встречных направлениях (i++, j-- !!!) }
    while a[i] < x do

```

```

    i := i + 1;
  while a[j] > x do
    j := j - 1;
  if i <= j then begin { Обмен элементов, находящихся не в своих
    половинах }
    w := a[i];
    a[i] := a[j];
    a[j] := w;
    i := i + 1;
    j := j - 1;
  end;
  until i > j;
  if L < j then
    RecSort(L, j); { Сортировка левой половины перепоручается вновь
      активируемому экземпляру этой же процедуры }
  if i < R then
    RecSort(i, R); { Сортировка правой половины перепоручается вновь
      активируемому экземпляру этой же процедуры }
  end; { RecSort }

begin {Sort}
  RecSort(1, n);
end;

```

В наше многопроцессорное и многоядерное время нельзя не заметить, что сортировка Хоара поддается распараллеливанию на  $\lfloor \log n \rfloor$  процессорах. (Поэтому Седжвик в своей книге [88] называет быструю сортировку алгоритмом «разделяй и властвуй».)

Мы не будем избегать рекурсии по техническим причинам, как это делалось 30 и более лет назад, и напишем еще и нерекурсивную версию как альтернативный вариант быстрой сортировки. Суть итеративного решения заключается во введении списка отложенных заданий на дальнейшее разделение. Поскольку на каждом этапе возникает две задачи по разделению, одну из них можно выполнить сразу, а другую — отложить, поместив в упомянутый список. При этом существенно, что отложенные запросы выполняются в обратном порядке: второй полумассив главного массива будет отсортирован последним. Обратный порядок вызван тем, что сортировку второй половины необходимо выполнять одновременно с первой, в соответствии с моделируемой нами логикой рекурсивной программы *QuickSort()*. Со стеком и обратным порядком выполнения подзаданий обхода мы уже встречались при нерекурсивном обходе дерева. И здесь этот порядок предопределен семантикой рекурсивного выполнения быстрой сортировки, характерным элементом которой является рекурсивное разветвление в два места. Задача упрощается еще и тем, что в стек заносятся не сами сортируемые подпоследовательности, а их границы в общем массиве.

Заметим, что экспериментальное сравнение рекурсивной и нерекурсивной версий обнаруживает их одинаковую временную сложность, и сегодня единственной причиной уклонения от рекурсии может быть только использование языка программирования, не обеспечивающего рекурсивного вызова процедур (старые версии Фортрана и Бейсика и ассемблер, на котором все придется моделировать вручную, даже имея аппаратную

поддержку стека).

```
procedure QuickSortNR(var a : seq);
{ Быстрая сортировка Хоара — нерекурсивная }
{ Основной принцип: список границ требуемых разделений инвертируется в стеке
}

type T = record { Элемент стека — граничная пара }
    L, R : index
end;

procedure Create(var s : Stack);
function Empty(var s : Stack) : boolean;
function Size(var s : Stack) : integer;
function Push(var s : Stack; t : T) : boolean;
function Pop(var s : Stack) : boolean;
function Top(var s : Stack) : T;
procedure Destroy(var s : Stack);

var st : Stack; { of T }
    i, j, L, R : index;
    x, w : item;
    t : T;
begin
    Create(st);
    t.L := 1;
    t.R := n;
    Push(st, t);
    repeat {Выбор запроса из стека}
        L := Top(st).L;
        R := Top(st).R;
        Pop(st);
        repeat {разделение A[L]..A[R]}
            x := a[(L + R) div 2]; { выбираем зерновой элемент }
            i := L;
            j := R;
            repeat {разделение массива}
                while a[i] < x do
                    i := i + 1;
                while a[j] > x do
                    j := j - 1;
                if i <= j then begin
                    w := a[i];
                    a[i] := a[j];
                    a[j] := w;
                    i := i + 1;
                    j := j - 1;
                end;
            end;
        end;
    end;
end;
```

```

    end;
until i > j;
if i < R then begin { откладываем запрос на сортировку правой половины
    в стек a la RecSort(i, R) }
    t.L := i;
t.R := R
    Push(st, t);
end;
{ Вторую половину сортируем немедленно RecSort(L, j); }
R := j; { L и R теперь ограничивают левую часть }
until L >= R;
until Empty(st);
end;

```

Проанализируем алгоритм быстрой сортировки. Сначала оценим трудоемкость процесса разделения. Выбрав некоторое зерновое значение  $x$  мы проходим по всему массиву и выполняем ровно  $n$  сравнений. Число обменов можно определить из следующих вероятностных соображений: при зафиксированном зерновом элементе  $x$  ожидаемое число обменов равно числу элементов в левой части разделяемой последовательности, т. е.  $n - 1$ , умноженному на вероятность того, что при обмене каждый такой элемент попадает в нужную половину на свое место. Для левой части обмен происходит, если элемент перед этим находился в правой части. Вероятность этого равна  $\frac{n - (x - 1)}{n}$ . Получить ожидаемое число обменов можно, усреднив эти ожидаемые значения для всевозможных границ  $x$ :

$$M = \frac{1}{n} \sum_{x=1}^n (x - 1) \frac{n - (x - 1)}{n} = \frac{1}{n^2} \sum_{u=0}^{n-1} u(n - u) = \frac{n(n - 1)}{2n} - \frac{2n^2 - 3n + 1}{6n} = \frac{n - 1/n}{6}.$$

Наилучшим образом быстрая сортировка работает тогда, когда всякий раз в качестве зернового элемента выбирается медиана — среднезначный элемент множества, делящий его на два равномощных подмножества (с точностью до 1 элемента). В этом случае каждый процесс разделения расщепляет массив на две половины и для сортировки требуется всего  $\log_2 n$  проходов. В результате общее число сравнений равно  $n \log_2 n$ , а общее число обменов —  $\frac{n \log_2 n}{6}$ . Несмотря на то, что вероятность выбора медианы только  $1/n$ , производительность быстрой сортировки при случайном выборе зернового элемента ухудшается лишь в постоянное число раз  $2 \ln 2$ .

Недостатками быстрой сортировки являются низкая производительность при небольших  $n$ , но этим грешат все усовершенствованные методы [54]. Для обработки случаев малых  $n$  в усовершенствованные сортировки включают один из простых методов. Проще всего это делать для рекурсивной сортировки Хоара. Небольшое, но эффективное усовершенствование возможно при выборе зернового элемента как среднего из трех случайно взятых. Наконец, в случае явного дисбаланса величин элементов к сортировке большей половины применяют пирамидальный алгоритм. И именно такая комбинированная стратегия используется в алгоритме *sort()* библиотеки STL.

Интересно, что для поиска медианы существует эффективный алгоритм, очень похожий на быструю сортировку и, как ни странно, двоичный поиск. На первом этапе

выбирается зерновой элемент и происходит разделение относительно него. Местом медианы в отсортированной последовательности будет  $n/2$ . Если зерно стоит на позиции  $k < n/2$ , то медиана находится справа от него, в противном случае — слева. Применив разделение к большей части, либо найдем медиану, либо получим следующий отрезок для ее рекурсивного поиска. Цена этого процесса линейная:

$$n + \frac{n}{2} + \frac{n}{4} + \dots + 1 = 2n.$$

Однако в самой неблагоприятной ситуации множество кандидатов уменьшается только на 1, а число сравнений оценивается  $O(n^2)$ . Впрочем, применив вышеописанные способы выбора разделяющего элемента, можно свести к нулю вероятность возникновения такой ситуации.

Еще одно усовершенствование быстрой сортировки связано с оценкой размера стека для хранения границ отложенных участков. В самом неблагоприятном случае, когда в стек заносятся одноэлементные участки, его размер оценивается в  $n$  элементов. Если же заносить в стек «более длинную часть» и продолжать разделение более короткой части, размер стека может быть ограничен  $\log_2 n$ . Тогда в программу нерекурсивной сортировки Хоара должны быть внесены следующие изменения:

```

if j – L < R – i then begin
    if i < R then begin { занесение в стек запроса на сортировку правой части }
        t.L := i;
        t.R := R;
        Push(st, t);
    end;
    R := j; { Продолжение сортировки левой части }
end
else begin
    if L < j then begin { занесение в стек запроса на сортировку левой части }
        t.L := L;
        t.R := j;
        Push(st, t);
    end;
    L := i; { Продолжение сортировки правой части }
end;

```

### 6.2.3 Сравнение методов внутренней сортировки

Теперь, когда нами изучены все методы внутренней сортировки, можно произвести их тотальное сравнение. Во-первых, прямые методы сортировки имеют точные оценки [54].

		Минимум	Среднее	Максимум
Вставка	Сравнений ( $C$ )	$n - 1$	$\frac{n^2 + n - 2}{4}$	$\frac{n^2 - n}{2} - 1$
	Присваиваний ( $M$ )	$2(n - 1)$	$\frac{n^2 - 9n - 10}{4}$	$\frac{n^2 - 3n - 4}{2}$

Выборка	Сравнений	$\frac{n^2 - n}{2}$	$\frac{n^2 - n}{2}$	$\frac{n^2 - n}{2}$
		Присваиваний	$3(n - 1)$	$n(\ln n + 0,57)$
Пузырьковая	Сравнений	$\frac{n^2 - n}{2}$	$\frac{n^2 - n}{2}$	$\frac{n^2 - n}{2}$
	Присваиваний	0	$\frac{3}{4}(n^2 - n)$	$\frac{3}{2}(n^2 - n)$

Усовершенствованные методы не имеют сколько-нибудь простых и точных формул сложностной оценки. Для сортировки Шелла затраты составляют  $O(n^{1+\delta})$ , а для пирамидальной и быстрой —  $O(n \log n)$ . Эти оценки позволяют разбить все сортировки на примитивные, прямые, квадратичные методы со сложностью  $O(n^2)$  и улучшенные или линеарифмические  $O(n \log n)$ . В реальных случаях на коэффициенты этих оценок могут влиять архитектурные факторы (ускорение пересылок и/или сравнений аппаратными средствами либо оптимизирующими компиляторами) и форма и местонахождение (резидентные, ...) ключей (простые или составные).

Прежде, чем давать сравнительную эмпирическую оценку всех сортировок, напомним соответствующие порядки роста величин-степеней 2.

$n$	$\log n$	$n \log n$	$n^{5/4}$	$n^2$
1	0	0	1	1
16	4	64	32	256
256	8	2,048	1,024	65,536
4,096	12	49,152	32,768	16,777,216
65,536	16	1,048,565	1,028,576	4,294,967,296
1,048,576	20	20,969,520	33,554,432	1,099,511,627,776
16,777,216	24	402,653,184	1,073,741,824	281,474,976,710,656
Линейный поиск	Дихотомия	Быстрые сортировки	Сортировка Шелла	Простые сортировки
	Двоичное дерево	(Дейкстры, Флойд, Хоар)		

Поскольку получить точные оценки для всех сортировок затруднительно, представляют интерес результаты вычислительного эксперимента, проведенного Н. Виртом на микроЭВМ Lilith с микропрограммной поддержкой языка Модула 2 (М-код) для массива данных из 2048 элементов со *скалярным* ключом:

	Упорядоченный	Случайный	Обратный порядок
Вставка	0,22	50,74	103,80
Двоичная вставка	1,16	37,66	76,06
Выборка	58,18	58,34	73,46
Пузырьковая	80,18	128,84	178,66
Шейкерная	0,16	104,44	187,36
Шелла	0,80	7,08	12,34
Пирамидальная	2,32	2,22	2,12
Гладкая	0,22?	2,22?	2,12?
Быстрая	0,72	1,22	0,76
Быстрая нерекурс.	0,72	1,32	0,80
Слияние	1,98	2,06	1,98

Времена, приведенные в этой таблице, во-первых, свидетельствуют о колоссальной разнице квадратичных и линеарифмических методов уже на массиве из 2048 элементов (а это всего выпускник крупного вуза!). Во-вторых,

- двоичная вставка по сравнению с простой почти ничего не дает, а в случае упорядоченного массива даже получается отрицательный эффект;
- пузырьковая сортировка определенно наихудшая из всех сравниваемых, ее усовершенствованная шейкерная версия все равно проигрывает почти всем методам;
- быстрая сортировка в 2–3 раза лучше пирамидальной, но уступает ее гладкой разновидности на почти упорядоченных последовательностях.

Гладкая сортировка TODO2005.

Сортировка Лукашевича (1984). Для упорядочения строк небольшой длины они отображаются на вещественный тип, после чего сортировка ускоряется за счёт гораздо более быстрого сравнения ключей. Незадача: иногда новые ключи склеиваются (на 32-битном real). Но всё равно идеи Рабина-Карпа живут и побеждают!

Цифровая(разрядная) сортировка [].

#### 6.2.4 Внешние сортировки

Рассмотренные нами алгоритмы внутренней сортировки существенно использовали прямой доступ к элементами массива и возможность их быстрой перестановки за постоянное время  $O(1)$ . Аппаратная поддержка прямого доступа обеспечивается устройствами основной (оперативной) памяти ЭВМ. Несмотря на ее гигабайтные сегодняшние размеры, эта память остается дефицитной, поскольку запросы и потребности практических задач растут опережающими темпами. Многие задачи сортировки работают с настолько большими массивами данных, что никакая существенная часть их не помещается в оперативной памяти. Среди устройств внешней памяти существуют магнитные диски, также называемые устройствами *прямого (произвольного) доступа*, но анализ их конструкции свидетельствует о существенной зависимости времени доступа от местонахождения искомого элемента данных. Мгновенно (электронно) осуществляется только переключение между головками одного цилиндра. Остальные движения: поступательное перемещение с цилиндра на цилиндр и вращательное движение — подвод сектора к головке — электромеханические и осуществляются хоть и за небольшое, но за *линейное время*. И именно это линейное время доминирует в оценках времени внешних сортировок на этих дисках. Алгоритмы внутренних сортировок становятся либо неприменимыми, либо должны быть модифицированы.

Итак, последовательный характер доступа так или иначе доминирует на всех внешних устройствах и налагает весьма сильное ограничение, вынуждающее нас использовать другие методы сортировки. Характерным методом внешней сортировки является *метод слияния*. Под слиянием понимается подобное слиянию двух рек объединение двух или более входных последовательностей в одну-единственную упорядоченную выходную последовательность с помощью повторяющегося выбора из доступных в данный момент резидентных (буферных) элементов. Слияние — очень простая и быстрая операция, которую можно приспособить для задач внешней сортировки.

Начнем рассмотрение слияния с частного случая слияния *упорядоченных последовательностей*. Сначала надо сравнить резидентные буферные переменные этих последовательностей ( $\text{Top}(s_i)$  или  $s_i \uparrow$ ) и запомнить из какой входной последовательности поступило минимальное значение. Далее необходимо извлечь (прочитать и удалить из буфера!) соответствующую компоненту этой последовательности и поместить ее в выходную последовательность. Повторяя этот процесс до окончания входных потоков данных, за линейное время ( $2n$  операций) мы получим отсортированную последовательность. В этом случае дополнительной памяти (рабочих лент) не требуется, а производительность сравнима с такими же отзывчивыми на входную упорядоченность внутренними сортировками как сортировка вставкой или гладкая сортировка.

Идея слияния может быть применена и к неупорядоченным входным данным, при этом их упорядоченность будет постепенно возрастать в процессе нескольких слияний подпоследовательностей исходных данных до их полной упорядоченности. Рассмотрим идею сортировки *простым слиянием*:

1. последовательность  $a$  разбивается на 2 половины  $b$  и  $c$ ;
2. части  $b$  и  $c$  *сливаются*; при этом одиночные элементы из разных частей образуют упорядоченные пары в выходной последовательности, т. е. первым в ней оказывается меньший из двух первых элементов этих частей;
3. полученная объединенная слиянием последовательность более упорядочена, чем исходная и она вновь подвергается разделению и слиянию; при этом упорядоченные пары переходят в упорядоченные четверки, те переходят в восьмерки и т. д. до полной упорядоченности.

Возьмем в качестве примера последовательность

$$\underbrace{44 \quad 55}_{\text{I}} \quad \underbrace{12 \quad 42}_{\text{II}} \quad \underbrace{94 \quad 18}_{\text{III}} \quad \underbrace{06 \quad 67}_{\text{IV}}$$

После разбиения на две части получим

$$\begin{array}{cccc} 44 & 55 & 12 & 42 \\ 94 & 18 & 06 & 67 \end{array}$$

Слияние одиночных компонент (т. е. упорядоченных последовательностей длины 1) в упорядоченные пары дает:

$$\underbrace{\begin{array}{ccccc} 44 & = & 94' & 18 & \leftrightarrow & 55' \\ & & & & & \end{array}}_{\text{I}'} \quad \underbrace{\begin{array}{ccccc} & & & & \\ 06 & \leftrightarrow & 12' & \quad & 42 \quad = \\ & & & & \end{array}}_{\text{II}'}$$

Деля эту последовательность пополам и сливая упорядоченные пары, получаем

$$06 \quad 12 \quad 44 \quad 94' \quad 18 \quad 42 \quad 55 \quad 67$$

Теперь осуществляется слияние четверок

$$\begin{array}{cccc} 06 & 12 & 44 & 94 \\ 18 & 42 & 55 & 67. \end{array}$$

Первый элемент первой четверки 06 меньше первого элемента второй четверки 18, поэтому он перемещается в выходную последовательность, а первым элементом первой четверки становится следующий за ним 12. Он выигрывает сравнение с 18 и также проходит на выход. Рассматривается очередной элемент первой четверки (уже пары!) 44. Он, наконец-то, проигрывает 18 и пропускает это значение вперед, после чего и для сравнения выбирается новый элемент второй четверки (42). Этот элемент меньше 44, поэтому перемещается в выходную последовательность, выбирая для сравнения следующий элемент этой же последовательности 55. Теперь продвигается вперед 44, а на его место приходит последний элемент первой четверки 94. Ему, как самому большому, придется уступить сначала 55, а потом 67:

Итак, третье разделение и слияние привели, наконец, к желаемому результату.

Терминология сортировки слиянием следующая. Однократное слияние или разделение всего множества называется *фазой*. *Проходом* или *этапом* называется пара фаз разделение-слияние. Сортировка приведенного примера происходит за три двухфазных прохода, каждый из которых состоит из фазы разделения и фазы слияния. Поскольку для такой сортировки нужны три магнитофона (три ленты), то она получила название «трехленточного слияния». Заметим, что фазы разделения носят вспомогательный характер и не улучшают упорядоченности элементов. Поскольку при разделении элементы не переставляются, то половина трудоемкости сортировки слиянием, связанная с разделением, непродуктивна и от нее следует избавиться. Для этого применяется очень простой прием: результат слияния сразу же распределяется по двум выходным лентам, которые станут входными при следующем просмотре, поменявшихся ролями с исходными лентами предыдущего этапа. Цена вопроса — еще один, четвертый, магнитофон (и лента). Таким образом, вместо двухфазной трехленточной сортировки получаем однофазную четырехленточную с вдвое меньшей трудоемкостью.

Проанализируем идею сортировки слиянием. Количество проходов для этой сортировки есть  $\lceil \log_2 n \rceil$ , поскольку при каждом проходе размер отсортированных отрезков удваивается. Общее число пересылок

$$M = n \cdot \lceil \log_2 n \rceil,$$

поскольку при каждом проходе все элементы копируются по одному разу. Число сравнений ключей  $C$  даже меньше  $M$ , т. к. при копировании остатков отрезков («сливе») сравнения не производятся. Однако поскольку сортировки слиянием происходят на внешних электромеханических устройствах, то время пересылки между устройством внешней памяти

и основной памятью на несколько порядков превышает время сравнения резидентных компонент в буферах оперативной памяти. Поэтому при анализе сортировок слиянием числом сравнений ключей обычно пренебрегают. Главным в оценке сортировки слиянием является большая пространственная сложность алгоритма — двойная память, требуемая последовательным характером слияния-разделения. За это многие программисты не вполне оправданно отказываются от ее применения для внутренней сортировки. А между прочим, сортировка слиянием, в отличие от быстрой и даже пирамidalной, изначально является устойчивой, и именно этот алгоритм предлагается в библиотеке STL под характерным названием *stable\_sort()*.

Что русскому хорошо, то немцу смерть.

Русская пословица

В оценке сортировки слиянием как внешней следуют другим традициям, считая двойное количество лент (магнитофонов) вполне допустимым. Тем более, что количество магнитофонов постоянно ( $O(1)!$ ) и совершенно не зависит от объема входных данных ( $O(n)$ ), а емкость каждого магнитофона потенциально бесконечна по сравнению с емкостью ОП.

За примерами конкретных программ прямого слияния в оперативной памяти мы отсылаем к пособию Н. Вирта [54] (Модуля 2) или к исходным файлам STL (заголовочный файл *<algorithm>*).

Еще более быстрым является *естественное слияние*. В основе этого алгоритма предположение, что в исходном файле уже могут быть упорядоченные отрезки. Для простоты положим, что сортируемая последовательность изначально расположена на ленте 1, а ленты 2, 3 и 4 свободны, и последовательность необходимо упорядочить по возрастанию. Первая часть естественного слияния заключается в разбиении файла на ленте 1 и записи его на ленты 3 и 4. Для этого из файла 1 считывается первая компонента и помещается на ленту 3. Если следующая компонента файла 1 не меньше предыдущей, то она тоже помещается на ленту 3. В противном случае ее следует записать на ленту 4, куда теперь будут помещаться все компоненты до встречи новой инверсии — двух подряд идущих элементов, не соответствующих устанавливаемому порядку. После инверсии необходимо снова начать запись отрезка на ленту 3, после еще одной инверсии — снова на 4 и т. д. вплоть до исчерпания исходного файла 1.

После такого разделения начинается операция слияния. Будем осуществлять его на ленту 1, поскольку она поступила в качестве исходных данных, она же должна содержать результат. Выполним слияние как обычно, запоминая при этом значение последней записанной на ленту 1 компоненты. Если эта последняя компонента больше, чем значения на лентах 3 и 4, то результат слияния необходимо направить на ленту 2. Как только ситуация повторится, следует вновь начать запись на ленту 1 и т. д. вплоть до исчерпания файлов 3 и 4.

Если в результате вышеописанной операции все данные окажутся в файле 1, а файл 2 окажется пуст, то сортировка завершена. Иначе следует повторить слияние с лент 1 и 2 на ленты 3 и 4, после чего проверить факт упорядоченности файла 3. В случае упорядоченности файл 3 следует переписать в файл 1 и закончить работу алгоритма. В противном случае необходимо повторить слияние с лент 3 и 4 на 1 и 2.

Отзывчивость естественного слияния на наличие упорядоченных отрезков в сортируемой последовательности — не единственное ее достоинство. Обычное слияние в случае

любого (упорядоченного, почти упорядоченного или совершенно неупорядоченного) файла выполняет строго  $n \cdot \lceil \log_2 n \rceil$ . Сортировка естественным слиянием работает так же плохо только в худшем случае обратно упорядоченного файла. В случае упорядоченного файла естественному слиянию необходимо пройти по файлу только один раз, в то время как обычному опять необходимо  $n \cdot \lceil \log_2 n \rceil$  операций. То есть обычное слияние даже в случае почти упорядоченных последовательностей трудоголически выполняет отмеренное число операций, совершенно не используя уже имеющийся частичный порядок.

Еще одна идея усовершенствования слияний — сократить число проходов сортировки, выполняя распределение и слияние отрезков более чем в два направления. Этот метод называется *многопутевым сбалансированным слиянием*. Его линеарифмическая оценка улучшается за счет повышения основания логарифма с двух до числа потоков  $k$ :

$$M = n \cdot \lceil \log_k n \rceil$$

Например, при удвоении числа магнитофонов с 4 до 8, время сортировки последовательности уменьшится в 2 раза (основное логарифмическое тождество):

$$\frac{M_2}{M_4} = \frac{n \cdot \lceil \log_2 n \rceil}{n \cdot \lceil \log_4 n \rceil} = \frac{\lceil \log_2 n \rceil \cdot \log_2 4}{\lceil \log_2 n \rceil} = 2.$$

### 6.3 Таблицы с прямым доступом

Ранее мы рассмотрели вопросы ускорения поиска с помощью упорядоченных таблиц и деревьев поиска, которые дали логарифмическое время доступа. Возможно ли дальнейшее ускорение доступа? Ведь обращение к каждому элементу данных осуществляется в конце концов по конкретному физическому адресу, и если мы сможем быстро вычислить этот адрес по ключу, то фактически мы получим прямой доступ за несколько большее, но *постоянное* время. Для этого необходимо построить отображение  $H$  ключей  $K$  в адреса  $A$  (или в индексы  $I$ ):

$$H: K \rightarrow A.$$

Поместим нашу таблицу в обычный массив и построим преобразование ключей в индексы. Напомним, что массив — это эффективная структура данных с доступом за постоянное время, и для доступа к его элементам уже применяется некоторая расстановочная функция  $A = b + (i - 1) \cdot \text{sizeof}(T)$ , где  $b$  — адрес начала массива,  $\text{sizeof}(T)$  — размер компоненты вектора в байтах, а  $i$  — индекс компоненты вектора, отсчитываемый от 1, как это принято в линейной алгебре. По соображениям эффективной аппаратной реализации массивы следует индексировать с 0, как это делается в языке Си. Для многомерного массива эта функция является многочленом, степень которого равна числу измерений массива, а коэффициенты — суть его размеры по соответствующим индексным координатам. Естественно, многочлен вычисляется по экономной схеме Горнера. Так, для матрицы  $3 \times 4$  из восьмибайтных целых функция  $H$  имеет вид:

$$H(i, j) = b + (i \cdot 4 + j) \cdot \text{sizeof}(T), \quad \text{где } i = 0 \dots 2, j = 0 \dots 3, \text{ sizeof}(T) = 8.$$

Итак, существует эффективный доступ к элементам массива за постоянное время. Применим это подход для доступа к элементам с произвольным ключом. Если ключ — некое слово  $w$  из букв  $a_1 \dots a_m$ , то мы можем подставить в нашу схему Горнера вместо

индексов элемента  $i$ ,  $j$ ,  $k$ , ... числовые коды этих букв  $\text{ord}(a_1)$ ,  $\text{ord}(a_2)$ , ... и, как все математики, считать задачу в принципе решенной. Однако как программисты мы вынуждены констатировать, что произведение этих кодов даже для восьмибуквенного ключа даст нам 16 384 терабайта, что составляет максимальное адресное пространство, аппаратно поддерживаемое современными процессорами. То есть при лобовом преобразовании ключей множество возможных значений значительно шире множества допустимых адресов в памяти (индексов массива). С другой стороны, делать такое полное отображение не имеет смысла, ведь реальные множества ключей значительно уже, и требовать взаимнооднозначного соответствия ключей и адресов излишне. Один из рецептов такого отображения — сопоставить нескольким ключам один и тот же адрес в надежде на малую вероятность одновременного появления в таблице ключей, претендующих на этот адрес. Таким же образом мы отображали целую окрестность точки вещественной оси в ее конечного рационального представителя.

Если же этот маловероятный случай произошел, и ключ преобразован в уже занятый адрес, то есть надежда вычислить за постоянное время еще один адрес, куда и переадресовать элемент с таким ключом. При поиске в таблице ситуация будет обратной: в случае неудачи по вычисленному адресу уже находится элемент с другим ключом, поэтому, как в случае поиска подстроки алгоритмом Рабина-Карпа, необходимо проверять точное совпадение ключей. Такие ситуации называются *коллизиями* или *конфликтами*, а метод прямой адресации таблиц на основании функции ключа получил название *хеширование* (hashing).

### 6.3.1 Выбор хеш-функции

Хорошая функция преобразования ключей должна равномерно распределять их по всему диапазону значений индекса. При этом преобразование может быть весьма случайным. Второе, не менее важное требование к хеш-функции — она должна эффективно вычисляться при помощи небольшого числа аппаратно поддерживаемых операций.

В качестве первого примера хеш-функции возьмем функцию  $\text{ord}(k)$ , обозначающую порядковый номер ключа  $k$  в множестве всевозможных ключей. Если таблица должна быть реализована в массиве из  $N$  элементов с индексами от 0 до  $N - 1$ , то хеш-функция

$$H(k) = \text{ord}(k) \bmod N$$

равномерно отображает значения ключа на весь диапазон изменения индексов. Неоднозначность этого отображения пропорциональна количеству оборотов, накручиваемому на длину массива мощностью типа ключа. Чем больше кратность, тем больше коллизий. Кроме того, если  $N$  является степенью 2, то эта функция эффективно вычисляется арифметическим сдвигом, а не делением. Однако, для часто встречающегося ключа — слова, отличающиеся только несколькими буквами, с большой вероятностью будут адресоваться одним и тем же индексом и не будут равномерно распределены по всему отрезку. Придется тратить дополнительные ресурсы на многократное *рехеширование*. Поэтому на практике в качестве  $N$  рекомендуют брать простое число и заменять сдвиг более сложным делением.

Еще один эффективный способ вычисления функции расстановки — применение битовых аппаратно-поддерживаемых операций к части ключа в двоичном представлении. Правда, этот эффективный способ нередко проигрывает в равномерности.

Другие примеры хеш-функций:

- размер таблицы выбирается как степень двойки, из двоичного представления ключа вырезается середина (см. л. р. №12). Этот способ обладает хорошей равномерностью распределения при равномерном поступлении ключей;
- умножение ключа на некоторую константу.

### 6.3.2 Рехеширование

Если при поиске в хеш-таблице обнаружилось, что строка, соответствующая заданному ключу, не содержит искомого элемента (с этим же ключом), то необходима вторая попытка переадресовать этот ключ внутри таблицы. Эта переадресация должна быть такой же быстрой, как первичное хеширование. Существует несколько методов генерации вторичного индекса:

1. организовать список строк с идентичным первичным ключом  $H(k)$ . Элементы этого списка размещаются либо в основной таблице, либо вне ее в области переполнения. Недостаток этого способа в том, что необходимо обеспечивать управление этими вторичными списками с их хоть и малым, но линейным временем поиска;
2. искать желаемый элемент или пустую строку в окрестности этого ключа (т. н. *открытая адресация*). Во второй попытке последовательность индексов должна однозначно вырабатываться из любого заданного ключа. Схема алгоритма поиска в хеш-таблице такова:

```
hi := H(k);
i := 0;
repeat
    if T[hi].key = k then
        { Элемент найден }
    else if T[hi].key = free then
        { Элемента в таблице нет }
    else begin { Конфликт }
        i := i + 1;
        hi := H(k) + G(i);
    end;
until { либо найден, либо его нет в таблице, либо она переполнена }
```

Функция рехеширования  $G(i)$  так же важна, как и сама функция хеширования. Если эта функция линейная, то мы смотрим следующую строку таблицы (строки закольцованны) и т. д. до тех пока не будет найден элемент с указанным ключом либо не встретится пустая строка (при вставке в таблицу — наоборот). Следовательно,  $G = i$ , а вторичные индексы  $h_i$ , употребляемые при последующих попытках, вычисляются по следующему рекуррентному соотношению:

$$h_0 = H(k)$$
$$h_i = (h_0 + i) \bmod N, \quad i = 1 \dots N - 1.$$

Недостатком линейного рехеширования является концентрация строк вокруг первичных ключей, в то время как желательно равномерное рассеивание вторичных ключей по массиву.

Другой распространенный способ рехеширования — *квадратичное*. В этом случае последовательность пробных индексов такова:

$$h_0 = H(k)$$

$$h_i = (h_0 + i^2) \bmod N, \quad i = 1 \dots N - 1.$$

Еще с тех времен, когда умножение было медленнее сложения, вычисление квадратов членов последовательности заменяют сложением с помощью рекуррентного соотношения, расписывая разность квадратов:  $h_i = i^2$ ,  $d_i = 2i + 1$ , начиная с  $h_0 = 0$ ,  $d_0 = 1$

$$h_{i+1} = h_i + d_i$$

$$d_{i+1} = d_i + 2, \quad i > 0.$$

Квадратичное рехеширование позволяет избежать кучности метода линейных проб, причем практически за ту же цену. Недостатком этого метода является то, что при поиске пробуются не все строки таблицы и во включении элемента может быть отказано, в то время как в таблице есть свободные элементы. Можно доказать, что если  $N$  — простое число, то при квадратичном рехешировании просматривается не меньше половины таблицы.

После составления хеш-таблицы нас ожидает разочарование: напечатать так быстро заполненную таблицу в упорядоченном виде, да еще сохранив естественный хронологический порядок равнозначных элементов, не удастся: организация массива в хеш-таблицу никак не упорядочивает его. Нужна сортировка. Но упорядочив элементы, мы тут же потеряем возможность быстрого доступа к данным по ключу с помощью хеш-функции.

Третий способ рехеширования — *случайное*. К вычисленному индексу добавляется случайная величина, масштабированная под длину таблицы. Главное в таком методе — суметь восстановить случайную последовательность при поиске в таблице. Эта проблема решается последовательностями псевдослучайных чисел, генерируемых линейным конгруэнтным датчиком [64]. В стандартной библиотеке C/C++ есть функция *rand()*, устанавливающая начальное положение датчика для генерации новой последовательности по заданному начальному числу, полностью определяющему эту последовательность. Очередное псевдослучайное число инициированной с помощью *rand()* последовательности можно получить посредством функции без параметров *rand()*.

Проанализируем метод прямого доступа к таблицам. Нетрудно себе представить наиболее неблагоприятные случаи метода при включении элементов и при их поиске. Существуют резонансные примеры, которые будут кучно стрелять по небольшому числу индексов и пропускать альтернативы при разрешении коллизий. Любое существенно неравномерное распределение исходных ключей неудобно для хеш-функций, составленных в обратном предположении. Для сложностной оценки метода хеширования необходимо убедиться в том, что среднее число проб будет небольшим. Если  $n$  — размер таблицы, а  $m$  — число занятых строк в ней, то среднее число попыток  $E$  для доступа к таблице к некоторому случайному ключу равно  $-\frac{\ln(1-a)}{a}$ , где  $a = \frac{m}{n+1}$  — коэффициент заполнения таблицы. Табулируя эту функцию, получим зависимость среднего числа попыток от коэффициента заполнения:

$a$	$E$
0,1	1,05
0,25	1,15
0,5	1,39
0,75	1,85
0,9	2,56
0,95	3,15
0,99	4,66

Эти цифры конкретно доказывают высокую производительность метода хеширования. Даже при 95% заполнении таблицы необходимо лишь 3 попытки стоимостью в постоянное время каждое, чтобы найти свободное место или обнаружить искомый ключ. При этом число попыток зависит лишь от коэффициента заполнения, а не от числа пустых строк в таблице. Детальное рассмотрение линейного рехеширования дает несколько худшее

число попыток  $E = \frac{1 - \frac{a}{2}}{1 - a}$ . Табулируя  $E(a)$  с тем же шагом, получаем:

$a$	$E$
0,1	1,06
0,25	1,17
0,5	1,50
0,75	2,50
0,9	5,50
0,95	10,50

Сведем в таблицу оценки различных методов доступа [91]:

Метод	Операция	Размер таблицы				Формулы
		10	50	200	1000	
Вектор	включение	5,50	25,5	101	501	
	поиск	5,50	25,5	101	501	
Дерево поиска на векторе	включение	9,3	31,7	108	511	
	поиск	2,7	4,8	7,0	9,0	
Дерево поиска на дин. структурах	включение	5,7	8,9	11,7	15	
	поиск	4,2	7,1	10,2	13,0	
Хеш-таблицы (заполнены наполовину)	включение	2,25	2,25	2,25	2,25	
	поиск	1,25	1,25	1,25	1,25	

Из таблицы видно, что даже такой простой способ разрешения коллизий как линейное рехеширование существенно лучше поисковых деревьев в их самом изощренном варианте.

Основной недостаток таблиц с функциями расстановки по сравнению с динамическими древовидными структурами — фиксированный размер таблиц и трудность настройки на реальные требования. Успех метода хеширования обуславливается хорошей априорной оценкой классифицируемых элементов, которая позволяет с одной стороны не завышать размер массива, а с другой стороны — избежать трудоемких линейных рехеширований. Грубо говоря, размер таблицы представляет собой несколько завышенную оценку максимального числа элементов.

Не менее существенным недостатком таблиц прямого доступа является неудобство изъятия строк из расстановочной таблицы, если имело место линейное, квадратичное или случайное рехеширование. Древовидные структуры для этой цели остаются более привлекательными.

# Глава 7

## Методы программирования

### 7.1 Модульное программирование

К настоящему времени нам известны две простые технологии программирования 60-х годов: структурное программирование или программирование с ограниченным набором конструкций (и без `goto!`), элементами которой являются нисходящая разработка структуры программы, пошаговая детализация и псевдокод [19], и процедурное программирование, основанное на широком использовании процедур и функций в том числе и для реализации операций и отношений реализуемых программистом типов данных [27]. Процедурное программирование позволяет оперировать крупными прикладными понятиями, укрывая детали реализации в телах процедур. Вспоминая определение типа данных вообще, мы можем вновь констатировать, что набор процедур и функций может быть использован для реализации нового типа данных, не поддерживаемого ни аппаратурой, ни системой программирования. Такие типы данных принято называть *абстрактными типами данных (АТД)* [40].

Рассмотрим средства стандарта Паскаля для реализации АТД. Во-первых, Паскаль расписан на составление монолитных программ (**program**) одним человеком за небольшое время. Подобно математику, Паскаль-программист должен сказать, что-то вроде «пусть  $x$ ,  $y$  и  $z$  — вещественные величины, а  $i$ ,  $j$ ,  $k$  — целые. И пусть  $p_1$ ,  $p_2$ ,  $p_3$  и  $f_1$ ,  $f_2$  — процедуры и функции, реализующие операции и отношения над ними». Тогда программирование над вновь введенным набором понятий становится простым и ясным и сводится к написанию последовательностей вызовов этих процедур и функций, образующих своеобразный специализированный интерпретируемый язык программирования. Однако воспользоваться построенным типом практически невозможно: ни продать, ни распространить для всеобщего использования в товарном виде, ни даже систематически использовать в других программах. Основная причина этого: процедуры и функции в Паскале не являются ни программными, ни текстовыми единицами этого языка и системы программирования. Эти внутренние процедуры — неотъемлемые части программной единицы, крепко связанные с ней глобальными переменными, описаниями типов, формальными и фактическими параметрами. Наличие только внутренних процедур существенно снижает эффективность процедурного программирования: программирование на стандарте Паскаля не может достигать промышленных масштабов. Его вершиной была реализация компилятора с Паскаля, состоящая примерно из десяти тысяч строк. Для того, чтобы программное обеспечение абстрактного типа данных было удобным для использования товарным

продуктом, необходимо добиться его автономного описания и функционирования в виде некоторой программной единицы. Услугами, предоставляемыми такой единицей, могут быть: доступ к атрибутам типа и его реализации (константам, описаниям типов, переменных, процедур и функций). То есть эта программная единица агрегирует программно-информационные компоненты реализации типа, доступные через программный интерфейс. Для описания таких программных единиц во многих практических языках программирования предусмотрены внешние, автономные, отдельно компилируемые процедуры (например, подпрограммы типа **SUBROUTINE** Фортрана, функции языка Си). Связь таких подпрограмм осуществляется *после компиляции* системным компоновщиком (линкером, редактором связей или связывающим загрузчиком). Автономия внешних подпрограмм удобна для создания библиотек откомпилированных подпрограмм. Но чтобы набор таких подпрограмм реализовал АТД, необходимо более тесно связать их по данным. Например, все процедуры работы со стеком должны быть связаны с массивом или динамической структурой, на которой реализован стек. То есть желаемая программная единица должна включать несколько процедур, работающих над общим набором данных. Кроме того, наши требования к типу данных вообще предполагают отделение абстрактного описания от реализации с сокрытием ее подробностей во внутренних, служебных структурах. Это называется *инкапсуляцией*.

### 7.1.1 Модули в расширениях языка Паскаль

Отсутствие модулей — внешних отдельно компилируемых процедур — было крупнейшим недостатком стандартного Паскаля. Поэтому этим обесокоился Создатель, начав исследовательскую работу по модульному расширению Паскаля, проводя попутно множество улучшений этого языка. В результате в 1976 году появился интересный язык с соответствующим названием — Модула. К сожалению, модульные средства этого языка были ориентированы в первую очередь на инкапсуляцию низкоуровневых средств (драйверов устройств, обращений к системным услугам ОС, управление памятью и др.). В 1979 году появился более развитый язык Модула 2, в котором уже стала возможной более или менее полноценная реализация АТД [66]. Кроме того, этот язык испытал некоторое благотворное влияние языка Си. Помимо автора языка улучшением языка озабочились и другие: сообщество GNU, группа научных и инженерных организаций, промышленные фирмы (DEC [92], IBM, Borland [55]). Все они вводили свои, более или менее удачные, но несовместимые друг с другом внешние процедуры и модули. Хорошо проработанным модульным расширением Паскаля стал язык Ада [61], разработанный по заказу Министерства Обороны США.

Например, в GNU Pascal используются три типа модулей: модули в стандарте Extended Pascal (научное сообщество), модули в стандарте Borland Pascal (основная промышленная реализация на платформе Intel) и упрощенные модули GNU Pascal [56]. Рассмотрим модуль работы с параметрами командной строки операционной системы. В разных ОС существуют различные способы получения этих параметров. Предположим, мы хотим написать модуль, реализующий эту связь единообразно для различных операционных систем. Для этого мы спрячем подробности реализации в тело модуля, которое будем менять от системы к системе, сохраняя при этом функциональную спецификацию. Рассмотрим пример модуля взаимодействия с ОС для различных версий Паскаля в НР Tru64 UNIX. Для GNU Pascal 2.0 эта услуга доступна через системные вызовы `_p_paramcount()` и `_p_paramstring()`. с использованием модулей Extended Pascal:

1. файл cmdlin.h — интерфейсная часть модуля

```
module cmdlin interface;

export cmdlin = (MaxLen, ArgType, ParamStr, ParamCount);

const MaxLen = 255;

type ArgType = string(MaxLen);

function ParamCount : integer;
function ParamStr(x : integer): ArgType;

end.
```

2. файл cmdlin.pas — реализация модуля

```
{ Включение файла с объявлением интерфейса }
#include "cmdlin.h"
module cmdlin implementation; { Особенность данной реализации –
программный код обращается к функциям системной библиотеки СП
GNU. В другой версии Паскаля тело функции может выглядеть иначе.
Интерфейс, скорее всего, не изменится! }

function _p_paramcount : Integer; C;
function _p_paramstr (x : integer; var s : string) : boolean; C;

function ParamCount : integer;
begin
  ParamCount := _p_paramcount;
end;

function ParamStr; { допускается опускать заголовок }
var t : ArgType;
begin
  if _p_paramstr(x, t) then
    ParamStr := t
  else
    ParamStr := '';
end;

to begin do
  writeln('Module_CmdLin_is_initializing'); { Пролог }
to end do
  writeln('Module_CmdLin_is_deinitializing'); { Эпилог }
end. { MODULE }
```

3. файл main.pas — основная программа, использующая описанный модуль.

```

#include "cmdlin.h" { Су-подобная директива препроцессора СП GNU.
    Включает интерфейсный файл в исходный код программы. }
program Main;
import cmdlin;

begin
    writeln('Число параметров данного вызова:', paramcount);
end.

```

В GNU Pascal 2.1 механизм работы с командной строкой устроен по-другому, в стиле Borland, но модульная структура примера сохраняется:

1. файл cmdlin.pas

```

unit cmdlin;

interface

const MaxLen = 255;
type ArgType = string(MaxLen);
function ParamCount : integer;
function ParamStr(x : integer): ArgType;

implementation

function _p_paramcount : Integer; C;
function _p_paramstr (x : integer; var s : string) : boolean; C;

function ParamCount : integer;
begin
    ParamCount := _p_paramcount;
end;

function ParamStr(x : integer): ArgType;
var t : ArgType;
begin
    if _p_paramstr(x, t) then
        ParamStr := t
    else
        ParamStr := '';
end;

```

2. файл Main.pas

```

program Main;
uses cmdlin;
begin
    writeln('Число параметров данного вызова:', ParamCount);
end.

```

В системе программирования Compaq Pascal получение параметров командной строки осуществляется по-другому, в соответствии со стандартом языка Си и ОС UNIX через функцию *ARGC()* и процедуру *ARGV()*. *ARGC()* и *ARGV()* являются Паскаль-аналогами параметров функции *main()* языка Си. Первым аргументом командной строки (часто единственным!) является имя программы.

Целочисленная функция *ARGC()* не имеет параметров и возвращает число аргументов командной строки (слов, разделённых пробелами и табуляциями), считая имя программы.

Процедура *ARGV()* выдаёт конкретный аргумент командной строки. Номер аргумента задаётся первым параметром. Нумерация ведётся с нуля. Второй параметр *ARGV()* — массив литер. Признаком конца строки является знак *chr(0)*.

Прежде чем запрашивать аргументы, следует убедиться в их наличии с помощью *ARGC()*.

```
program t(output);
type str = packed array [1.. 81] of char;
procedure print(var s : str);
var c : integer;
begin
  c := 1;
  while s[c] <> chr(0) do begin
    write(s[c]);
    c := succ(c);
  end;
end;
var s : str;
  i : integer;
begin
  i := 0;
  while(i < (argc - 1)) do begin { Распечатка параметров командной строки }
    argv(i, s);
    print(s);
    writeln;
    i := i + 1;
  end;
end
```

Итак, мы рассмотрели несколько способов реализации начинки модуля работы с командной строкой. Еще раз подчеркнем, что интерфейсная часть модуля при этом не меняется.

#### 7.1.1.1 Модуль абстрактного типа данных очередь

В качестве нетривиального примера приведем модуль реализации очереди на кольцевом буфере и программу, сортирующую очереди различными методами сортировки от пузырка до быстрой, преломленными не только на последовательности, но и на очереди с их разрушающим чтением. Эта программа импортирует процедуры работы с очередью, в терминах которых и написаны все сортировки.

{ файл queue.p }

{ компилируется gpc – с queue.p }  
{ Д. Райли, Т.Э. Журавлева, С.С. Крылов }

module Queues interface;

export Queues = (TValue, Queue, Init, Empty, Pop, Top, Push, Display, Size);

**const** N = 10;

**type** TValue = integer;

Queue = **record**

First, Total : 0..N;

Body: **array** [1..N] **of** TValue

**end**;

**procedure** Init(**var** Q : Queue);

{ инициализировать }

**function** Empty(Q : Queue) : **boolean**; { пусто ? }

**procedure** Pop(**var** Q : Queue); { удалить первый элемент }

**function** Top(Q : Queue): TValue; { значение первого элемента }

**procedure** Push(**var** Q : Queue; V : TValue); { добавить элемент в конец }

**procedure** Display(Q : Queue); { распечатать содержимое }

**function** Size(Q : Queue): **integer**; { количество элементов }

**end.**

module Queues implementation;

**procedure** Error(S : string); { Внутренняя процедура, не экспортируется }

**begin**

writeln(S);

halt;

**end;**

**procedure** Init(**var** Q : Queue); { Внешняя процедура, экспортируется через интерфейс модуля }

**begin**

Q.Total := 0;

Q.First := 1

**end;**

**function** Empty(Q : Queue): **boolean**;

**begin**

Empty := Q.Total = 0

**end;**

**procedure** Pop(**var** Q : Queue);

```

begin
  if Q.Total = 0 then
    Error('!_Queue_is_empty')
  else begin
    dec(Q.Total); {Q.Total := Q.Total-1}
    Q.First := (Q.First mod N) + 1
  end
end;

function Top(Q : Queue): TValue;
begin
  if Q.Total=0 then
    Error('!_Queue_is_empty')
  else
    Top := Q.Body[Q.First]
  end;

procedure Push(var Q : Queue; V : TValue);
begin
  if Q.Total = N then
    Error('!_Queue_overflow')
  else begin
    Q.Body[((Q.First + Q.Total - 1) mod N) + 1] := V;
    inc(Q.Total);
  end
end;

procedure Display(Q : Queue);
var i, c : 0..N;
begin
  c := Q.First;
  write('[');
  for i := 1 to Q.Total do begin
    write(Q.Body[c] : 2);
    write(',');
    c := (c mod N) + 1
  end;
  write(']');
  writeln
end;

function Size(Q : Queue): integer;
begin
  Size := Q.Total
end;

```

end.

### 7.1.1.2 Модуль внешних сортировок абстрактного типа данных очередь

```
{ файл sort.p }
{ ===== }
{ Программный модуль сортировки, импортирующий процедуры работы с
  очередью. Во всех случаях, кроме быстрой сортировки, в процедурах
  используются две вспомогательные очереди, каждая из которых по очереди
  служит входной/выходной }

Д. Райли, Т.Э. Журавлева, С.С. Крылов }
```

```
program QUSort;

import Queues;

var i, k : integer;
  c : char;
  TEST : array [1..3] of string(10);
  Q : Queue;

{ ===== }
{ сортировка ВЫБОРКОЙ }

function Min(Q : Queue) : TValue;
{ определить минимальный элемент очереди }
var m : TValue;
begin
  m := Top(Q); { Минимальный элемент определяется выталкиванием всех
    компонент локальной копии очереди (параметр Q передается по значению!) }
  Pop(Q); {Разрушающее чтение!}
  while not Empty(Q) do begin
    if Top(Q) < m then
      m := Top(Q);
    Pop(Q)
  end;
  Min := m;
end;

procedure Remove(var Q, Q1 : Queue; V : TValue);
{ переписать в выходную очередь Q1 все элементы очереди Q кроме элемента V }
begin
  Init(Q1);
  { Часть очереди до элемента V... (может быть пустой!) }
  while Top(Q) <> V do begin
    Push(Q1, Top(Q));
  end;
end;
```

```

Pop(Q)
end;
Pop(Q); { ... собственно элемент V... (обязательно присутствует в очереди
как выбранный из её состава минимальный элемент!) }
{ ... и хвост очереди после элемента V (тоже может быть пустым) }
while not Empty(Q) do begin
    Push(Q1, Top(Q));
    Pop(Q)
end;
end;

procedure FetchSort(var Q : Queue);
{ по очереди удалять из исходной очереди минимальный элемент }
{ и вставлять его в результирующую очередь }

var QQ : array [boolean] of Queue; { Массив из двух вспомогательных очередей с
булевским индексом. Удобен для их переключения, не требует копирований
очередей }
m : TValue;
i : boolean;
begin
    i := false;
    QQ[i] := Q; { Сортируемая очередь копируется в первую рабочую очередь... }
    Init(Q); { ... после чего используется как выходная последовательность }
    while not Empty(QQ[i]) do begin { Пока есть неотсортированные элементы }
        m := Min(QQ[i]); { Выбираем минимальный из них }
        Remove(QQ[i], QQ[not i], m); { Удаляем его из первой рабочей очереди,
переписывая оставшиеся элементы во вторую рабочую очередь }
        Push(Q, m); { Помещаем минимальный элемент в отсортированную
последовательность }
        i := not i; { Вспомогательные очереди меняются ролями }
    end;
end;

{ ===== }
{ сортировка ПРОСТОЙ ВСТАВКОЙ }

procedure Insert(Q:Queue; var Q1: Queue; V: TValue); { Локальная
вспомогательная процедура }
{ переписать в Q1 все элементы Q, вставив на нужное место V }

begin
    Init(Q1);
    if Empty(Q) then { Очередь пуста, место вставки не надо }
        Push(Q1, V)
    else begin
        while not Empty(Q) do begin { Поиск места для вставки, перебираемые
элементы очереди переносятся в выходную очередь Q1 }

```

```

if Top(Q) > V then { Найден больший элемент, вставка перед ним! }
    break; { Модули и break есть в стандарте Extended Pascal }
Push(Q1, Top(Q)); { Поиск продолжается с поэлементным переносом из
    входной очереди в выходную }
Pop(Q); { Перенесенный элемент из входной очереди удаляется }
end;

Push(Q1,V); { В выходную очередь на найденное место помещается очередной
    элемент сортируемой последовательности }

while not Empty(Q) do begin { Копирование остатка входной очереди в
    выходную, работает и для пустых остатков }
    Push(Q1,Top(Q));
    Pop(Q)
end
end
end;

procedure InsertSort(var Q : Queue);
var QQ : array [boolean] of Queue;
    i : boolean;
begin
    i := false;
    Init (QQ[i]);
    while not Empty(Q) do begin
        Insert (QQ[i], QQ[not i], Top(Q)); { Верхушку неотсортированной очереди Q
            вставляем в текущую частично отсортированную последовательность
            QQ[i] с направлением результата в QQ[not i]... }
        Pop(Q); { ... и удаляем из исходной очереди }
        i := not i
    end;
    Q := QQ[i]; { Результат сортировки возвращаем в исходной очереди }
end;
{ ===== }
{ сортировка СЛИЯНИЕМ }

procedure Merge(Q1, Q2: Queue; var Q : Queue); { Вспомогательная процедура }
{ слить упорядоченные очереди Q1 и Q2, результат дописать в Q. }
{ Q не инициализируется, так что новые элементы добавляются в конец }
begin
    { Синхронный просмотр сливаемых очередей пока хотя бы одна из них не
        опустеет }
    while not Empty(Q1) and not Empty(Q2) do
        if Top(Q1) < Top(Q2) then begin { Верхушка первой очереди меньше
            верхушки второй }
            Push(Q, Top(Q1)); { Переливаем ее в выходную очередь }

```

```

Pop(Q1) { отливая ее из входной }
end
else begin { В противном случае переливается верхушка второй очереди }
    Push(Q, Top(Q2));
    Pop(Q2)
end;
{ Перелив остатка первой очереди.}
while not Empty(Q1) do begin
    Push(Q, Top(Q1));
    Pop(Q1)
end;
{ Перелив остатка второй очереди.}
while not Empty(Q2) do begin
    Push(Q, Top(Q2));
    Pop(Q2)
end
{ Один из этих двух последних циклов всегда не выполняется, т. к.
соответствующая очередь пуста по одному из условий завершения
основного цикла; если Q1 = Q2 = Ø, то ни один из циклов не выполняется }
end;

```

**procedure** MergeSort(**var** Q: Queue);  
{ выделять с начала исходной очереди упорядоченные подочереди; сливать их  
парно и присоединять результат вхвост выходной очереди. Когда исходная  
очередь закончится, провести аналогичную процедуру с результатом и т. д.  
до тех пор, пока вся очередь не станет упорядоченной.

Двухфазная четырехленточная сортировка (не считая @ Q) – дополнительная  
лента нужна для простоты программирования в связи с тем, что чтение из  
очереди разрушающее }

```

var QQ: array[boolean] of Queue;
    Q1, Q2 : Queue;
    V : TValue;
    i, all : boolean;
begin
    i := false; { Переключатель рабочих очередей }
    QQ[i] := Q; { QQ[false] := Входная очередь }
    Init(QQ[not i]); { QQ[true] := пусто }
repeat
    { Установление факта завершенности процесса сортировки }
    if Empty(QQ[i]) then begin
        i := not i;
        all := true
    end
    else {Слияние продолжается}
        all := false;

```

```

V := Top(QQ[i]); { Из рабочей очереди вынимается голова. }
Pop(QQ[i]);
Init(Q1);
Init(Q2);
{ Упорядоченный отрезок из QQ[i] переписывается в Q1 }
while not Empty(QQ[i]) do begin
    if V > Top(QQ[i]) then { Отрезок заканчивается, как только обнаружится
        инверсия. }
        break;
    Push(Q1, V); { Очередной элемент отрезка добавляется в хвост активной
        очереди. }
    V := Top(QQ[i]) { Из рабочей очереди извлекается следующий. };
    Pop(QQ[i])
end;
Push(Q1, V);
{ Если разделяемая очередь непуста }
if not Empty(QQ[i]) then begin
    all := false;
    { ... то переписать в текущую выходную (активную) очередь Q2 еще один
        упорядоченный отрезок (аналогично). }
    V := Top(QQ[i]);
    Pop(QQ[i]);
    while not Empty(QQ[i]) do begin
        if V > Top(QQ[i]) then
            break;
        Push(Q2, V);
        V := Top(QQ[i]);
        Pop(QQ[i])
    end;
    Push(Q2, V);
    { Слитъ очереди с помещением результата в приемник QQ[not i] }
    Merge(Q1, Q2, QQ[not i])
end
{ Поскольку Q2 пуста, то слияние как таковое не выполняется. Вместо
этого упорядоченный отрезок из Q1 доливается в QQ[not i] }
else begin
    while not Empty(Q1) do begin
        Push(QQ[not i], Top(Q1));
        Pop(Q1)
    end
end
until all ;
Q := QQ[not i] { Отсортированная очередь из приемника помещается на место
исходной }
end;

```

```

{ =====
{ БЫСТРАЯ сортировка }

procedure Append(var Q1 : Queue; Q2 : Queue); { Вспомогательная процедура }
{ конкатенация очередей: присоединить Q2 в конец Q1 }
var V : TValue;
begin
  while not Empty(Q2) do begin
    Push(Q1, Top(Q2));
    Pop(Q2) { Локальная копия очереди Q2 исчерпывается }
  end
end;

{ БЫСТРАЯ сортировка в таком виде требует ОЧЕНЬ много памяти для
  вспомогательных очередей и чудовищно неэффективна }

procedure QuickSort(var Q: Queue);
var Q1, Q2 : Queue; { Локальные переменные—очереди, отводятся всякий раз при
  рекурсивной активации процедуры }
V : TValue;
begin
  if not Empty(Q) then begin
    { Инициализация комплекса локальных очередей текущей активации
      процедуры }
    Init(Q1);
    Init(Q2);
    V := Top(Q); { из Q удаляется первый элемент }
    Pop(Q); { он будет «зерном» сортировки }
    while not Empty(Q) do begin
      if Top(Q) < V then { остаток Q разделяется на две очереди: }
        Push(Q1, Top(Q)) { В Q1 заносятся все элементы Q, меньшие зерна }
      else
        Push(Q2, Top(Q)); { В Q2 – все элементы Q, большие зерна }
      Pop(Q) { Рассматриваемый элемент удаляется из исходной очереди }
    end;
    QuickSort(Q1);
    QuickSort(Q2); { Подочереди Q1 и Q2 сортируются рекурсивно }
    Push(Q1, V);
    Append(Q1, Q2); { результирующая очередь образуется }
    Q := Q1 { как конкатенация: }
  end { (отсортированная Q1) || зерно || (отсортированная Q2) }
end;

{ =====
{ сортировка ПУЗЫРЬКОМ }

```

```

procedure BubbleSort(var Q : Queue);
{ проверить, есть ли в очереди два подряд идущих элемента, первый из которых
  большие второго. Если да, создать новую очередь—копию исходной в которой
  эти два соседних элемента обменены. }

var QQ : array[boolean] of Queue;
    i, bbl: boolean;
    V : TValue;
begin
    i := false;
    QQ[i] := Q;
    repeat
        Init(QQ[not i]);
        bbl := false; { Обменов пока не было }
        V := Top(QQ[i]); { Обмен будет происходить через буфер V }
        Pop(QQ[i]);
        while not Empty(QQ[i]) do begin
            if V > Top(QQ[i]) then
                break; { Требуется обмен }
            { Обмен не нужен }
            Push(QQ[not i], V); { копировать в выходную очередь все }
            V := Top(QQ[i]); { элементы пока соблюдается упорядоченность, }
            Pop(QQ[i]); { удаляя их из входной очереди }
        end;
        if not Empty(QQ[i]) then begin { переставить элементы }
            bbl := true;
            Push(QQ[not i], Top(QQ[i])); { Следующий, меньший элемент направляется
                на выход }
            Pop(QQ[i]); { и удаляется из входной очереди }
        end;
        Push(QQ[not i], V); { А теперь на выход следует «придержанный» больший
            элемент }

        { копировать остаток входной очереди, ленивый вариант: всего один обмен за
          проход! }
        while not Empty(QQ[i]) do begin
            Push(QQ[not i], Top(QQ[i]));
            Pop(QQ[i])
        end;
        i := not i
    until not bbl; {усовершенствованный пузырёк}
    Q := QQ[i]
end;

{ =====
{ модифицированная пузырьковая сортировка с просеиванием }

```

```

procedure SiftSort(var Q : Queue);
{ проверить, есть ли в очереди два подряд идущих элемента, первый из которых
  большие второго. Если да, создать новую очередь—копию исходной в которой
  эти два элемента обменены. Кроме того, если обмен состоялся, попытатьсяся
  еще продвинуть меньший элемент к началу выходной очереди }
var QQ : array[boolean] of Queue;
    Q1 : Queue; { Локальная вспомогательная очередь. Всего используется 4
                  очереди }
    i, bbl : boolean;
    V, V1 : TValue;
begin
    QQ[i] := Q;
    repeat
        Init(QQ[not i]);
        Init(Q1);
        V := Top(QQ[i]);
        bbl := false;
        Pop(QQ[i]);
        while not Empty(QQ[i]) do begin
            if V > Top(QQ[i]) then
                break;
            Push(Q1, V); { копировать во вспомогательную очередь Q1 }
            V := Top(QQ[i]); { элементы пока соблюдается упорядоченность }
            Pop(QQ[i]);
        end;
        if not Empty(QQ[i]) then begin { упорядоченность нарушена, переставить
                                         элементы }
            bbl := true;
            V1 := Top(QQ[i]);
            Pop(QQ[i]);
            { ... и переместить меньший элемент к началу Q1, выполнив его вставку в
              выходную последовательность }
            while not Empty(Q1) do begin
                if V1 <= Top(Q1) then
                    break;
                Push(QQ[not i], Top(Q1));
                Pop(Q1)
            end;
            Push(QQ[not i], V1); { Вставляем на найденное место }
            while not Empty(Q1) do begin { копировать остаток Q1 }
                Push(QQ[not i], Top(Q1));
                Pop(Q1)
            end;
            Push(QQ[not i], V); { копировать остаток исходной очереди }
            while not Empty(QQ[i]) do begin
                Push(QQ[not i], Top(QQ[i]));

```

```

    Pop(QQ[i])
end;
i := not i
end
else begin
    Q := Q1;
    Push(Q,V) { Вставляем в конец очереди наибольший элемент из
    участвовавших в сравнении }
end;
until not bbl { повторять пока была перестановка }
end;

{ рекурсивный реверс очереди – фактически создается резидентная копия очереди
  в стеке значений экземпляров переменной V, неявно образующемся при
  рекурсивном вызове }

procedure Reverse(var Q : Queue);
var V : TValue;
begin
if not Empty(Q) then begin
    V := Top(Q);
    Pop(Q);
    Reverse(Q);
    Push(Q, V)
end
end;

```

```

begin
TEST[1] := '0123456789';
TEST[2] := '9876543210';
TEST[3] := '7832927405';
repeat
    write('F)etch_I)nsertion_B)ubble_Q)uick_M)erge_S)ift_R)everse_E)xit>');
    readln(c);
    if c in ['F', 'I', 'R', 'B', 'Q', 'M', 'S'] then begin
        for k := 1 to 3 do begin
            Init(Q);
            for i := 1 to 10 do
                Push(Q, ord(TEST[k][i]) - ord('0'));
            writeln('тест_№_', k : 2);
            write('Исходная_очередь:');
            Display(Q);
            case c of
                'F': FetchSort(Q);
                'I': InsertSort(Q);
                'B': BubbleSort(Q);
                'Q': QuickSort(Q);

```

```

'M': MergeSort(Q);
'S': SiftSort (Q);
'R': Reverse(Q)
end;
write('Результат:_____');
Display(Q);
end;
writeln;
end;
until c = 'E'
end.

```

### 7.1.2 Экспорт и импорт объектов

Модуль представляет собой программную единицу, изолированную от объектов других программ. Он как бы заключен в оболочку, предохраняющую внутренние объекты модуля от нежелательного доступа извне. К некоторым данным и услугам модуля доступ может быть разрешен и соответствующие разрывы в его оболочке называются *интерфейсом модуля*. Таким образом, для организации межмодульного взаимодействия его участники должны обменяться взаимными объявлениями желаемых и доступных объектов друг друга. В соответствии с направленностью этих интерфейсов они получили название *экспорта* и *импорта* [27]. Экспортируемые объекты модуля — это константы, типы, переменные и процедуры, которые доступны для использования в других модулях. Импорт объектов — это получение доступа к экспортируемым объектам других модулей. Списки объектов экспорта-импорта перечисляются в интерфейсной части модуля.

## 7.2 Программирование в абстрактных типах данных

Абстракция, как и типы, в той или иной степени присуща любому языку программирования сколько-нибудь высокого уровня [46]. Как пишет Э. Дейкстра, даже понятие переменной представляет собой абстракцию соответствующего текущего значения. Тем более следует считать абстракцией понятия типа и системы типов. Описание тех же типов в терминах стандартной реализации так же является абстракцией по отношению к сугубо машинному описанию. Таким образом, абстракция в программировании имеет уровни, высота которых измеряется по отношению к целевой машине, которая, в свою очередь, может быть абстрактной или программно эмулируемой [59].

Абстракция — это не только отвлечение от чего-то несущественного и потому помогающее лучше отразить суть дела. Это также инструмент познавательной деятельности человека, приводящий к абстрактным понятиям. Поэтому имеет смысл говорить о средствах абстракции, и в частности о средствах абстракции в языках программирования, которые сами являются средством понимания и построения алгоритмов и обмена мыслями и результатами между программистами.

После процедурной абстракции Паскаля был осуществлен переход к более высокому уровню абстракции, связанному с понятием *абстрактного типа данных*. АТД — это, по существу, определение некоторого понятия в виде класса (одного или более) объектов с некоторыми свойствами и операциями. Так как свойства обычно выражаются в терминах операций и предикатов, которые тоже считаются операциями, то абстрактный тип данных

часто отождествляется с соответствующим множеством операций. Например, алгебраический (операционистский) взгляд на понятие стека в терминах операций «втолкнуть элемент в стек», «вынуть из него», «создать новый стек», «показать верхний элемент» и т. д. с технологической позиции воплощает нисходящую точку зрения на АТД как на набор операций, но не как на множество оперируемых объектов.

В языке программирования такое определение оформляется как специальная синтаксическая конструкция — часть программы, которая называется классом (в языках Симула 67 и Concurrent Pascal), кластером (CLU), формой (Alphard), модулем (Modula, Euclid), определением (Mesa), пакетом (Ада), капсулой (Russel). Семантическая роль этой конструкции, ее содержимое и взаимодействие с другими частями программы в разных языках могут быть существенно разными. В самой развитой форме в определение АТД входят следующие четыре части:

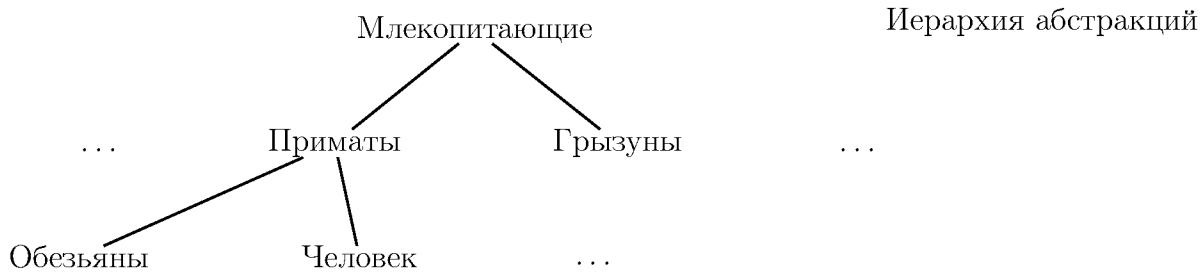
1. внешность (видимая часть, сопряжение, интерфейс), содержащая имя определяемого типа (понятия), имена операций с указанием типов их аргументов и значений и т. п.;
2. абстрактное описание операций и объектов, с которыми они работают, средствами некоторого языка спецификаций, допускающего, в частности, формулирование свойств;
3. конкретное (логическое) описание этих операций на обычном распространенном языке программирования предыдущего поколения;
4. описание связи между 2 и 3, объясняющее, в каком смысле часть 3 корректно представляет часть 2.

В наиболее развитой форме АТД присутствуют в исследовательских языках Alphard, CLU, и, конечно же, в полноценных средах объектно-ориентированного программирования, поглотивших АТД в качестве неотъемлемой составной части.

### **7.2.1 Методы абстракции в языках программирования**

Процесс абстракции может быть рассмотрен как некоторое обобщение [21]. Он позволяет нам забыть об информации и, следовательно, рассматривать различные предметы так, как если бы они были эквивалентны. Мы выполняем это в надежде упростить наш анализ, отделяя существенные атрибуты от несущественных. Однако при этом важно знать, что критерий такого отделения во многом зависит от контекста. В контексте школьного курса мы учимся абстрагировать от  $(8/3) * 3$  и  $5 + 3$  к понятию, представленному числом 8. В дальнейшем мы узнаем, что при работе на большинстве вычислительных машин такая абстракция приводит к неприятностям [40].

Например, рассмотрим структуру, приведенную на рисунке. Понятием здесь является «млекопитающие». Все млекопитающие обладают определенными общими характеристиками. Например, все женские особи этого класса вырабатывают молоко. На таком уровне абстракции мы фокусируемся именно на этих общих характеристиках и игнорируем различия между существующими видами млекопитающих.



На нижнем уровне абстракции мы сосредоточиваем свое внимание на конкретных представителях семейства млекопитающих. В этом случае мы можем абстрагироваться, рассматривая не отдельных представителей или даже особей, а группы родственных особей, например, приматов или грызунов. Здесь опять мы рассматриваем общие характеристики, например тот факт, что все приматы носят своих детенышей, а не отличия между, скажем, человеком и шимпанзе. Эти отличия остаются существенными и на более низких уровнях абстракции.

Эта иерархия относится к вполне определенной области зоологии, однако она приложима также и к другим областям этой науки. Более машинно-ориентированным примером, весьма полезным при составлении большей части программ, является концепция файла. Файлы абстрагированы от конкретного носителя, реализуя долговременное и постоянно доступное хранилище поименованных единиц. Операционные системы отличаются по способам работы с файлами. Например, структура имен файлов может меняться от системы к системе. Изменяется также и способ их размещения на устройствах внешней памяти.

В данном разделе мы рассмотрим различные виды абстракций и общие принципы применения их в программах. Наиболее существенным достижением в этой области является на сегодняшний день развитие языков высокого уровня. Имея дело непосредственно с конструкциями языка высокого уровня, а не с различными наборами машинных инструкций, в которые данные конструкции могут быть транслированы, программист существенно упрощает свой труд.

В последние годы программистов перестал удовлетворять уровень абстракции, достичь гаемый в программах, написанных даже на языке высокого уровня. Рассмотрим фрагмент программы линейного поиска:

<pre> found :=false i := lowbound(a) while i &lt; highbound(a) + 1 do   if a[i] = e     then z := i     found :=true   end   i := i + 1 end </pre>	<pre> found :=false i := highbound(a) while i &gt; lowbound(a) - 1 do   if a[i] = e     then z := i     found :=true   end   i := i - 1 end </pre>
--	--

На уровне абстракции, определенном использованным языком высокого уровня, очевидно, что приведенные фрагменты отличаются друг от друга: если *e* присутствует в *a*, то первый фрагмент отыскивает индекс его первого вхождения, а второй - индекс последнего вхождения. Первая программа устанавливает *i* в *highbound(a)* + 1, а вторая

- в  $lowbound(a) - 1$ . Обе программы, однако, были написаны для выполнения одной и той же функции: установить в  $found$  значение **false**, если  $e$  отсутствует в  $a$ , а в противном случае установить в  $found$  **true** и в  $z$  — индекс вхождения  $e$  в  $a$ . Если нам необходимо выполнение именно этого требования, то становится очевидным, что приведенные фрагменты программ не находятся на требуемом уровне абстракции.

Одним из подходов к решению такой проблемы является создание языков более высокого уровня, реализованных на базе некоторого фиксированного набора относительно обобщенных универсальных структур данных и мощном наборе примитивов, используемых для манипуляции с ними. Например, рассмотрим язык, в котором имеются примитивы `is_in` и `index_of`, позволяющие осуществлять соответствующие операции над массивами. Тогда рассматриваемая задача легко реализуется следующим образом:

```
found := is_in (a, e)
if found then z := index_of (a, e) end
```

Недостатком такого подхода является предположение о том, что разработчик языка включит в него большинство абстракций, которые могут понадобиться пользователю. Предвидеть все возможные ситуации довольно затруднительно. Однако, даже если это и удалось бы сделать, получившийся язык содержал бы столь много встроенных абстракций, что работа с ним стала бы невозможной.

Альтернативой может быть создание таких языковых механизмов, которые позволяют программисту создавать свои собственные абстракции по мере необходимости. Наиболее распространенным механизмом такого рода является использование *процедур*. Разделяя в программе тело процедуры и обращения к ней, язык высокого уровня реализует тем самым два важных метода абстракции: *абстракция через параметризацию* и *абстракция через спецификацию*.

### 7.2.1.1 Параметризационный метод

Абстракция через параметризацию позволяет нам, используя параметры, представить фактически неограниченный набор различных вычислений одной программой, которая есть абстракция всех этих наборов [40]. Рассмотрим следующий текст программы:

$$x * x + y * y$$

В ней описаны вычисления, в процессе которых квадрат значения некоторой переменной  $x$  складывается с квадратом значения переменной  $y$ . С другой стороны, выражение  $\lambda$

$$\lambda x, y : \text{int}.(x * x + y * y)$$

описывает набор вычислений, в процессе которых возводится в квадрат значение, хранимое в некоторой целочисленной переменной (к полученному результату мы временно ссылаемся как к  $z$ ) и этот результат добавляется к квадрату значения, хранимого в другой целочисленной переменной, временно называемой нами  $u$ . В этом выражении мы называем  $x$  и  $y$  формальными параметрами, а выражение  $x * x + y * y$ -телом выражения. Мы производим вычисления, связывая формальные параметры с аргументами и вычисляя затем все выражение. Например,

$$x, y : \text{int}.(x * x + y * y)(w, z) \text{ эквивалентно } w * w + z * z$$

В более привычном представлении мы можем заменить приведенное выше выражение  $\lambda$  на

```
squares = proc (x, y: int) returns (int)
    return (x * x + y*y)
    end
```

а замену формальных параметров — на фактические и вычисление выражения можем выполнять через обращение к процедуре: `squares(w,z)`

Программисты часто используют абстракцию через параметризацию, даже не замечая этого. Например, предположим, что нам необходима процедура, которая сортирует массив целых чисел  $a$ . В дальнейшем, вероятно, понадобится отсортировать некоторый другой массив, возможно, в другом месте этой же программы. Однако маловероятно, что каждый такой массив будет иметь имя  $a$ . Мы, следовательно, используем абстракцию через параметризацию, обобщая этим процедуру и делая ее более универсальной.

Абстракция через параметризацию является важным средством повышения универсальности программ. Программа `sort()`, сортирующая произвольный массив чисел, полезнее той, которая может работать только с конкретным массивом целых чисел. Дальнейшее абстрагирование позволяет еще больше обобщить программу. Например, мы можем определить абстракцию `sort()`, которая работает над массивами как целых, так и действительных чисел или даже вообще над различными массивоподобными структурами.

Абстракция через параметризацию — весьма полезное средство. Она не только позволяет относительно просто описывать большое (даже бесконечное) число вычислений, но и является легко и эффективно реализуемой на языках программирования. Однако этот способ не позволяет полностью реализовать тот уровень обобщения, который можно достичь при работе с процедурами.

### 7.2.1.2 Спецификационный метод

Абстракция через спецификацию позволяет нам абстрагироваться от процесса вычислений, описанных в теле процедуры, до уровня знания лишь того, что данная процедура должна в итоге реализовать [40]. Это достигается путем задания для каждой процедуры *спецификации*, описывающей эффект ее работы, после чего смысл обращения к данной процедуре становится ясным через анализ этой спецификации, а не самого тела процедуры.

Мы пользуемся абстракцией через спецификацию всякий раз, когда связываем с процедурой некий комментарий, достаточно информативный для того, чтобы иметь возможность работы с ней без анализа тела процедуры. Одним из удобных способов составления подобных комментариев является использование пар *утверждений*. Утверждение *requires* (или *начальное условие*) задает на входе процедуры истинность или ложность некоторого условия. Обычно на практике правильное выполнение процедуры задается некоторым набором таких условий. (Это часто просто означает, что некоторое логическое утверждение истинно.) Утверждение *effects* (или *конечное условие*) задает некоторое условие, которое предполагается истинным по завершению данной процедуры в предположении, что для этой процедуры было удовлетворено начальное условие.

Рассмотрим, например, процедуру `sqrt()`.

```
% requires coef > 0
```

```
% effects возвращает приближенное значение корня из coef
```

```

ans: real=coef/2.0
i:int:=1
while i < 7 do
    ans:=ans-((ans*ans-coef)/(2.0*ans))
    i:=i+1
end
return (ans)
end sqrt

```

Поскольку в процедуре имеется спецификация, мы можем игнорировать тело процедуры. Обращение вида **call**  $y := \text{sqrt}(x)$  будет иметь следующий смысл: «Если при вызове процедуры  $x$  больше 0, то после выполнения процедуры  $y$  есть приближение квадратного корня из  $x$ ». Отметим, что начальные и конечные условия позволяют нам не упоминать о значении  $y$  для случая, когда  $x$  не больше 0.

При анализе спецификации для уяснения смысла обращения к процедуре мы придерживаемся двух четких правил:

1. После выполнения процедуры можно считать, что конечное условие выполнено.
2. Можно ограничиться *только* теми свойствами, которые подразумевает конечное условие.

Эти два правила демонстрируют два преимущества, предоставляемых абстракцией через спецификацию. Первое из них состоит в том, что использующие данную процедуру программисты не обязаны знакомиться с ее телом. Следовательно, они освобождены от необходимости уяснения подробностей выполнения описанных в теле вычислений, устанавливая, что процедура действительно извлекает квадратный корень из аргумента. Это является большим преимуществом при работе со сложными процедурами и даже с простыми, но использующими незнакомые алгоритмы.

Второе правило уточняет, что мы на самом деле абстрагировались от тела процедуры, позволив себе не обращать внимания на несущественную информацию. Именно это «забывание информации» и отличает абстракцию от декомпозиции. Анализируя тело процедуры `sqrt()`, пользователи данной процедуры могут извлечь для себя большое количество информации, не следующее из конечного условия, например, что `sqrt(4)` возвратит результат `+2`. Однако в спецификации мы говорим, что подобная информация о возвращаемом результате игнорируется. Этим мы утверждаем, что процедура `sqrt()` есть абстракция, представляющая собой набор всех вычислений, при котором определяется «приближение квадратного корня из  $x$ ».

### 7.2.2 Виды абстракции в языках программирования

Абстракции через параметризацию и через спецификацию являются мощными средствами создания программ. Они позволяют нам определить три различных вида абстракций: процедурную абстракцию, абстракцию данных и абстракцию через итерацию. В общем случае каждая процедурная абстракция, абстракция через данные и абстракция через итерацию используют оба способа.

Например, абстракцию `sqrt()` можно сравнить с операцией: она абстрагирует отдельное событие или задачу. Мы будем ссылаться к абстракциям такого рода как к *процедурным*.

*абстракциям*. Отметим, что абстракция *sqrt()* включает в себя как абстракцию через параметризацию, так и абстракцию через спецификацию.

Процедурная абстракция является мощным средством. Она позволяет нам расширить заданную некоторым языком программирования виртуальную машину новой операцией. Такой вид расширения наиболее полезен в том случае, когда мы работаем с задачами, которые легко представить в виде набора независимых функциональных единиц. Однако часто оказывается более удобным добавить к виртуальной машине несколько объектов данных с новыми типами.

Поведение объектов данных наиболее естественно представлять в терминах набора операций, применимых к данным объектам. Такой набор включает в себя операции по созданию объектов, получению информации от них и, возможно, их модификации. Например, операции *push()* и *pop()* принадлежат к классу операций, имеющих смысл при работе со стеками, в то время как для работы с целыми числами используются обычные арифметические операции. Таким образом, *абстракция данных* (или *тип данных*) состоит из набора объектов и набора операций, характеризующих поведение этих объектов.

В качестве примера рассмотрим мульти множества (*multisets*). Мульти множества сходны с обычными множествами, за исключением того, что элемент может входить в мульти множества несколько раз. Операции для работы с подобными мульти множествами включают в себя операции *empty()*, *insert()*, *delete()*, *number\_of()* и *size()*. Эти операции создают пустое мульти множество, удаляют и добавляют в него элементы, вычисляют, сколько раз данный элемент входит в мульти множество и сколько всего элементов содержится в мульти множестве. Данные операции могут быть реализованы в языке программирования через соответствующие процедуры. Программисты, работающие с мульти множествами, не должны беспокоиться о том, каким образом эти процедуры реализованы. Для них операции *empty()*, *insert()*, *delete()*, *number\_of()* и *size()* являются абстракциями, определяемыми функциональными спецификаторами вида:

The size of the multiset insert(s, e) is equal to size(s) + 1

(Размер мульти множества *insert(s, e)* равен *size(s) + 1*)

For all e the number\_of times e occurs in the multiset empty() is 0.

(Для всех *e* число вхождений *e* в пустое мульти множество *empty()* равно 0.)

Важно отметить, что каждый из этих операторов работает сразу с несколькими операциями. Мы не приводим независимые определения каждой операции, но скорее определяем их через взаимосвязи. Этот акцент на взаимосвязях между операциями и делает абстракцию данных существенно отличной от набора процедур. Важность этого отличия еще будет обсуждаться.

В дополнение к процедурным абстракциям и абстракциям данных мы рассмотрим также *абстракцию через итерацию*. Абстракция через итерацию дает возможность не рассматривать информацию, не имеющую прямого отношения к управляющему потоку или циклу. Типичная абстракция итерации позволяет нам обрабатывать все элементы мульти набора без накладывания каких-либо ограничений на последовательность обработки.

Далее мы покажем, как осуществлять декомпозицию программы на базе абстракции через итерацию. Акцент будет делаться на абстракции данных. Мы считаем, что, хотя процедурные абстракции и абстракции через итерацию и играют существенную роль, организация процесса программирования начинается, как правило, с абстракции данных.

При программировании на С получаются более совершенные реализации абстрактных

типов данных. Более того, привлекая инструментальные средства системы программирования ОС UNIX мы можем автоматизировать сборку программ модульной структуры с помощью утилиты make.

Заголовочный файл stack.h определяет интерфейс стека, файл stack.c — реализацию.

```
#ifndef _STACK_H_
#define _STACK_H_

#include <stdbool.h>
#include <stdio.h>

typedef int value_type;

struct stack_s;
typedef struct stack_s stack;

stack* stack_create();
bool stack_is_empty(const stack* st);
void stack_push(stack* st, value_type val);
value_type stack_pop(stack* st);
size_t stack_size(const stack* st);
void stack_destroy(stack* s);
void stack_print(const stack* st, FILE* where);

#endif

#include <stdlib.h>

#include "stack.h"

enum { max_stack_size = 100 };

struct stack_s {
    value_type data[max_stack_size];
    size_t top;
};

stack* stack_create()
{
    stack* s = (stack*)malloc(sizeof(stack));
    s->top = 0;
    return s;
}

bool stack_is_empty(const stack* s)
{
    return s->top == 0;
```

```

}

void stack_push(stack* s, value_type val)
{
    s->data[s->top++] = val;
}

value_type stack_pop(stack* s)
{
    return s->data[--s->top];
}

size_t stack_size(const stack* s)
{
    return s->top;
}

void stack_destroy(stack* s)
{
    free(s);
}

void stack_print(const stack* s, FILE* file)
{
    putc('(', file );
    if (s->top != 0)
        fprintf( file , "%d", s->data[0]);
    for (size_t i = 1; i < s->top; ++i)
        fprintf( file , ",%d", s->data[i]);
    putc(')', file );
}

```

Алгоритм сортировки стека определен в отдельных файлах sort.h и sort.c.

```

#ifndef _SORT_H_
#define _SORT_H_

```

```
#include "stack.h"
```

```
void stack_sort(stack* s);
```

```
#endif
```

В реализации сортировки используются несколько вспомогательных функций, невидимых извне.

```
#include "sort.h"
```

```
value_type find_max(stack *s1, stack *s2)
```

```

{
    value_type max = stack_pop(s1);
    while (!stack_is_empty(s1)) {
        value_type tmp = stack_pop(s1);
        if (tmp > max) {
            stack_push(s2, max);
            max = tmp;
        }
        else
            stack_push(s2, tmp);
    }
    return max;
}

void stack_sort(stack *s)
{
    stack *s1 = s;
    stack *s2 = stack_create();
    stack *res = stack_create();
    while (!stack_is_empty(s1)) {
        stack *tmp;
        stack_push(res, find_max(s1, s2));
        tmp = s1;
        s1 = s2;
        s2 = tmp;
    }
    while (!stack_is_empty(res))
        stack_push(s, stack_pop(res));
}

```

Пример программы, сортирующей стек:

```

#include "stack.h"
#include "sort.h"

int main()
{
    int x;
    stack *s = stack_create();
    while (scanf("%i", &x) == 1)
        stack_push(s, x);
    stack_sort(s);
    stack_print(s, stdout);
    putchar('\n');
    stack_destroy(s);
}

```

Как уже отмечалось, Makefile позволяет существенно сократить затраты на сборку

проекта (если только это не тотальная пересборка).

```
CC = gcc
CFLAGS = -c -pedantic -Wall -std=c99
LD = gcc
LDFLAGS =

OBJ = stack.o sort.o main.o
PROGNAME = lab26

.SUFFIXES: .c .o

all : $(PROGNAME)

clean:
    rm -f *.o $(PROGNAME)

$(PROGNAME): $(OBJ)
    $(LD) $(LDFLAGS) $(OBJ) -o $(PROGNAME)

.c.o:
    $(CC) $(CFLAGS) $<

main.c: sort.h stack.h
sort.c: sort.h stack.h
stack.c: stack.h
```

## 7.3 Типовые абстракции

### 7.3.1 Типизация языка

Типизация языка программирования — это наделение языка той или иной системой типов [46]. Существуют бестиповые языки, наиболее известным из которых среди специалистов является Lisp — в связи с тем, что он основан на бестиповом  $\lambda$ -исчислении, получившем блестящую математическую интерпретацию Д. Скотта. Бестиповыми являются и предшественники Си — языки системного программирования В и BCPL, язык BLISS, а также весьма интересный язык Сетл. В бестиповых языках все объекты одного и того же типа, например машинные слова, адреса или строки [21].

В обычных многотиповых фон-Неймановских языках программирования вопрос типизации тесно связан с контролем типов. Ведущие специалисты по языкам программирования Ч. Хоор, Б. Лисков и другие высказываются в пользу строгой типизации языка, под которой понимается статическая фиксация во время компиляции типа каждой переменной, массива, компоненты записи, выражения, функции и параметра, причем тип каждой переменной должен быть явно описан в программе и не должен меняться во время

выполнения программы.

### 7.3.2 Контроль типов

Деятельность по определению типов переменных, выражений и т. д. в программе часто называют *контролем типов*. Контроль, проводимый во время компиляции, называют *статическим*, а в период выполнения — *динамическим* [46].

Типовой контроль может быть определен также в терминах синтаксиса и семантики [21]:

- рассматриваются синтаксически правильные программы, удовлетворяющие контекстным условиям (вроде согласованности формальных и фактических параметров при вызовах процедур); эта задача эффективно разрешима алгоритмически;
- среди синтаксически правильных программ — тоже алгоритмически — могут быть выделены типово-правильные программы, удовлетворяющие правилам типизации языка — приписыванию типов переменным и константам, определению типов выражений по типам их частей, согласование типов частей конструкций (например, присваиваний) и т. п., так что понятие типово-правильной программы становится частью синтаксиса языка как свойство текста, проверяемое в период компиляции до выполнения программы;
- семантика выполнения программы на языке предусматривает понятие *типовской ошибки*, зависящее от понятия типа в языке (несоответствие типа текущего значения переменной (параметра и т. д.) типу, требуемому операцией, процедурой, вообще конструкцией, в которой находится эта переменная (параметр и т. д.). Например, если текущее значение индекса  $i$  в выражении  $a[i]$  выходит за границы массива  $a$ , то это типовая ошибка, если границы входят в понятие типа массива (как в Паскале), и runtime error в Фортране;
- динамический контроль требует от семантики языка наличия возможности продолжения выполнения программы после типовой ошибки (оставляя ее незамеченной) либо осуществления предварительного динамического типового контроля.

Объединяя в различной комбинации эти требования, можно получать языки с полным и неполным, с динамическим, статическим и смешанным контролем типов.

Таким образом, типизация языка и контроль типов являются важнейшими элементами аппарата защиты языка программирования. Они прогнозируют (объявляют) такие свойства поведения объектов, как принадлежность к определенному типу, указание области действия и ограничений на допустимые значения в определенных контекстах, и осуществляют контроль за соблюдением ограничений (в частности, управление реакцией на нарушение объявленного поведения).

### 7.3.3 Преобразование и передача типов

Если в языке программирования разрешается одновременное использование данных нескольких типов, например, в одном выражении или разных частях оператора присваивания, то вступает в действие механизм согласования типов [46]. Этот механизм может быть

полностью отключен, и тогда внутреннее представление объекта одного типа (отведенная ему память) предоставляется для обработки средствами другого типа. Например, печать указателя в виде целого числа с помощью вариантной записи Паскаля. Другой пример — бинарная запись данных в файл без преобразования значений в изображение. Для этого, например, в языке Си можно преобразовать указатель на переменную данного типа к типу **char\***, после чего записать байты непосредственно в файл. В Паскале запись двоичного кода без преобразования происходит всегда при обмене с *нетекстовым* файлом. Такая передача значения одного типа под юрисдикцию другого типа иногда так и называется — передача типа (*transferring type*). Если согласуемые типы в математическом или ином смысле родственные, то может иметь смысл определение соответствующих функций преобразования всех значений одного типа в допустимые значения другого типа и наоборот, задающих приведение исходного типа к целевому. Например, целый, вещественный и комплексный типы как математические множества (алгебраические структуры) естественно вкладываются друг в друга  $\mathbb{Z} \in \mathbb{R} \in \mathbb{C}$ . Но аппаратные (и программные) реализации этих типов не идеальны и существенно отличаются. В силу известных причин преобразование значений может осуществляться неточно, быть необратимым и требовать некоторого немалого чаще всего линейного времени (схема Горнера). То есть память преобразуемого к новому типу значения существенно меняется. Даже в таких строгих языках как Паскаль многие преобразования выполняются неявно и необходимо понимать происходящие при этом закулисные действия.

В отличие от преобразования значений согласованных типов *передача типа* — трактовка данных одного типа как значений другого типа [59]. В современных версиях Паскаля имя любого типа, как предопределенного, так и объявленного программистом, является также именем функции преобразования типа. Передача типа преобразует лишь тип (интерпретацию) данного как области памяти, а не само хранимое значение. Так, функция **float**, согласующая целый и вещественный типы, превращает целое значение аргумента в эквивалентное данное, хранимое как значение типа **real**: **float(3)** = 3.0. Функция передачи типа **real** присваивает этот тип области памяти с целым значением 3, не делая попыток превратить само это значение в эквивалентное значение типа **real**, т. е. значение **real(3)** почти наверняка не равно 3.0 !

### 7.3.4 Адресный тип

Для обеспечения процессов разнотипной интерпретации памяти необходимо применение родовых типов. *Родовые* типы данных уже заложены в конструкцию универсальных ЭВМ, память которых представляет собой слова, предназначенные для хранения данных произвольных типов [59]. Ослабляя ограничения строго типизированных языков, можно ввести родовой тип данных **word**, не предполагающий какого-либо особенного использования памяти, выделяемой для его объектов. Для типа **word** определены лишь операция присваивания и отношение равенства. Также, как и тип множество, тип **word** является внутримашинным, нетекстовым, непечатным. По этой причине в этом типе нет констант, изображающих конкретные значения.

Основное применение типа **word** связано с ослаблением контроля типов при передаче параметров. Любой формальный параметр типа **word** считается совместимым с любым фактическим параметром, занимающим ровно одно слово в памяти ЭВМ. В обычном употреблении тип **word** чужд другим типам, что уменьшает потенциальную опасность

его применения и локализует места его использования.

Другой, более универсальный способ ослабления типового контроля, основан на применении бестиповых, *родовых* указателей. В стандартном Паскале указатель может ссылаться на объекты не более, чем одного типа. В современных версиях Паскаля бестиповые указатели вводятся как `pointer to word` ( $\uparrow$  `word`), распространяя типовую свободу `word` на мир ссылок и указателей. Фактически тип родового указателя вводит в обиход языка высокого уровня адреса соответствующей ЭВМ, единообразно ссылающиеся на данные любых типов, размещенные в ячейках ее памяти. Поэтому он и получил название *адресного*.

*Адресный тип* имеет множеством значений диапазон допустимых адресов ЭВМ, быть может суженный до сегмента памяти процесса в мультипрограммных системах с защитой памяти. Объекты адресного типа совместимы по присваиванию и сравнению с любыми ссылочными переменными, а формальные параметры — с любыми ссылочными фактическими. Кроме того, адресный тип естественно совместим с целым и к нему применимы арифметические операции, что открывает в языке типа Паскаль набор гибких, но весьма опасных средств свойственной языку Си адресной арифметики. Адресный тип полностью устраняет контроль типов, выполняемый компилятором, и его следует употреблять только для разработки родовых модулей низкого уровня.

### 7.3.5 Родовые модули

Понятие родового модуля противоречит строгой типизации данных. Родовой модуль, реализующий стек, способен вталкивать и выталкивать элементы различных типов данных [59]. Один программист может импортировать этот модуль и использовать его в качестве стека целых значений, а другой — применять его как стек текстовых строк. Однако в Паскале целые числа и строки считаются несовместимыми типами.

Правильность типов данных для операций обычно контролируется. Если же контроль типов ослабляется, возникает нарушение строгой типизации данных. Для большинства родовых операций характерна та или иная форма нарушения строгой типизации. Например, можно хранить в целочисленном стеке коды литер, допустив легкое и полностью контролируемое нарушение строгой типизации данных [21].

В качестве примера ограниченно родового модуля часто приводят родовую очередь скалярных значений, реализуемую с помощью типа `word` как очередь слов [59]. `array of word` без указания индексных границ устраниет все виды контроля типов и допустим лишь в качестве типа формальных параметров, которому может соответствовать фактический параметр любого типа и длины.

### 7.3.6 Полиморфизм

*Параметризованные* или *полиморфные* типы — это один из аспектов общего понятия полиморфизма (многоликости вообще и многотипности в частности) [46]. Понятие полиморфной операции, т. е. операции, у которой хотя бы один аргумент допускает значения не одного, а нескольких типов, встречается почти в любом языке программирования, начиная с Фортрана, где операция `+` может применяться как к вещественным значениям, так и к целым [21]. Тоже можно сказать и о присваивании: один и тот же синтаксический объект `:=` перегружен и интерпретируется по-разному в зависимости от контекста.

```
var i, j: integer;
var x, y: real;
```

```
i := j;
x := y;
```

загружая разнотипные регистры машины фон Неймана.

От абстракции имен относительно конкретных значений можно перейти к абстрагированию конструкций по отношению к имени типа. Например, можно определить абстрактную по отношению к типу или полиморфную процедуру `swap`:

```
procedure swap (T : type, var p, q : T);
var t : T;
begin
  t := p;
  p := q;
  q := t
end;
```

Здесь в отличие от примера со знаком `:=` внутри процедуры `swap` смысл присваиваний неизвестен, поскольку тип `T` не фиксируется, и мы имеем дело с типовой абстракцией. Абстракция заключается уже в неопределенности типа, и может стать явной, как это сделано с помощью параметра-типа в процедуре `swap`, вызов которой имеет вид

```
swap(integer, x, y);
```

При компиляции такой процедуры надо запланировать передачу набора атрибутов типа (операций, отношений, ...) в процедуру например в виде вектора ссылок или как множество ссылок на процедуры (конкретных значений процедурного типа).

Уже в языке Алгол 68 можно описывать полиморфные процедуры или операции, называемые также родовыми или семействами (*generic*). Например, можно ввести процедуру `add(x, y)`, которая будет складывать либо целые числа, либо логические значения, либо векторы в зависимости от типов фактических параметров. Пользу такого вида полиморфизма обычно видят в удобстве иметь одно обозначение для нескольких вещей. Но как отметил Д. Грис, объединять под одним именем несколько операций (процедур) полезно тогда, когда они обладают существенно одинаковыми свойствами (это будет хороший полиморфизм в отличии от плохого, примером которого может служить процедура `add(x, y)`). Например, хорошо иметь полиморфную процедуру `sort`, упорядочивающую любые массивы, для элементов которых определено отношение порядка `<`. Спецификация такой процедуры могла бы иметь вид

```
sort( a: array of t)
```

где `t` — параметр для типа элемента массива `a`, значением которого может быть любой тип, упорядоченный каким-то отношением `<`. Для разных типов-значений параметра `t` алгоритм сортировки будет одним и тем же, использующим только свойства отношения `<`.

Средства абстракции (инкапсуляции) данных представляют хорошую возможность введения полиморфных процедур с параметрами-типами. Процедуру `sort()` можно включить как одну из операций в параметризованный абстрактный тип данных и описать

как параметр АТД типа **type** в заголовке его спецификации. Эта возможность появилась более 20 лет назад в экспериментальных языках CLU и Alphard, на которых отрабатывались идеи и методы объектно-ориентированного программирования, и была поддержана существенной интерпретативностью систем программирования на этих языках.

Определения типов с параметрами — это, по существу, схемы определения классов подобных друг другу типов. Аналогично конструкции (процедуры и т. д.), содержащие тип как параметр, — это схемы подобных конструкций. Введение той или иной формы полиморфизма, очевидно, повышает уровень абстракции языка. Однако при их реализации возникают дополнительные трудности. Отметим в связи с этим наличие эффективно реализуемой формы полиморфизма в языке Ада.

Существует внутренняя связь между полиморфизмом и абстрактными типами данных: абстрактная программа рассматривается как полиморфная процедура, а абстрактный тип — как ее тип-параметр. Выбор конкретного типа-представления означает выбор значения для этого абстрактного типа-параметра.

Современные языки программирования имеют средства для *обобщенного* (*generic!*) программирования. Самым распространенным в настоящее время языком, поддерживающим обобщенное программирование, является язык C++. Обобщенное программирование в C++ реализовано в виде набора алгоритмов, являющихся, наряду с итераторами и контейнерами, одним из трех китов, на котором основана библиотека STL.

Так, для реализации полиморфных функций, зависящих от типа передаваемых аргументов, в C++ используется понятие *шаблона* функции. Шаблон — это некоторая подпрограмма (или итератор, или контейнер), предъявляющая определенные требования к типу своих аргументов. Рассмотрим пример с функцией (обобщенным алгоритмом!) *swap()*:

```
template<class T>
void inline swap(T& lhs, T& rhs)
{
    T t = lhs;
    lhs = rhs;
    rhs = t;
}
```

Единственным требованием, предъявляемым к типу аргумента, является наличие у типа оператора присваивания *operator = ()*.

Вызов функции-шаблона выглядит так:

```
int x = 5, y = 7;
double z = 10;

swap<int>(x, y); // Компилятор C++ сам определяет параметризацию шаблона
swap(x, z); // Ошибка — неоднозначность!
```

Рассмотрим еще один пример — функция *min()*:

```
template<class T>
T& inline min(T& lhs, T& rhs)
{
```

```

    return (lhs < rhs) ? lhs : rhs;
}

```

Эта функция предъявляет помимо наличия у типа  $T$   $operator =()$  еще и требование наличия  $operator <()$  для сравнения двух значений типа. Функции сортировки (особенно устойчивой) могут предъявлять дополнительные требования наличия  $operator ==()$  для проверки аргументов на равенство.

Главным достоинством такого подхода является полное отсутствие затрат на вызов виртуальных функций и вообще приведения к некоему «общему» типу Object, CObject, TObject и др., что только ускоряет работу программ.

### 7.3.7 Процедурный тип

Еще одним средством реализации родовых модулей является процедурный тип. Фактически процедурный тип представляет собой целый класс типов, отдельные типы которого есть множества глобально определенных процедур с идентичной спецификацией, включая и тип результата для функций [59]. Константами процедурных типов являются имена глобальных процедур, также разбитые на непересекающиеся классы равнозначности в соответствии с заголовками этих процедур. Таким образом, процедурный тип есть своеобразный вариант перечислимого (порядок можно задать через адресный и целый типы!). Переменные процедурных типов принимают значения из соответствующего спецификации заголовка множества процедур. Процедурную переменную можно сравнить с однотипной или присвоить ей допустимое значение другой переменной или константы того же типа. Но, главное, ее можно вызвать, как и любую другую обычную процедуру. Согласно этому делению все процедуры без параметров совместимы и также составляют единый тип.

Поскольку для вызова процедуры в конечном счете нужно знать ее адрес, его роль может играть ссылка на процедуру. Комбинация бестипового указателя и процедурных типов образует эффективную и опасную гремучую смесь, позволяющую, например, не покидая Паскаль-программы, построить в целочисленном массиве программу на машинном языке и тут же выполнить ее.

Заметим также, что процедурные типы в раздельно компилируемых модулях делают эти модули родовыми по отношению к процедурам, а не к типам данных.

Процедурный тип позволяет просто и систематически описывать более универсальные родовые модули с хранимыми процедурами, поскольку подстановка параметров процедурного типа существенно повышает параметризацию такого модуля. Можно также импортировать тип элемента очереди из модуля-посредника с заранее фиксированным именем, обеспечивая строгий контроль использования типов. Внимательный слушатель заметит, что это уже почти объектно-ориентированное программирование.

В Си и C++ таких тонких понятий, как процедурный тип, просто нет. Вместо этого предусмотрены указатели на функции — такие же, как на скалярные типы данных, с возможностью приведения к обобщенному указателю **void\***:

```

typedef double (*binary_function)(double, double); // Указатель на функцию
двух переменных

double f(double x, double y)
{
    return x + y;

```

```

}

binary_function pf = f;

double z0 = pf(1, 2); // Вызов функции через указатель
double z1 = (*pf)(3, 4); // Тоже вариант вызова

```

Указатели на функции очень распространены в системном программировании вообще и в WinAPI в частности, где они используются для задания оконных процедур, функций обратного вызова для обработки срабатывания таймера и др.

### 7.3.8 Элементы объектно-ориентированного программирования в расширениях языка Си

Для реализации полиморфизма в абстрактном типе данных очередь напишем дополнения к модулю Queues. Рассмотрим три подхода к решению проблемы.

#### 7.3.8.1 Полиморфизм с использованием бестиповых указателей

Полиморфная очередь может содержать объекты *разных* типов: TCircle (окружность), TTriangle (треугольник), TExecutable (выполнимый объект, вызывающий процедуру саморисования).

```

/* Вспомогательный тип для идентификации объекта */
typedef enum {idNone, idCircle, idTriangle, idExecutable} objectID;

/* Тип точки, используемый в описании геометрических фигур */
typedef struct {
    float x;
    float y;
} TPoint;

typedef struct {
    objectID id; // Идентификатор должен идти первым!
    TPoint center;
    float radius;
} TCircle, *PCircle;

typedef struct {
    objectID id; // Идентификатор должен идти первым!
    TPoint A, B, C;
} TTriangle, *PTriangle;

typedef void (*PProcedure)(void);

typedef struct {
    objectID id; // Идентификатор должен идти первым!
    PProcedure ProcPtr;
}

```

```

} TExecutable, *PExecutable;

/* Запись, необходимая для идентификации реального объекта в памяти */
typedef struct {
    objectID id; // Как и во всех предыдущих типах — идет первым
                  // для совпадения оффсета.
} TIDRecord, *PIDRecord;

```

Абстрактный указатель должен иметь размер адреса конкретной архитектуры. Например, возможно использование указателя (*char* ↑, *integer* ↑), в этом случае размер определяется автоматически; или использование типа соответствующего размера:

```
#include <stddef.h>
```

```

typedef ptrdiff_t word;
typedef *word Pointer;

```

Тип данных для элемента очереди TValue определяется как абстрактный указатель:

```
typedef void* TValue;
```

Везде, где для печати элемента очереди вызывается процедура printf, необходимо заменить ее на вызов процедуры Draw, которая, в свою очередь, вызывает конкретные методы печати для соответствующих типов элементов очереди:

```
/* Особенности отрисовки на экране или вывода на экран определяются конкретной
реализацией */
```

```

void DrawCircle (TCircle T)
{
    printf("CIRCLE(X=%f,?Y=%f,?R=%f)|n", T.Center.x, T.Center.y, T.Radius);
}
```

```

void DrawTriangle (TTriangle T)
{
    printf("TRIANGLE(A=(%f, %f), ", T.A.x, T.A.y);
    printf("B=(%f, %f), ", T.B.x, T.B.y);
    printf("C=(%f, %f))|n", T.C.x, T.C.y);
}
```

```

void DrawExecutable (TExecutable T)
{
```

```
/* Выполнение процедуры T.ProcPtr поуказателю T.ProcPtr. Эта процедура
самостоятельно визуализирует встроенный объект */
```

```

if (T.ProcPtr != NULL)
    T.ProcPtr();
}
```

```
void Draw (TValue P)
```

```
{
/* Для определения типа реального объекта используется приведение указателя на
него к идентифицирующему типу PIDRecord */
```

```

switch (( (PIDRecord)P )->id)
{
    case idCircle :
        DrawCircle(*( (PCircle)P ));
        break;
    case idTriangle:
        DrawTriangle(*( (PTriangle) P) );
        break;
    case idExecutable:
        DrawExecutable(*( (PExecutable) P) );
        break;
}

```

Для создания нового объекта и помещения его в очередь используются следующие функции-конструкторы, сводящие все к стандартной процедуре **new**:

```

TValue NewCircle (float x, float y, float r)
{
    PCircle Z;
    Z = malloc(sizeof(TCircle));
    Z->center.x = x;
    Z->center.y = y;
    Z->radius = r;
    Z->id = idCircle; /* Приведение конкретного типа окружность к типу
                        универсальных элементов очереди TValue (* word) */
    return (TValue) Z;
}

```

```

TValue NewTriangle (float ax, float ay, float bx, float by, float cx, float cy)
{
    PTriangle Z;
    Z = malloc(sizeof(PTriangle));
    Z->A.x = ax; Z->A.y = ay;
    Z->B.x = bx; Z->B.y = by;
    Z->C.x = cx; Z->C.y = cy;
    Z->id = idTriangle; /* Здесь треугольниковый элемент приводится к
                           тому же универсальному типу элементов очереди */
    return (TValue) Z;
}

```

```

TValue NewExecutable (PProcedure T)
{
    PExecutable Z;
    Z = malloc(sizeof(TExecutable));
    Z->ProcPtr = T;
    Z->id = idExecutable; /* Здесь приведение также возможно, поскольку
                           все указатели в Extended Pascal родственны */
}

```

```
    return (TValue) Z;
}
```

Для правильного уничтожения объектов очереди необходимо использовать процедуру `DestroyObject`, а не стандартную процедуру `free`:

```
void DestroyObject (TValue P)
{
    switch (((PIDRecord) P)->id)
    {
        /* Для определения типа реального объекта используется приведение
        указателя на него к идентифицирующему типу PIDRecord */
        case idCircle:
            free ((PCircle) P);
            break;
        case idTriangle:
            free ((PTriangle) P);
            break;
        case idExecutable:
            free ((PExecutable) P);
            break;
    }
}

void ExampleDraw(void)
{
    /* Это единственная процедура нашей программы, вляющейся объектом, а не
    субъектом. Это константа простейшего процедурного типа, которая может быть
    сослана и разыменована. Чтобы вызвать процедурную переменную, надо
    употребить ее имя (процедурной переменной, а не конкретной процедуры!) с
    фактическими параметрами вызова и с точкой с запятой. Если процедурная
    переменная доступна по ссылке, ее надо разыменовать */
    printf("EXECUTING\n");
}
```

Рассмотрим пример добавления нескольких разнородных элементов в очередь:

```
int main() {
    Queue queue;
    Create(&queue);

    Push(&queue, NewCircle(10,10,5));
    Push(&queue, NewTriangle(1,2,3,4,5,6));
    Push(&queue, NewExecutable(ExampleDraw));

    while (!Empty(&queue)) {
        Draw(Top(&queue)); // Здесь функция Draw сама определит тип
    }
}
```

```

    // возвращаемого элемента очереди и вызовет
    // соответствующую функцию печати на экран.
    Pop(&queue);
}
}

```

### 7.3.8.2 Реализация с использованием указателей процедурного типа

В предыдущем варианте для реализации действий с элементами очереди использовались функции (например, Draw), выполняющие определенные действия над объектом в зависимости от типа объекта. Такой подход неудобен тем, что при добавлении нового объекта приходится модифицировать все соответствующие функции (Draw, DestroyObject и др.). Кроме того, функции, присущие объектам, концептуально отделяются от самих объектов. В данном примере процедуры, присущие объектам, задаются в самих объектах как поля процедурного типа. Кроме того, добавляются дополнительные свойства объектов, необходимые для реализации сортировки. Необходимость в идентификаторе objectID отпадает:

```

// Тип точки, используемый в описании геометрических фигур
typedef struct {
    float x;
    float y;
} TPoint;

// Результат сравнения ключей.
typedef enum {Less, Equal, More, NotEqual, NotComp} CmpType;

/* Тип TObject заменяет используемую в предыдущем примере TIDRecord
он содержит поля?ссылки на родовые процедуры, которые должны
располагаться в том же порядке, что и в самих объектах */

typedef struct TObject_ {
    void (*Draw)(struct TObject_*); // Функция отрисовки на первой позиции.
    void (*Compare)(struct TObject_*, struct TObject_*); // Функция сравнения на
        // второй позиции.
} TObject, *PObject;

typedef CmpType (*PCmpFunction)(PObject U, PObject V);

// Тип указателя на функцию, будет во всех типах элементов очереди.
typedef void (*PProcedure)(PObject);

// Тип указатель на функция для хранения в TExecutable.
typedef void (*PDrawProc)(void);

// Теперь – особое внимание на абсолютный порядок полей в структурах.

```

```
// Указатель на функцию отрисовки ВСЕГДА первый, функция сравнения всегда
// вторая, в соответствии с их расположением в TObject.
```

```
typedef struct {
    PProcedure Draw; // Функция отрисовки на первой позиции.
    PCmpFunction Compare; // Функция сравнения на второй позиции.
    TPoint Center;
    float Radius;
} TCircle, *PCircle;
```

```
typedef struct {
    PProcedure Draw; // Функция отрисовки на первой позиции.
    PCmpFunction Compare; // Функция сравнения на второй позиции.
    TPoint A, B, C;
} TTriangle, *PTriangle;
```

```
typedef struct {
    PProcedure Draw; // Функция отрисовки на первой позиции.
    PCmpFunction Compare; // Функция сравнения на второй позиции.
    PDrawProc ProcPtr;
} TExecutable, *PExecutable;
```

Тип данных для элемента очереди TValue теперь определяется как указатель на абстрактный объект TObject. Необходимость в бестиповом указателе отпадает:

```
typedef PObject TValue; // Тип элемента очереди – указатель на TObject.
```

Везде, где для печати элемента очереди вызывается процедура printf, необходимо заменить ее на вызов процедуры, на которую ссылается поле Draw конкретной записи. Начальные значения этому полю присваиваются в процедурах инициализации.

```
/* Особенности отрисовки или вывода объекта на экране определяются конкретной
реализацией. В данном примере мы просто печатаем геометрическое описание.*/
void DrawCircle (PObject P)
{
    PCircle T;
    T = (PCircle) P;
    printf("CIRCLE(X=%f, Y=%f, R=%f)|n", T->Center.x, T->Center.y, T->Radius);
}

void DrawTriangle (PObject P)
{
    PTriangle T;
    T = (PTriangle) P;
    printf("TRIANGLE(A=(%f, %f), ", T->A.x, T->A.y);
    printf("B=(%f, %f), ", T->B.x, T->B.y);
    printf("C=(%f, %f))|n", T->C.x, T->C.y);
}
```

```

void DrawExecutable (PObject P)
{
    if (((PExecutable) P)->ProcPtr != NULL)
        (((PExecutable) P)->ProcPtr) ();
}

```

Для печати (отрисовки) объекта следует использовать конструкцию:  $P \uparrow .Draw \uparrow (P)$ , где  $P$  — указатель на объект типа  $PObject$ . К сожалению, каждый раз приходится передавать в процедуру  $Draw()$  указатель на сам объект, т. к. процедура  $Draw()$  не может определить, по указателю из какой конкретной записи она была вызвана. Объектно-ориентированный подход, рассмотренный в следующем примере, избавляет нас от такой необходимости.

Определение процедур сравнения:

```

/* Сравнение основано на том, что у каждого объекта есть указатель на функцию
Draw и у объектов одинакового исходного типа, указатели на эту функцию будут
одинаковыми. */

//проверка эквивалентности типов
bool EqType (PObject U, PObject V)
{
    /* Если указатели на родовые процедуры совпадают ? типы одинаковы! */
    return U->Draw == V->Draw;
}

/* Соответствует TCmpFunc */
CmpType CmpCircle (PObject U, PObject V)
{
    if (!EqType(U, V)) // Сравнение совпадания исходных типов.
        return NotComp;
    else // Сравнение радиусов. (Чей радиус больше – та и фигура больше)
    {
        if (((PCircle) U)->Radius == ((PCircle) V)->Radius)
            return Equal;
        if (((PCircle) U)->Radius < ((PCircle) V)->Radius)
            return Less;
        if (((PCircle) U)->Radius > ((PCircle) V)->Radius)
            return More;
    }
}

/* Расстояние между точками */
float Dist (TPoint A, TPoint B)
{
    return sqrt(pow(A.x-B.x, 2)+pow(A.y-B.y, 2));
}

```

```

}

/* Сумма длин сторон ? критерий сравнения треугольников */
float Sum (TTriangle U)
{
    return Dist(U.A, U.B) + Dist(U.A, U.C) + Dist(U.B, U.C);
}

```

```

/* Соответствует TCmpFunc */
CmpType CmpTriangle (PObject U, PObject V)
{
    if (!EqType(U, V))
        return NotComp;
    else
    {
        if (Sum(*((PTriangle) U)) == Sum(*((PTriangle) V)))
            return Equal;
        if (Sum(*((PTriangle) U)) < Sum(*((PTriangle) V)))
            return Less;
        if (Sum(*((PTriangle) U)) > Sum(*((PTriangle) V)))
            return More;
    }
}

```

```

CmpType CmpExecutable (PObject U, PObject V)
{
    if (!EqType(U, V))
        return NotComp;
    else
        if (((PExecutable) U)->ProcPtr == ((PExecutable) V)->ProcPtr)
            return Equal;
        else
            return NotEqual;
}

```

Для создания нового объекта и помещения его в очередь используются следующие функции-конструкторы:

```

TValue NewCircle (float x, float y, float r)
{
    PCircle Z;
    Z = malloc(sizeof(TCircle));
    Z->Center.x = x;
    Z->Center.y = y;
    Z->Radius = r;
    Z->Draw = DrawCircle;
    Z->Compare = CmpCircle;
}

```

```
    return (TValue) Z; //приведение типа к типу элементов очереди
}
```

```
TValue NewTriangle (float ax, float ay, float bx, float by, float cx, float cy)
{
    PTriangle Z;
    Z = malloc(sizeof(TTriangle));
    Z->A.x = ax;
    Z->A.y = ay;
    Z->B.x = bx;
    Z->B.y = by;
    Z->C.x = cx;
    Z->C.y = cy;
    Z->Draw = DrawTriangle;
    Z->Compare = CmpTriangle;
    return (TValue) Z;
}
```

```
TValue NewExecutable (PProcedure T)
{
    PExecutable Z;
    Z = malloc(sizeof(TExecutable));
    Z->ProcPtr = (PDrawProc)T;
    Z->Draw = DrawExecutable;
    Z->Compare = CmpExecutable;
    return (TValue) Z;
}
```

```
void ExampleDraw()
{
    printf("EXECUTING\n");
}
```

Программа добавления разнородных объектов в очередь; пример идентичен предыдущему, хотя очередь реализована совершенно по-другому:

```
int main() {
    Queue queue;
    Create(&queue);
    Push(&queue, NewCircle(10,10,5));
    Push(&queue, NewTriangle(1,2,3,4,5,6));
    Push(&queue, NewExecutable(ExampleDraw));

    while (!Empty(&queue)) {
        Top(&queue)->Draw(Top(&queue));
        Pop(&queue);
    }
}
```

}

## 7.4 Объектно-ориентированное программирование

### 7.4.1 Наследование

До сих пор мы интересовались созданием программ «с нуля», не учитывая потребность использовать имеющиеся программные услуги при создании новых. Соответствующее свойство языка принято называть *развиваемостью* [21]. В частности, к развивающейся относят любые средства расширения языка: процедурность, модульность, наследуемость и объектная ориентация. Последние две фундаментальные концепции нам предстоит еще рассмотреть.

Работающая программа — это материализованный интеллект и труд высококвалифицированных людей, который не только дорого стоит, но и содержит в себе элемент творчества. Поэтому идеал развивающейся является голубой мечтой программиста.

На заре программирования развивающаяся выражалась в библиотеках стандартных программ и средствах модуляризации. Это были средства заимствования, не защищенные от несанкционированного использования и разрушения. Пример: директива `#include`.

Идеал развивающейся должен гарантировать защиту программных услуг: защиту авторского права разработчика услуг и защиту потребителя от неправильного использования услуг, но не настолько тотальную, чтобы этим всем нельзя было пользоваться.

В настоящее время развивающаяся считается критичным свойством языка программирования, поскольку уже разработано колоссальное количество программных услуг во всех областях человеческой деятельности.

Основными аспектами развивающейся являются:

- Модульность. Она обеспечивает развивающуюся за счет фиксации сопряжения (интерфейса) между создателем и потребителем услуг.
- Стандартизация. В дополнение к модульности, устраняет нерациональное разнообразие сопряжений.
- Наследуемость. Подразумевает гибкий аппарат развития, заимствования и защиты, действующий на уровне практически произвольных языковых объектов, а не только на уровне заранее предусмотренных модулей.

Такой уровень гибкости позволяет легко приспособить программу к обслуживанию объектов, тип которых неизвестен не только при ее создании, но и при трансляции (с гарантией статического контроля типов).

Впервые развитая наследуемость появилась в Симуле-67 в соответствующем году, предвосхитив современное понимание идеала.

Итак, модульность обеспечивает упаковку программных услуг в модули-контейнеры, стандартизация — доставку упакованных услуг потребителю в работоспособном состоянии, а наследуемость — изготовление контейнера новых услуг с минимальными затратами, с минимальным риском и в рамках законности.

Уточним идеал наследуемости с учетом типизации языковых объектов: должно быть дозволено:

- Определять новый тип, наследующий те и только те атрибуты исходного типа, которые желательны.
- Пополнять перечни атрибутов нового типа по сравнению с атрибутами исходного типа (обогащение типов (возможность вводить дополнительные поля при объявлении производного типа) и виртуальные операции, заменяющие старые операции при действиях с обогащенными значениями, даже в старых программах).
- Гарантировать применимость сохраняемых операций исходного типа к объектам нового типа.

Основные понятия и неформальные аксиомы наследования:

- Основные понятия:
  - Отношение наследования между родителем и наследником
  - Атрибуты наследования
  - Накопление атрибутов у наследника по отношению к родителю
  - Типы объектов и экземпляры (объектов) определенных типов
- Отношение наследования определяется для типов, а не для экземпляров (в живой природе — наоборот, индивиды-наследники наследуют свойства индивидов-родителей).
- Наследник обладает всеми атрибутами родителей, обратное неверно.
- Право участвовать в операциях определенного класса — это атрибут наследования! Таким образом, наследник имеет право замещать родителя в любых таких операциях. Определить этот класс иногда бывает непросто: даже присваивание в неполнообъектных языках для обогащенных объектов может замещать источник, но не получатель.
- Все экземпляры (объекты) одного типа обладают идентичными атрибутами (но не их значениями!). Следствие: индивидуальные свойства объектов (определенные, в частности, значениями атрибутов) не наследуются и по наследству не передаются.
- Наличие наследников не влияет на атрибуты родителей. Следствие: свойство иметь наследников не считается атрибутом и, стало быть, не наследуются.
- Атрибуты могут быть абстрактными (виртуальными), и тем самым требовать конкретизации (настройки на контекст перед использованием или в процессе использования), или конкретными, и тем самым такой настройки не требовать.
- Результат конкретизации абстрактных атрибутов в общем случае определяется тем контекстом, где объекты создаются. Следствие: поскольку в этот контекст входят не только типы, но и конкретные объекты (экземпляры типов), то конкретные значения абстрактных атрибутов могут зависеть от свойств конкретных объектов контекста, а не только от их типов.

Настройку на контекст естественно выполнять при создании (инициализации) объектов. Например, объект-очередник при создании получает конкретные значения атрибута, указывающего на стоящего впереди объекта.

- Накопление в некоторых языках может состоять как из абстрактных, так и (в беднообъектных языках) из конкретных атрибутов. Примерами абстрактных атрибутов являются атрибуты- поля записей и виртуальные операции.

Преимущества развитой наследуемости:

- Гармония открытости и защиты. Принцип защиты авторского права реализуется в естественной гармонии с принципом открытой системы. Пользователь открытой системы не только получает доступ к декларированным возможностям системы, но и может перестраивать систему для своих нужд способом, не предусмотренным явно ее создателем.
- Поддерживаемая технология развития программной системы (пошаговая модификация работающей основы) способствует оптимизации и структуризации мышления, программирования, памяти. Более того, можно применить концепцию расслоенного программирования (пошаговое наращивание новых слоев работающей версии программы от минимальной работоспособной основы до цели — разветвленной системы услуг). Эта концепция отечественного программиста А. Л. Фуксмана на западе получила название *быстрого прототипирования* (rapid prototyping).
- Поддерживаемый стиль мышления адекватен естественному развитию от простого к сложному, от общего — к частному, от единого корня — к разнообразию, в отличие от классического структурного программирования, подразумевающего лишь пошаговую детализацию сверху вниз.
- Говоря более конкретно, развитое наследование обеспечивает расширяемость объектов, типов и операций с защитой авторского права и с гарантией сохранности старых программ. Можно воспользоваться программой, развивать ее, но нельзя украдь секреты фирмы, если они скрыты автором.

В заключение подчеркнем существенное отличие изложенного понятия наследования от аналогичного биологического. В последнем случае не только сами типы, но и конкретные экземпляры объектов таких родительских типов, как животные, кошки, собаки и т.п. реально существуют лишь как некоторые абстракции, представленные, например, совокупностью генов или изменчивым набором ныне живущих особей (обладающих, конечно, неисчислимым множеством свойств никак не охватываемых соответствующим типом). А в информатике и типы, и экземпляры объектов представлены вполне реальными разрядами, записями, фреймами и т.п., соответственно и наследование пока представлено как определение и обработка определений типов, а не как результат жизнедеятельности объектов. Однако нет оснований полагать, что так будет всегда.

Математической моделью идеального наследования является гомоморфизм [21].

#### 7.4.2 Построение объектной технологии

Активные объекты (данные и процедуры), присутствуя в языках программирования, были лишены существенных прав обычных данных, а обычные данные не имели свойств активных объектов, т. е. нельзя было оперировать с процедурами и выполнять данные. Поэтому необходимо освободиться от неестественных ограничений и подняться на следующий

уровень абстракции. Введем концепцию языкового объекта, обобщающую представление об активных и пассивных объектах. Это фактически означает, что любой объект можно рассматривать как потенциально активный, если явно не оговорено обратное. Эта цель оказалась достигнутой на современном этапе развития программирования в рамках объектно-ориентированного программирования [59].

Основными идеями объектно-ориентированного программирования являются:

- Программный модуль является нормальным языковым объектом и имеет вполне определенный тип.
- Активный объект может предоставлять услуги как по извлечению, так и по преобразованию значений собственных атрибутов.
- Активные объекты могут обмениваться сообщениями и самостоятельно реагировать на них как на запросы услуг.

Свойства объектной ориентации:

- Новый уровень абстракции. Объектная ориентация выводит на новый уровень абстракции и обновляет фундаментальные концепции языка: управления, развития, защиты, модульности, динамизма, спецификации, технологии и сближает проблематику языков программирования с такой главой информатики, как искусственный интеллект. Например, понятие объекта сходно с понятием фрейма, одним из основных понятий в теории представления знаний.
- Интеграция понятий и средств информатики. Объектно-ориентированное программирование знаменует очередной этап сближения (интеграции) понятий и средств информатики, характерный для нее в последние годы.
- Целостность языков. Распространенные объектные языки естественным образом выросли из своих предшественников (Паскаль, Си), не нарушив естественную структуру языка, и эволюционным образом вводя фон Неймановского программиста в светлое будущее полнообъектных сред.

#### **7.4.3 Объектно-ориентированное программирование и структурное программирование**

Объектно-ориентированное программирование базируется на структурах (типах) данных и лучше всего работает, когда основная сложность построения алгоритма заключена в выборе организации данных. Структурное программирование основано на нисходящем применении ограниченного набора простых структур управления и дает хорошие результаты при сложном управлении и простых структурах данных.

Преимущество нисходящего проектирования в том, что на каждом этапе разработки алгоритм определен полностью. Это позволяет программисту понять роль каждого фрагмента во всем программном комплексе. При объектно-ориентированном программировании основное внимание уделяется объектам. И порой бывает не вполне ясно, как объекты и операции взаимодействуют при решении задачи.

Поскольку в объектно-ориентированном подходе выделено понятие операции, то обычно это естественным образом приводит к прекрасной модульной структуре программы. Структурный подход весьма слабо поддерживает процесс декомпозиции программы на процедуры, а для больших программ проблема их декомпозиции на модули очень непроста.

При объектно-ориентированном подходе также упрощается использование модулей заимствования. После того как выделены объекты и операции над ними, программисту не составляет труда определить дальнейшую полезность разработанного программного обеспечения.

В результате объектно-ориентированного программирования основной алгоритм легко читается и понимается, так как запрограммирован в терминах процедур, похожих на абстрактные операции. Операции короткие и понятные, поскольку связаны с независимыми четко определенными задачами.

Упрощение внесения изменений — еще одно достоинство объектно-ориентированного программирования. Операция проще идентифицируется и изменяется, если она реализована в виде отдельной процедуры.

В заключение отметим, что наибольший эффект дает совокупное использование обоих методов: объекты и операции должны выбираться в рамках объектно-ориентированного подхода, а реализацию операций лучше всего проводить сверху-вниз.

Удобнее первые шаги разработки программы провести в соответствии с объектно-ориентированным подходом, выполняя разбиение на модули.

Если подходящая структура данных без исследования алгоритма неясна, надо начать нисходящее проектирование.

Если составление алгоритмов без предварительного выбора структуры данных не получается, необходимо провести объектно-ориентированное проектирование.

После каждого уровня детализации полезно выделить в алгоритме характерные объекты.

Каждую операцию можно рассматривать как отдельную задачу для нисходящего проектирования.

При выборе объектов следует использовать заимствуемые компоненты.

При проектировании объектов и операций надо опираться на абстрактные типы данных.

#### 7.4.4 ООП в фон Неймановских языках

Н. Вирт развивал Паскаль в сторону объектной технологии (Модула, Модула-2, Оберон, ...) — см., например, [59]. Фирма Borland уже в Turbo Pascal 5.5 предоставила возможность объектно-ориентированного программирования на Паскале. Эти средства стали фактическим стандартом и, в частности, вошли в GNU Pascal. Однако, официального стандарта объектно-ориентированного Паскаля до сих пор нет.

##### 7.4.4.1 Полиморфная очередь с использованием механизмов ООП

Очередь реализуется из объектов типов TCircle (окружность), TTriangle (треугольник), TExecutable (выполнимый объект, вызывающий процедуру саморисования), которые являются наследниками TObject: Очередь реализуется из объектов типов TCircle (окруж-

ность), TTriangle (треугольник), TExecutable (выполняемый объект, вызывающий процедуру саморисования), которые являются наследниками TObject:

```
/* Процедуры, которые должны вызываться по ссылке на TObject, объявляются виртуальными. При обращении к такой процедуре как к члену базового класса будет вызвана процедура класса—наследника. Экземпляров абстрактного класса не бывает по определению, возможны только указатели на них, через которые можно вызвать общие для разнотипных потомков функции.*/
typedef struct {

    void (*Draw)(void*);
    void (*Done)(void*);

} TObject, *PObject;

/* У конкретных классов конструктор будет описан. Конструктор абстрактного класса не имеет смысла, поскольку абстрактные классы используются как исходная спецификация для конкретных классов.*/
void TPointInit(TPoint *this, float x, float y);

/* Деструктор объявлен виртуальным, поскольку уничтожение контейнера—очереди потребует поочередного уничтожения всех ее элементов, что приводит к вызову их деструкторов. Поскольку тип элемента очереди—абстрактный, то вызов абстрактного деструктора будет заменен вызовом конкретного.*/
void deleteTObject(TObject *this);

/* Тип точки, используемый в описании геометрических фигур.*/
typedef struct TPoint {
    TObject Parent; // Эквивалентно
    //void (*Draw)(void*);
    //void (*Done)(void*);

    float x;
    float y;
} TPoint, *PPoint;

/* У нового точкового типа есть своя процедура визуализации.*/
void TPointDraw(void *this);

/* Может потребоваться для освобождения памяти, занятой членами класса в куче, различных ресурсов компьютера и ОС, как то объекты GDI, порты, сокеты...*/
void TPointDone(void *this);

/* Для сохранения концептуальной совместимости с предыдущими примерами TCircle наследуется от TObject. Для полной концептуальной законченности необходимо наследовать TCircle и Tpoint от TLocation, который в свою очередь наследуется от
```

```
TObject вместе с TExecutable.* /  
  
typedef struct TCircle {  
  
    TObject Parent;  
    TPoint *center;  
    float radius;  
  
} TCircle, *PCircle;
```

```
typedef struct TTriangle {  
  
    TObject Parent;  
    TPoint *A, *B, *C;  
  
} TTriangle, *PTriangle;
```

```
typedef void(*PProcedure)(void);  
  
typedef struct TExecutable {  
  
    TObject Parent;  
    PProcedure ProcPtr;  
  
} TExecutable, *PExecutable;
```

Тип данных для элемента очереди TValue определяется как указатель на объект типа TObject, что обеспечивает совместимость с указателями на все его потомки:

```
typedef PObject TValue;
```

Для создания нового объекта и помещения его в очередь используются следующие функции-конструкторы, сводящие все к стандартной процедуре **new:bfnew**:

```
TPoint* newTPoint(float x, float y) {  
    TPoint *newElement = (TPoint*) malloc(sizeof(TPoint));  
    TPointInit(newElement, x, y);  
    return newElement;  
}
```

```
TCircle* newTCircle(float x, float y, float radius) {  
    TCircle *newElement = (TCircle*) malloc(sizeof(TCircle));  
    TCircleInit(newElement, x, y, radius);  
    return newElement;  
}
```

```
TTriangle* newTTriangle(float ax, float ay, float bx, float by, float cx, float cy) {  
    TTriangle *newElement = (TTriangle*) malloc(sizeof(TTriangle));  
    TTriangleInit(newElement, ax, ay, bx, by, cx, cy);
```

```

    return newElement;
}

TExecutable* newTExecutable(PPprocedure P) {
    TExecutable *newElement = (TExecutable*) malloc(sizeof(TExecutable));
    TExecutableInit(newElement, P);
    return newElement;
}

```

Везде, где для печати элемента очереди вызывается процедура `writeln`, необходимо заменить ее на вызов метода `Draw`, например:

$$printf(P) \leftarrow P \uparrow .Draw(); \{ \text{где } P : PObject \}$$

При этом автоматически вызывается метод `Draw()`, соответствующий объекту, указатель на который содержится в данном элементе очереди.

```

/* Процедура — метод Draw для объекта TPoint */
void TPointDraw(void *_this) {
    TPoint *this = (TPoint*) _this;
    printf("X = %f, Y = %f", this->x, this->y);
}

/* Процедура — метод Draw для объекта TCircle */
void TCircleDraw(void *_this) {
    TCircle *this = (TCircle*) _this;
    printf("CIRCLE()");
    this->center->Parent.Draw(this->center);
    printf(", R=%f)\n", this->radius);
}

/* Процедура — метод Draw для объекта TTriangle */
void TTriangleDraw(void *_this) {
    TTriangle *this = (TTriangle*) _this;
    printf("TRIANGLE()");
    this->A->Parent.Draw(this->A);
    printf(", ");
    this->B->Parent.Draw(this->B);
    printf(", ");
    this->C->Parent.Draw(this->C);
    printf("\n");
}

void TExecutableDraw(void *_this) {
    TExecutable *this = (TExecutable*)_this;
    if (this->ProcPtr != NULL)
        this->ProcPtr();
    /* Выполнение процедуры по указателю */
}

```

```
}
```

```
void ExpamleDraw() {
    printf("PERFORMING\n");
}
```

Для создания нового объекта и помещения его в используется функции-конструкторы:

```
void TPointInit(TPoint *this, float x, float y) {
    this->Parent.Draw = TPointDraw;
    this->Parent.Done = TPointDone;

    this->x = x;
    this->y = y;
}
```

```
void TCircleInit(TCircle *this, float x, float y, float radius) {
    this->Parent.Draw = TCircleDraw;
    this->Parent.Done = TCircleDone;

    this->center = newTPoint(x, y);
    /* Порождение точки—центра окружности! */
    this->radius = radius;
}
```

```
void TTriangleInit(TTriangle *this, float ax, float ay, float bx, float by, float cx,
    float cy) {
    this->Parent.Draw = TTriangleDraw;
    this->Parent.Done = TTriangleDone;

    /* Создание точек — вершин треугольника */
    this->A = newTPoint(ax, ay);
    this->B = newTPoint(bx, by);
    this->C = newTPoint(cx, cy);
}
```

```
void TExecutableInit(TExecutable *this, PProcedure P) {
    this->Parent.Draw = TExecutableDraw;
    this->Parent.Done = TExecutableDone;

    this->ProcPtr = P;
}
```

Правильное уничтожение объектов очереди обеспечивается виртуальным деструктором Done, который автоматически вызывается при вызове расширенной функции *deleteTObject()*:

```
void deleteTObject(TObject *this) {
```

```

this->Done(this);
free ( this );
}

/* Здесь не требуется уничтожать точку специально, поскольку ни одна из ее
компонент не размещается в куче, поэтому и не требует удаления. Она будет
уничтожена автоматически при выходе из блока.*/
void TPointDone(void *this) {}

/* В уничтожении нуждается только точка—центр окружности, порожденная при ее
создании.*/
void TCircleDone(void *_this) {
    TCircle *this = (TCircle*) _this;
    delete TObject((TObject*)this->center);
}

void TTriangleDone(void *_this) {
    TTriangle *this = (TTriangle*) _this;
    delete TObject((TObject*)this->A);
    delete TObject((TObject*)this->B);
    delete TObject((TObject*)this->C);
}

void TExecutableDone(void *this) {}

Пример добавления объекта в очередь:

int main() {

    TQueue *Q = newTQueue();

    Q->Push(queue, (TObject*) newTTriangle(1,2,3,4,5,6));
    Q->Push(queue, (TObject*) newTCircle(1,2,3));
    Q->Push(queue, (TObject*) newTExecutable(ExampleDraw));

    while (!Q->Empty(queue)) {
        Q->Top(queue)->Draw(Q->Top(queue));
        Q->Top(queue)->Done(Q->Top(queue));

        Q->Pop(queue);
    }
}

```

#### 7.4.4.2 C++

В C++ все вышеописанное выглядит короче и яснее. Во-первых, в силу краткости и выразительности языка, а во-вторых *все* конструкторы и деструкторы вызываются авто-

матически при объявлении переменных-экземпляров класса и выходе из блока, в котором они были объявлены.

Так, все объекты Delphi размещаются только в куче и при этом синтаксис инструкций различается. Например, для размещения в куче переменной типа **integer** необходимо написать

```
new(pi);
```

А чтобы разместить некоторый класс *X*, необходимо писать совершенно другие инструкции в духе

```
po := X.Init;
```

или

```
po := new(X, Init);
```

В C++ эти инструкции выглядят более однообразно и логично

```
int* pi = new int;  
X* px = new X;
```

Благодаря этому использование переменной типа класс ничем не отличается от использования переменной любого встроенного типа.

#### 7.4.4.3 Java

Синтаксис языка Java очень похож на синтаксис C/C++. Отличия заключаются в том, что все модули Java являются классами, а точкой входа для выполнения программы служит метод *main()*, определенный в одном из классов (впрочем, этот метод может быть определен в каждом классе). Еще одним отличием является автоматическое управление памятью и, как следствие, «сборка мусора», поэтому все объекты (кроме переменных скалярных типов, которые объектами не являются) размещаются исключительно в куче. Среда выполнения, называемая *виртуальной Java-машиной* (Java Virtual Machine — JVM), следит за использованием объектов кучи и автоматически уничтожает их в том случае, когда на них не остается ссылок. Отсутствие деструкторов в стиле C++ (вместо них используется метод *finalize()*) не позволяет в некоторых случаях возложить такие естественные операции как освобождение ресурсов на деструктор и его неявный вызов. Вместо этого приходится явно вызывать метод *finalize()*.

#### 7.4.4.4 C#

В недавно предложенном Microsoft языке C# унаследованы лучшие черты C++ и Java, учтены уроки их реализации и использования.

### 7.4.5 ООП в абсолютно объектных средах

И C++, и Java, и C#, и Eiffel, и Object Pascal страдают существенным недостатком — это фон Неймановские языки, в которые внедрена чуждая им объектная концепция. Однако уже довольно давно существуют абсолютно объектные языки программирования SmallTalk [73, 60] и CLOS [73].

# Глава 8

## Языки программирования не фон Неймановских моделей

### 8.1 Языки реляционного типа (SETL, SQL)

В обычных языках программирования типа Паскаля и Си для получения результата необходимо описать сам процесс получения результата. Тем не менее, существуют области информатики, достаточно подробно исследованные с точки зрения математики. Одной из таких областей являются базы данных. В 1970 году доктор Кодд создал очень популярную сейчас модель представления данных в виде прямоугольных таблиц [67]. Такие таблицы, согласно Кодду, являются множествами строк-кортежей, поэтому к ним применимы операции объединения, пересечения, вычитания и др. Кроме операций над строками Кодд предусмотрел операции над столбцами таблицы: проекции, условия выборки, слияния и др. Благодаря математической основе такого подхода пользователям баз данных больше не приходилось описывать, как произвести выборку или поменять значение в одной из строк таблицы, достаточно лишь было указать что необходимо сделать. Для описания действий с прямоугольными таблицами был разработан язык SQL, выгодно отличающийся от остальных языков тем, что любые запросы к СУБД являются читабельными и корректными предложениями на английском языке. В настоящее время SQL является стандартом де-факто для всех СУБД [45].

Отметим еще один интересный язык такого класса — SETL [69, 94].

### 8.2 Языки логического программирования (PROLOG)

[58].

Традиционные языки программирования основаны на фон Неймановской модели вычислений, в которой вычисления проводятся в соответствии с закодированным определенным образом алгоритмом. Понятие алгоритма, определяемое формальными математическими конструкциями типа машины Тьюринга, является ключевым для всех языков программирования фон Неймановского типа, или *императивных языков программирования*.

Альтернативами алгоритмическому подходу являются парадигмы программирования, основанные на других математических моделях. Например, *аппликативные языки программирования* (в частности, ЛИСП) основаны на  $\lambda$ -исчислении (т. н. *функциональное*

*программирование*), декларативные — на логике предикатов первого порядка (*логическое программирование*). Функциональное и логическое программирование играют важнейшую роль в таком разделе информатики, как *искусственный интеллект*.

Реализацией логического программирования на ЭВМ является язык программирования Пролог. «Программа»<sup>1</sup> на Прологе представляет собой набор *фактов* о некоторых именованных объектах (называемых *атомами*) и правил. Основным действием Пролог-системы является ответ на специальным образом сформулированный запрос пользователя относительно объектов программы.

Для ответа на запрос Пролог-система должна быть способна проводить над содержащимися в программе фактами *рассуждения* в соответствии с указанными правилами. Основной деятельностью вычислительной системы таким образом становится не выполнение строго специфицированной последовательности действий, а способность к *логическому выводу*. Для машинной реализации на традиционных фон Неймановских ЭВМ в Пролог-систему закладывается алгоритм логического вывода, именуемый *логическим интерпретатором*<sup>2</sup>.

Рассмотрим простой пример программы, предсказывающей оценку студента на экзамене по информатике. Программа будет содержать факты, подобные приведенным ниже:

```
умеет_программировать(вася).
умеет_программировать(петя).
читал(vasya, бауэр_гооз).
читал(петя, гарри_поттер).
книга_про(бауэр_гооз, информатика).
книга_про(гарри_поттер, фантастика).
```

Здесь перечислены студенты, умеющие программировать (по результатам семинарских занятий), а также информация о том, кто какие книги читал для подготовки к экзамену. Также факты содержат информацию о направленности той или иной книги.

Затем необходимо указать правила определения оценки. Будем считать, что 5 получает студент, имеющий практические и теоретические знания, 4 — студент с чисто практическими навыками, и 3 — любой явившийся на экзамен студент. (Наши гипотетические правила приема экзамена, вероятно, не согласуются с представлениями читателей, в этом случае им представляется самостоятельно изменить программу в соответствии со своими представлениями).

```
экзамен(X, 5) :- умеет_программировать(X), знает_материал(X).
экзамен(X, 4) :- умеет_программировать(X).
экзамен(X, 3).
```

```
знает_материал(X) :- читал(X, Y), книга_про(Y, информатика).
```

Для прогнозирования оценки достаточно ввести запрос вида

```
?— экзамен(vasya, X).  
X = 5
```

<sup>1</sup>Здесь термин «программа» применяется в смысле, отличном от привычного нам понятия программы на императивном языке программирования как синонима понятия «алгоритм».

<sup>2</sup>Чудес не бывает, поэтому весь логический вывод основан на переборе фактов, известных Пролог-системе.

и получить в ответ прогноз оценки.

С формальной точки зрения для описания ситуации Пролог использует *предикаты*, т. е. функции, принимающие истинные или ложные значения в зависимости от значений аргументов. Предикат представляет собой частный случай более общей структуры, именуемой *структурным термом*. Примером структурного терма может быть подробное (более подробное, чем в предыдущем примере) описание книги:

книга(бауэр\_гооз, информатика, издание(мир, 1987), 230).

где указаны авторы книги, название, издание (издательство и год издания) и количество страниц. Структура «издание» представляет собой вложенный структурный терм.

Внимательный читатель уже вероятно понял, что вложенные структурные термы являются естественным способом представления деревьев и, соответственно, многих других структур данных (списков, матриц и т. д.). Из-за естественности представления деревьев Пролог оказывается удобным для реализации алгоритмов, интенсивно использующих структуры данных в процессе работы. Так, например, алгоритм поиска в глубину запишется на Прологе следующим образом:

```
depth_search(Initial, Final, GetterName, Path) :- depth_search(Initial, Final,  
    GetterName, [Initial], Res), reverse(Res, Path).
```

```
depth_search(Final, Final, _, Path, Path).
```

```
depth_search(Initial, Final, GetterName, Path, Res) :- Getter =.. [GetterName,  
    Initial, Next], call(Getter), not(memberchk(Next, Path)), depth_search(Next,  
    Final, GetterName, [Next | Path], Res).
```

Поиск в глубину осуществляется четырехместным предикатом **depth\_search**, который является оболочкой для пятиместной версии предиката. Исходный предикат принимает в качестве аргументов начало и конец пути, имя предиката-«поставщика» вершин, соседних с данной, а также искомый путь. Таким образом, кроме поиска пути предикат может быть использован и для проверки, существует ли указанный путь между вершинами.

Поскольку в Прологе можно легко добавлять элементы в начало списка, то найденный путь будет записан в обратном порядке. Чтобы переписать его в естественном виде, используется *встроенный* предикат **reverse**. Этот предикат может быть легко реализован на самом Прологе; в этом смысле множество встроенных предикатов Пролога так же устроено и выполняет те же функции, что и стандартная библиотека языков С/С++).

Для поиска пути Пролог-система будет перебирать вершины, смежные с данной при помощи предиката **GetterName** (в Паскаль-программе это было реализовано перебором элементов строки матрицы смежности). Пролог-система при помощи оператора =.. (читается «юнив») сформирует предикат из имени и аргументов, после чего полученный предикат **Getter** будет вызван для поиска смежной вершины. Таким образом, **depth\_search** может быть использован для поиска в любых графах, в том числе и таких, число ребер в которых хотя и является конечным, но очень большим.

Еще одним стандартным предикатом является **memberchk**. Этот предикат находит *первое* вхождение элемента в список и тоже может быть реализован на Прологе. Кроме него существует предикат **member**, который при помощи переборов с возвратами найдет *все* вхождения данного элемента в список.

Существенно более сложный алгоритм поиска в глубину запишется на Прологе лишь немного сложнее:

```
advance([Node | Path], GetterName, Res) :- Getter =.. [GetterName, Node, NewNode],  
    findall([NewNode, Node | Path], (Getter, not(memberchk(NewNode, [Node | Path  
])), Res)).  
  
breadth_search(Initial, Final, GetterName, Path) :- breadth_search_h(Final,  
    GetterName, [[Initial]], Res), reverse(Res, Path).  
  
breadth_search_h(Final, _, [[Final | Path] | _], [Final | Path]).  
breadth_search_h(Final, GetterName, [Path | Paths], Res) :- advance(Path,  
    GetterName, NewPaths), append(Paths, NewPaths, Paths1), breadth_search_h(  
    Final, GetterName, Paths1, Res).
```

Этот пример сложнее поиска в глубину. Но сложность связана главным образом с реализацией очереди путей в виде *списка списков*. Предикат **breadth\_search\_h** использует предикат **advance** для построения всех продолжений какого-либо пути. Предикат **findall** находит все варианты продолжений, проверяя их на допустимость (вершины не должны повторяться во избежание зацикливания!) и помещая все допустимые пути в список (списков!) *Res*. После получения всех продолжений они дописываются в конец очереди *Paths* и поиск продолжается.

Сравните этот пример с приведенным ранее примером на языке Паскаль! Пример на Прологе оказывается более естественным за счет наличия в нем встроенных в язык естественных структур данных, а также за счет того, что собственно алгоритм поиска неявно содержится в логическом интерпретаторе, а Пролог-программа является собой лишь список свойств процесса поиска.

Впрочем, и Пролог не свободен от недостатков. Одним из них является предикат **not**. Операция отрицания многократно усложняет логику первого порядка и в принципе не все выражения этой логики могут быть вычислены именно из-за **not**. Поэтому эта операция в рамках Пролога означает «не удалось доказать». С этим связана еще одна особенность всех Пролог-систем. Возвращаясь к примеру с экзаменом рассмотрим такой запрос:

```
?- экзамен(коля, X).  
X = 3
```

Интересный результат! Дело в том, что Пролог-система ничего не знает о таком студенте, поэтому не может доказать, что он получит 5 или 4. Единственное, что остается — это оценка 3. Эту особенность называют *предположением о замкнутости мира*, т. е. любая Пролог-система может считать истинными те и только те факты, которые есть в ее базе данных.

## 8.3 Языки функционального программирования (LISP, AFP)

В настоящее время мы можем констатировать лишь такие особенности Лиспа как список общего вида в качестве основной структуры данных, рекурсивно-интерпретативную природу выполнения и возможность смешанных вычислений. Знакомство с элементами Лиспа

полезно для понимания многих примеров, программ и идей, излагаемых в обычных книгах по информатике. Это элемент американской информатической культуры [50].

Система функционального программирования AFP была предложена Дж. Бэкусом в его знаменитой статье о порочности фон Неймановского стиля программирования [26].

Учебное издание

**Гайсарян Сергей Суренович  
Зайцев Валентин Евгеньевич  
КУРС ИНФОРМАТИКИ**

Редактор *P.M.Белозерова*  
Художественный редактор *Д. А. Абрамов*  
Технический редактор *Д. В. Сошиников*  
Корректор *В.Е.Зайцев*  
Обложка художника *А. В. Крапивенко*

---

Сдано в набор 31.10.93

Формат  $60 \times 90 \frac{1}{16}$

Бум. офсетная

Подписано в печать 28.12.93

Гарнитура Computer modern.

Уч.-изд. л. 25

Тираж 1000 экз.

---

Издательство Вузовская книга  
125993, Москва, Волоколамское шоссе, 4.  
Типография МГУ  
119992, Москва, ул. Академика Хозлова, 11.