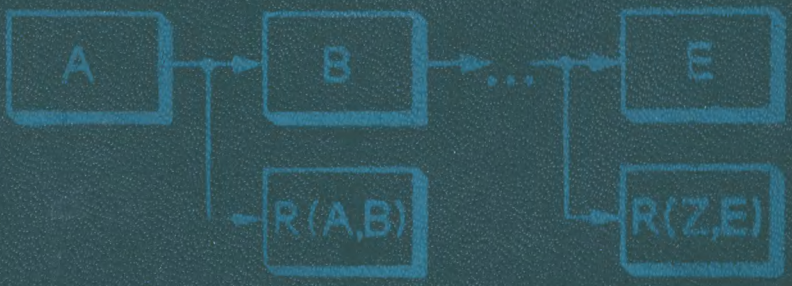
А.Г. Кушниренко
Г.В. Лебедев

Программирование для математиков



А. Г. КУШНИРЕНКО
Г. В. ЛЕБЕДЕВ

ПРОГРАММИРОВАНИЕ ДЛЯ МАТЕМАТИКОВ

Допущено Министерством
высшего и среднего специального образования СССР
в качестве учебного пособия
для студентов высших учебных заведений
по специальностям
«Математика» и «Прикладная математика»



МОСКВА «НАУКА»
ГЛАВНАЯ РЕДАКЦИЯ
ФИЗИКО-МАТЕМАТИЧЕСКОЙ ЛИТЕРАТУРЫ
1988

ББК 22.18
К96
УДК 519.6(075.8)

Кушниренко А. Г., Лебедев Г. В. **Программирование для математиков:** Учеб. пособие для вузов — М.: Наука. Гл. ред. физ.-мат. лит., 1988.— 384 с.

ISBN 5-02-014235-2

Книга содержит расширенный вариант начального курса программирования, который читается на механико-математическом факультете МГУ с 1980 г. Цель курса — заложить фундамент общей программистской культуры слушателей и научить их грамотно программировать практически задачи объемом несколько тысяч строк. Основу курса составляют понятие исполнителя, технология программирования «сверху вниз» и развитые структуры данных. В числе изучаемых в курсе законченных программных систем — управление станком с ЧПУ и «луноходом», реализация простейшего компилятора арифметических формул, построение изображения полиэдра с удалением невидимых линий, ссылочная реализация списка, хеширование, двумерное хеширование по равномерной сетке, реализация виртуальной памяти, простейшей файловой системы и компонент экранного редактора текстов.

Изложение ведется в едином стиле с использованием понятия исполнителя на учебном языке программирования с русской лексикой.

Для студентов математических факультетов университетов и факультетов прикладной математики вузов.

Табл. 11. Ил. 96.

Рецензенты:
кафедра прикладной математики МИЭМ,
доктор физико-математических наук *А. Л. Семенов*

К $\frac{1702070000-158}{053\,(02)-88}$ КБ-14-63—88

ISBN 5-02-014235-2

© Издательство «Наука», Главная редакция физико-математической литературы, 1988

ОГЛАВЛЕНИЕ

Предисловие 6

Глава 1. **Начала программирования** . . . . . .

1. Основные понятия программирования, примеры исполнителей и простейших программ 9

Основные понятия программирования (9). Исполнитель «Резчик металла» (10). Исполнитель «Путник» (13). Стек (15). Исполнитель «Стековый калькулятор» (16). Исполнитель «Счетчик» (18). Задачи и упражнения (19).

2. Процесс выполнения программы. Управляющие конструкции и утверждения 19

Управляющие конструкции (20). Конструкции выбора (23). Циклы (23). Утверждение, отказ и ничего не делать (24). Примеры программ (25). Задачи и упражнения (27).

3. Основная задача программирования и технология «сверху вниз» 27

Программы и их вызовы (27). Основная задача программирования (29). Технология программирования «сверху вниз» (30). Шаг декомпозиции (31). Пример разработки программы по технологии «сверху вниз» (33). Задачи и упражнения (41).

4. Процесс разработки программ. Рекурсия, итерация, проектирование цикла с помощью инварианта 43

Разбиение задачи на подзадачи (44). Вызовы программ и «дано»/«получить» (46). Пример (49). Рекурсия (50). Конструкция циклов (53). Итерация (53). Схема проектирования цикла с помощью инварианта (56). Задачи и упражнения (62).

5. Процесс разработки программ. Один пример 64

Формулировка задачи (64). Решение (66). Задачи и упражнения (76).

6. Объекты, параметры, типы. Схема вычисления инвариантной функции 77

Объекты (77). Локальные объекты программ (79). Глобальные объекты исполнителей (81). Входные и выходные параметры программ (82). Программы, вырабатывающие значение (84). Типы объектов (85). Описания типов (87). Примеры программ (88). Схема вычисления значения инвариантной функции (90). Задачи и упражнения (92).

7. Способы конструирования типов, объектов и исполнителей 93

Перечисление (93). Отрезок (94). Запись (95). Структуры (96). Стек (97). Очередь (98). Дек (98). Множество (99). Нагруженное множество (100). Последовательность (100). Л2-список (101). Л1-список (102). Вектор (103). Матрица (104). Динамический вектор (104). Структуры как способы конструирования исполнителей (104). Структуры как способы конструирования объектов (105). Примеры программ (106). Циклы «для каждого» (107). Задачи и упражнения (111).

8. Индуктивное вычисление функций на пространстве последовательностей 113

Индуктивные функции (114). Стационарные значения (116). Индуктивные расширения функций (117). Минимальное индуктивное расширение (118). Теорема существования (119). Разные пространства и доопределения (120). Практика (122). Замечания (130). Задачи и упражнения (134).

Глава 2. **Несколько примеров программ** 136

9. Проект «Выпуклая оболочка» последовательно поступающих точек плоскости 136

Формулировка задачи (136). Основные идеи (137). Первый шаг декомпозиции (138). Второй шаг декомпозиции (142). Третий шаг декомпозиции (148). Задачи и упражнения (149).

10. Компиляция и интерпретация. Реализация простейшего компилятора с языка арифметических формул 150

Язык и грамматика (150). Компиляция (154). Рекурсивная реализация компилятора правильных формул (154). Компиляция и интерпретация (161). Реализация компилятора с помощью стека (161). Задачи и упражнения (166).

11. Проект «Построение изображения полиэдра» 167

Постановка задачи (168). Реализация (170). Оптимизация (183). Фильтрование граней (183). Предкомпиляция граней (187). Задачи и упражнения (187).

Глава 3. **Структуры данных и их реализации** 189

12. Примеры реализации одних структур данных на базе других. Непрерывные реализации на базе вектора 190

Последовательность на базе двух очередей (190). Непрерывные реализации на базе вектора (194). Ограниченный стек (194). Ограниченная очередь на базе «циклического» вектора (196). Задачи и упражнения (200).

13. Ссылочные реализации на базе вектора 202

Задачи и упражнения (210).

14. Три способа реализации множества на базе вектора. Последовательный поиск, бинарный поиск, хеширование 211

Непрерывная реализация; последовательный поиск (211). Непрерывная реализация; бинарный поиск (214). Битовая реализация (220). Хеширование (223). Задачи и упражнения (234).

15. Двумерное хеширование по равномерной сетке. Оптимизация алгоритма построения изображения полиэдра 235

Идея двумерного хеширования (235). Реализация (238). Оценки эффективности (245). Задачи и упражнения (248).

16. Виртуальная память 249

Кэш-память (252). Виртуальная память (254). Хеш-реализация виртуальной памяти (261). Задачи и упражнения (264).

17. Простейшая файловая система 265

Постановка задачи (266). Основные идеи реализации (969). Реализация (270). Задачи и упражнения (281).

18. Иерархия структур данных при разработке программ . . . 283

Идеи реализации (287). Задачи и упражнения (298).

Глава 4. **Логическое устройство и принципы работы ЭПВМ** 300

19. Логическое устройство и принципы работы ЭВМ 300

Память (301). Процессор (304). Работа процессора (305). Команды процессора (306). Задание аргументов команд (311). Примеры программ (313). Внешние устройства, магистраль, адрес (315). Подпрограммы и их вызовы (318). Прерывания (319). Задачи и упражнения (322).

20. Работа программиста на ЭВМ 323

Начальная загрузка (324). Загрузка операционной системы (324). Программирование в кодах (325). Компиляция (325). Интерпретация (327). Соотношение компиляции и интерпретации (327). Синтаксически ориентированные редакторы (328). Инструментальные среды (329). ЭПВМ (330). Работа на ЭВМ непрограммиста (330).

Глава 5. Программирование на языке Фортран 332

21. Справочные сведения о языке Фортран. Простейшие примеры программ 333

Пример программы (333). Размещение информации внутри строки (333). Секции (335). Управляющие конструкции (337). Предопределенные типы (339). Способы конструирования (342). Задание объектов и типов объектов (342). Параметры (343). Функции (344). Константы (345). Ввод/вывод (345). Примеры перекодировки программ на Фортран (346). Некоторые вопросы устройства Фортран—машины (352). Расположение и отождествление объектов в памяти (352). Механизм передачи параметров (356). Часто встречающиеся ошибки (356). Задачи и упражнения (358).

22. Реализация исполнителей на Фортране. Примеры реализации структур данных 359

Приложение. Краткое неформальное описание использованного в книге языка программирования 374

Предметный указатель 380

ПРЕДИСЛОВИЕ

Эта книга содержит расширенный вариант начального курса программирования, который читается на механико-математическом факультете МГУ с 1980 г. Основная цель курса — заложить фундамент общей программистской культуры слушателей, в частности научить их грамотно программировать задачи объемом до нескольких тысяч строк. Значительную часть курса составляет изучение мини-проектов, т. е. законченных программных систем объемом несколько сотен строк. В их числе управление станком с ЧПУ и «луноходом», реализация простейшего компилятора арифметических формул, построение изображения полиэдра с удалением невидимых линий, ссылочная реализация списка, хеширование, двумерное хеширование по равномерной сетке, реализации виртуальной памяти, простейшей файловой системы и компонент экранного редактора текстов.

Курс посвящен программированию как таковому: вычислительные методы, приближенные вычисления, операционные системы и т. п. в нем не затронуты вовсе; первые две трети курса вообще не связаны с ЭВМ; синтаксису языков программирования не уделяется почти никакого внимания. Основной методический прием курса — обучение на примерах. Любые технические детали изучаются лишь в процессе решения какой-нибудь содержательной задачи.

В основе курса лежат три «кита»: понятие исполнителя, технология программирования «сверху вниз», развитые структуры данных. Эти три кита в свою очередь опираются на «черепаху» — курс предполагает активное изучение 6—8 тыс. строк эталонных текстов программ и создание новых программ общим объемом 1—2 тыс. строк. Курс можно изучать и без ЭВМ вообще, однако использование ЭВМ со специализированным программным обеспечением существенно повышает эффективность и качество обучения.

Центральное понятие курса — понятие исполнителя. Исполнителя можно представлять себе как робота с некоторой системой команд. Действия, выполняемые роботом по той или иной команде, могут зависеть от предыстории (состояния «памяти» робота) и от обстановки, в которой поступила команда. С программистской точки зрения исполнитель — это пакет программ, работающих над общими данными, т. е. информационно-прочный модуль в классификации Майерса. Его аналогами являются экземпляр класса в языке

Симула-67, объект в языке Smalltalk-80, пакет программ в языке Ада, модуль в языке Модула. Понятие исполнителя, однако, является в курсе первичным и не сводится просто к объединению программ. Одна из важнейших задач курса — научить слушателей мыслить именно в терминах исполнителей, а не отдельных программ: сначала придумывать исполнителя в целом, а уж потом решать, какие действия он будет уметь выполнять, какими программами эти действия будут реализованы и какая информация будет храниться в его памяти. Технология «сверху вниз» и структуры данных также излагаются в терминах исполнителей.

Книга состоит из пяти глав. Глава 1 — «Начала программирования» — содержит изложение основных понятий программирования, технологии «сверху вниз», управляющих конструкций и структур данных. Этот материал доступен даже школьнику и не требует никакого предварительного знакомства с программированием. Кроме того, в гл. 1 входит материал по базисным схемам обработки информации: рекурсия, итерация, проектирование цикла с помощью инварианта, вычисление инвариантной функции, индуктивное вычисление функций на пространстве последовательностей. Этот материал мог бы составить предмет отдельного небольшого курса, читаемого параллельно с курсом программирования, а его изложение предполагает наличие у читателя минимальной математической культуры.

Глава 2 посвящена решению трех задач: построению выпуклой оболочки последовательно поступающих точек плоскости; реализации простейшего компилятора с языка арифметических формул; построению изображения полиэдра с удалением невидимых линий. Выбор задач и стиль их изложения предполагают, по крайней мере, знание того, что такое выпуклая оболочка, ориентированная площадь, нормаль, полупространство и т. п. Глава не является обязательной для понимания следующих глав и, за исключением специальных вопросов, связанных с компиляцией (язык и грамматика), не содержит нового теоретического материала.

Глава 3 посвящена структурам данных (стек, дек, очередь, список, последовательность, множество и др.) и методам их реализации на базе линейной памяти. Изучаются непрерывные и ссылочные реализации, последовательный и бинарный поиск, хеширование, двумерное хеширование по равномерной сетке, реализации виртуальной памяти и простейшей файловой системы. Весь этот материал изложен в едином стиле с использованием понятия исполнителя.

В главах 1—3 программы записываются на учебном языке программирования с русской лексикой. Неформальное описание этого языка приведено в приложении 1.

Глава 4 является традиционной и посвящена устройству ЭВМ (на примере ЭВМ архитектуры PDP-11). Она содержит примеры программ в «кодах» и описание основных компонент базового программного обеспечения. Основное назначение этой главы — дать представление об устройстве современных ЭВМ и завершить формирование программистской «картины мира».

Глава 5 демонстрирует, как при работе на Фортране можно применить стиль и методы программирования первых трех глав. Излагается язык программирования Фортран, правила перекодировки программ на язык Фортран, методика реализации исполнителей, а также некоторые вопросы устройства Фортран-машины. Выбор языка объясняется тем, что Фортран является наиболее распространенным языком для научно-технических расчетов и различных систем автоматизации.

Авторы благодарны заведующему лабораторией вычислительных методов А. В. Михалеву, ст. н. с. В. Б. Бетелину, а также Д. В. Варсанофьеву, А. Г. Дымченко, Г. В. Маслову, Н. Н. Молчанову и многим другим сотрудникам, аспирантам и студентам механико-математического факультета МГУ, участвовавшим в начальной постановке, организации и дальнейшем проведении описанного в книге курса программирования, в создании специализированного программного обеспечения этого курса.

ГЛАВА I

НАЧАЛА ПРОГРАММИРОВАНИЯ

1. Основные понятия программирования, примеры исполнителей и простейших программ

Основные понятия программирования. *Исполнитель* — это человек, организация, механическое или электронное устройство, робот и т. п., умеющий выполнять некоторый вполне определенный набор *действий*. Каждое действие, которое способен выполнить исполнитель, называется *предписанием*, а вся совокупность таких действий — *системой предписаний* исполнителя. Приказ на выполнение действия, выраженный формальным, заранее фиксированным способом, называется *вызовом предписания*.

Удобно представлять себе исполнителя в виде устройства с кнопками на передней панели — возле каждой кнопки написано имя предписания, которое будет выполнено при нажатии на эту кнопку. Система предписаний такого исполнителя совпадает с совокупностью кнопок на его передней панели. Чтобы выписать ее на бумаге, достаточно переписать имена, написанные возле кнопок. Вызов предписания нужно представлять себе как нажатие на кнопку, после которого исполнитель выполняет соответствующее действие.

Всякое действие производится над некоторыми *объектами* и состоит в изменении *состояний* этих объектов. Объекты могут быть *внутренними* или *внешними*.

Внутренние объекты исполнителя называются *глобальными*. Глобальные объекты исполнителя образуют его память, хранящую некоторую информацию об истории работы исполнителя, — их состояния могут меняться в результате выполнения одних предписаний и сказываться на ходе выполнения следующих. Термин «глобальные» подчеркивает, что эти объекты относятся сразу ко всем предписаниям данного исполнителя, т. е. ко всему исполнителю целиком.

Внешние объекты называются *параметрами* предписания. Параметры связываются с исполнителем при вызове конкретного предписания на время выполнения этого предписания и делятся на *входные*, *выходные* и *входно-выходные*. Состояние входного параметра сказывается на выполнении предписания, но в результате выполнения не меняется. Состояние выходного параметра не сказывается на выполнении предписания, но может измениться в результате выполнения. Входно-выходные параметры являются и

входными, и выходными: их состояния и сказываются на результате, и могут меняться в результате выполнения предписания.

Если представлять себе исполнителя в виде устройства с кнопками, то глобальные объекты — это внутренние части устройства, состояние которых меняется при нажатии на кнопки. Возле некоторых кнопок на передней панели исполнителя есть отверстия, куда перед нажатием кнопки (перед вызовом предписания) следует поместить внешние объекты. Количество отверстий и их тип определяются конструкцией исполнителя. Сами внешние объекты можно представлять в виде магнитофонных кассет разных размеров, содержимое (состояние) которых может измениться, если объект (а точнее, соответствующее отверстие) является выходным. Эффект нажатия на кнопку — эффект вызова предписания — зависит не только от того, какая кнопка нажата, но и от состояний входных объектов (содержимого кассет, вставленных в отверстия предписания), а также от того, что происходило с исполнителем раньше, т. е. от состояния глобальных объектов, памяти исполнителя.

Таким образом, результат выполнения предписания полностью определяется состоянием:

— глобальных объектов исполнителя,

— входных и входно-выходных параметров предписания и состоит в изменении состояний;

— глобальных объектов исполнителя,

— выходных и входно-выходных параметров предписания.

Одно и то же предписание может быть выполнимым при одних состояниях глобальных объектов, входных и входно-выходных параметров и невыполнимым при других. В последнем случае возникает предусмотренная конструкцией исполнителя исключительная ситуация — *отказ* — и продолжение работы с исполнителем становится невозможным (в рамках приведенной выше модели отказ можно понимать как перегорание предохранителей).

Командуя исполнителями, мы обычно достигаем желаемого результата не одним вызовом какого-нибудь предписания, а определенной *последовательностью вызовов* разных предписаний. Такая последовательность (а точнее, некоторое ее формальное описание) называется *программой*. Соответственно деятельность по составлению таких формальных описаний называется *программированием*. Как правило, программа гораздо короче, чем та последовательность действий, которую она описывает. Неудачно составленная программа может описывать бесконечную последовательность действий или последовательность, приводящую к отказу одного из исполнителей.

Рассмотрим несколько примеров исполнителей и простейших программ для них.

Исполнитель «Резчик металла» (РМ).

Пример 1. Этот исполнитель представляет собой простейший станок с числовым программным управлением и состоит

из прямоугольного стола, на который можно класть листы металла, и резака, расположенного над столом. Резак можно поднимать, опускать и перемещать параллельно краям стола. Поскольку мы будем рисовать стол и металл на бумаге (на доске), то будем говорить о перемещениях резака вправо, влево, вверх и вниз. При опускании резака и при движении в опущенном состоянии в листе металла под резаком образуется прорезь.

Будем считать, что резак имеет размер 0.5*0.5 мм и перемещается с шагом 0.5 мм. Идеализируя ситуацию, можно считать стол Резчика металла клетчатым полем (с клетками размером 0.5*0.5 мм). Резак всегда находится в некоторой клетке этого поля; его можно перемещать в соседние клетки по горизонтали и по вертикали, но не по диагонали. При опускании или передвижении опущенного резака вырезается целиком вся клетка.

Резчик металла снабжен датчиками, которые позволяют анализировать, находится ли резак в клетке у верхнего, нижнего, левого или правого края стола и есть ли металл в клетке под резаком.

Система предписаний Резчика металла состоит из следующих предписаний:

1. начать работу
2. поднять резак
3. опустить резак
4. шаг вправо
5. шаг влево
6. шаг вверх
7. шаг вниз
8. резак над металлом : да/нет
9. резак у верхнего края стола : да/нет
10. резак у нижнего края стола : да/нет
11. резак у правого края стола : да/нет
12. резак у левого края стола : да/нет
13. кончить работу

Запись «резак над металлом : да/нет» означает, что у предписания «резак над металлом» есть один выходной объект и этот объект может принимать одно из двух состояний: либо да, либо нет. (Можно считать, что в кнопку «резак над металлом» встроена лампочка, которая после нажатия на кнопку может либо загореться, либо не загореться.) Предписания такого вида мы будем называть *вырабатывающими значение типа* да/нет.

Более компактно систему предписаний Резчика металла можно записать в виде

1. начать работу

2. поднять/опустить резак
3. шаг вправо/влево/вверх/вниз
4. резак над металлом : да/нет
5. резак у верхнего/нижнего/правого/левого края стола : да/нет
6. кончить работу

В дальнейшем мы будем пользоваться именно такой сжатой формой записи системы предписаний. Цифры слева в системе предписаний нумеруют строки (группы предписаний).

Поясним теперь, что происходит при выполнении отдельных предписаний Резчика металла (как принято говорить в программировании, поясним их *семантику*). По предписанию «начать работу» резак оказывается в левом верхнем углу стола в поднятом положении. При попытке сделать шаг вверх, когда резак уже находится у верхнего края стола, возникает исключительная ситуация «отказ». Аналогично отказ происходит при попытке сделать шаг вправо, когда резак у правого края, и т. п. Предписания «поднять/опустить резак» переводят резак в требуемое положение независимо от того, был ли он поднят или опущен, и никогда не приводят к отказу. Резак можно двигать и в поднятом, и в опущенном состоянии независимо от того, есть под ним металл или нет.

Пример программы для исполнителя «Резчик металла»:

```
программа прямоугольник
| дано      : | лист металла закрывает весь стол Резчика
| получить: | от левого верхнего угла отрезан прямоугольник
|           | размером 2*3 шага Резчика металла
| --------
| РМ.начать работу
| РМ.шаг вниз
| РМ.шаг вниз
| РМ.опустить резак
| РМ.шаг вправо
| РМ.шаг вправо
| РМ.шаг вправо
| РМ.шаг вверх
| РМ.шаг вверх
| РМ.поднять резак
| РМ.кончить работу
конец программы
```

Эта программа описывает последовательность из 11 вызовов предписаний исполнителя «Резчик металла». По окончании ее выполнения в листе металла будут прорезаны клетки, заштрихованные на рис. 1.1.

Конец примера 1.

Напомним: программа — это формальное описание последовательности вызовов предписаний (в примере 1 — предписаний Резчика металла). Слово «формальное» означает, что описание должно проводиться в соответствии со строгими, вполне определенными правилами. Мы не будем сейчас перечислять их, однако заметим, что в эти правила входят наличие и порядок слов программа, дано,

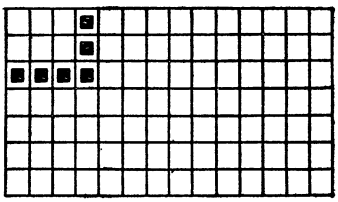


Рис. 1.1

получить, конец программы, составленная из точек вертикальная линия, соединяющая слово **программа** со словами **конец программы,** наличие отступа (т. е. сдвига имен предписаний вправо от вертикальной черты) и т. п.

Если в строке программы встречается знак |, то текст, начиная от этого знака и до конца строки, считается комментарием. *Комментарий* — это произвольный текст, который никак не влияет на последовательность вызовов предписаний при выполнении программы, а служит лишь для пояснения смысла программы при чтении ее человеком. Например, в программе «прямоугольник» комментарием является текст после слов **дано** и **получить.**

Исполнитель «Путник» (П).

Пример 2. Этого исполнителя следует представлять в виде лунохода, который находится где-то в далекой неизвестной местности (назовем ее *квадрантом*). Путником можно управлять (например, по радио) с помощью следующей системы предписаний:

1. начать работу
2. впереди/справа/слева свободно : **да/нет**
3. впереди/справа/слева занято : **да/нет**
4. сделать шаг
5. шагать до упора
6. повернуться направо/налево
7. повернуться на север/юг/запад/восток
8. кончить работу

Квадрант Путника является прямоугольным клетчатым полем неизвестных размеров. Некоторые клетки квадранта могут быть заняты препятствиями. По краю квадрант обнесен сплошной стеной,

По предписанию «начать работу» Путник оказывается в северо-западной (левой верхней) клетке квадранта, лицом на восток (рис. 1.2).

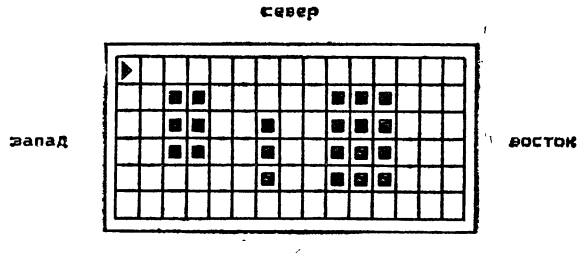


Рис. 1.2

На этом и следующих рисунках Путник изображается стрелкой, указывающей ориентацию. Например, стрелка ▶ указывает ориентацию на восток, а стрелка ▼ — на юг.

По предписанию «слева свободно» Путник анализирует, что находится слева от него. Если слева стена или клетка препятствия, то он отвечает нет. Если же слева пустая клетка квадранта, то он отвечает да. Предписание «слева занято» выполняется аналогично. В состоянии, изображенном на рис. 1.2, в ответ на «слева занято» Путник ответит да, а на «слева свободно» — нет.

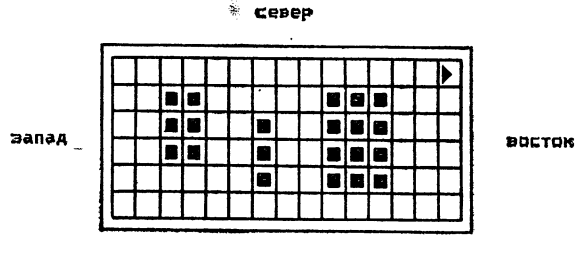


Рис. 1.3

Если клетка впереди Путника свободна, то он может переместиться в нее по предписанию «сделать шаг». Если же впереди стена или препятствие, то попытка выполнить предписание «сделать шаг» приведет к отказу. По предписанию «шагать до упора» Путник перемещается вперед до тех пор, пока не упрется в стену или в препятствие. Из состояния, изображенного на рис. 1.2, по предписанию «шагать до упора» Путник переместится в северо-восточную (правую верхнюю) клетку квадранта и остановится в ней лицом по-прежнему на восток (рис. 1.3).

Если в этот момент Путник получит предписание «шагать до упора» еще раз, то он никуда не сместится, однако и отказа не возникнет. Таким образом, вызов предписания «шагать до упора» никогда не приводит к отказу.

Наконец, по предписанию «повернуться ...» Путник поворачивается. При этом ему говорится, либо куда он должен повернуться (налево — на 90° против часовой стрелки или направо — на 90° по часовой стрелке), либо в каком положении он должен оказаться (лицом на север, на запад и т. п.).

Два примера программ для Путника:

```
программа прогулка
. дано      : | препятствия не прилегают к стенам квадранта
. получить: | Путник обошел квадрант по периметру
. .....
. Путник.начать работу
. Путник.шагать до упора
. Путник.повернуться направо
. Путник.шагать до упора
. Путник.повернуться направо
. Путник.шагать до упора
. Путник.повернуться направо
. Путник.шагать до упора
. Путник.повернуться направо
. Путник.кончить работу
конец программы
```

```
программа назад до упора
. дано      : | Путник где-то в квадранте
. получить: | Путник сместился назад до упора,
.           | ориентация Путника не изменилась
. --------
. Путник.повернуться налево
. Путник.повернуться налево
. Путник.шагать до упора
. Путник.повернуться налево
. Путник.повернуться налево
конец программы
```

Конец примера 2.

Стек. Для следующего примера понадобится понятие стека. *Стеком* называется любая структура, в которой могут накапливаться какие-то элементы и для которой выполнено следующее основное условие: *элементы из стека можно доставать только в порядке, обратном порядку добавления их в стек.* Это условие часто называют

также принципом «последним пришел — первым ушел» или LIFO (Last In — First Out). Примерами стеков могут служить:

а) обыкновенная детская пирамидка (рис. 1.4, *а*): большое колечко, которое было надето раньше, нельзя снять, не сняв предварительно маленькое, которое было надето позже:

б) железнодорожный тупик (рис. 1.4, *б*): вагон Б нельзя выгнать из тупика, не выгнав предварительно вагон В, который был загнан позже;

в) труба с одним запаянным концом, куда помещают разноцветные «бочонки» (рис. 1.4, *в*). Последний пример в наибольшей

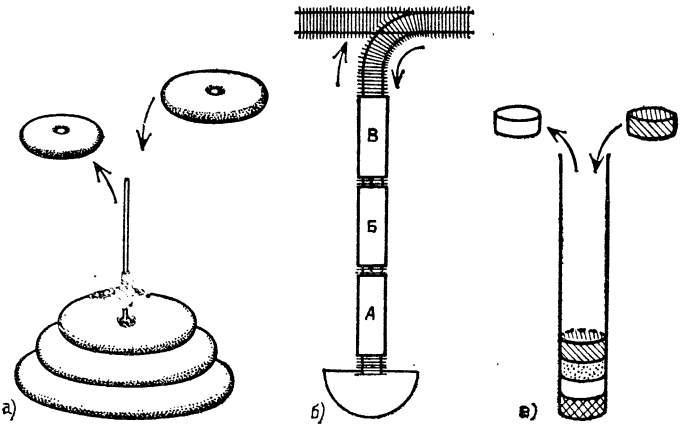


Рис. 1.4. а) детская пирамидка, б) железнодорожный тупик, в) труба с одним запаянным концом

степени соответствует программистскому понятию стека: заглядывая в трубу, мы можем видеть, что бочонков в трубе нет, или видеть цвет верхнего бочонка, но не можем видеть, есть ли бочонки под верхним, сколько их и каких они цветов.

Элемент стека, который в данный момент можно взять, т. е. самый «верхний» (верхнее колечко на пирамидке, последний загнанный вагон в тупике и т. п.), называется *вершиной стека*. Если число элементов в стеке не может превышать некоторой величины, то стек называют *ограниченным*, а максимальное число элементов в нем — *глубиной стека*. Стек, в котором нет ни одного элемента, называется *пустым*.

Исполнитель «Стековый калькулятор» (СК).

Пример 3. Система предписаний:

1. начать работу
2. добавить ⟨вх: число⟩ в стек

3. сложить/вычесть/умножить/разделить
4. показать результат
5. кончить работу

Запись «добавить ⟨вх: число⟩ в стек» означает, что это предписание имеет один входной объект (входной параметр), который является числом.

Исполнитель «Стековый калькулятор» имеет два глобальных объекта: стек чисел (т. е. стек, элементами которого являются числа) и табло, на котором по предписанию «показать результат» изображается вершина стека (сам стек при этом не меняется).

После вызова предписания «начать работу» стек калькулятора пуст, а на табло ничего не изображено. Предписание «добавить ⟨вх: число⟩ в стек» добавляет указанное при вызове число в стек. Предписание «показать результат» изображает на табло вершину стека. После этого табло не меняется до следующего вызова предписания «показать результат». При выполнении арифметических операций (сложить, вычесть, умножить, разделить) Стековый калькулятор достает из стека последовательно сначала правый, а затем левый аргументы операции, выполняет операцию и полученный результат помещает в стек (рис. 1.5).

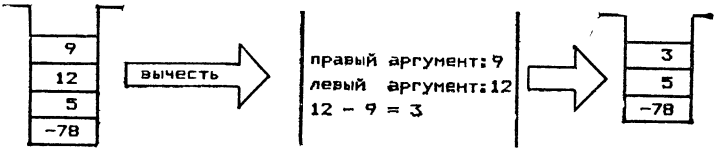


Рис. 1.5

Порядок выборки аргументов из стека легко запомнить: если мы хотим вычислить с помощью калькулятора разность 12 — 9, то аргументы операции «—» надо *поместить* в стек в том порядке, в котором мы их читаем — сначала 12, а потом 9.

При выполнении любой арифметической операции число элементов в стеке уменьшается на 1. Попытки выполнить операцию, когда в стеке меньше двух элементов, или показать результат, когда стек пуст, приводят к отказу.

Пример программы для Стекового калькулятора:

программа вычисление
• **дано** :
• **получить:** | значение формулы $5*(7+8)+25$ на табло СК
• ---------
• СК.начать работу
• СК.добавить ⟨5⟩ в стек

. СК.добавить ⟨7⟩ в стек
. СК.добавить ⟨8⟩ в стек
. СК.сложить
. СК.умножить
. СК.добавить ⟨25⟩ в стек
. СК.сложить
. СК.показать результат
. СК.кончить работу
конец программы

Последовательные состояния стека при выполнении этой программы изображены на рис. 1.6.

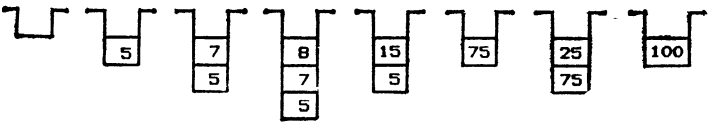


Рис. 1.6

Конец примера 3.

Исполнитель «Счетчик».

Пример 4. Система предписаний:

1. начать работу
2. установить в нуль
3. увеличить на единицу
4. показать значение
5. кончить работу

Исполнитель имеет два глобальных объекта: внутренний, состоянием которого может быть неотрицательное целое число, и табло, на котором по предписанию «показать значение» это число изображается. Семантика предписаний очевидна из их названий.

Пример программы для исполнителя «Счетчик»:

программа три
. **дано** :
. **получить:** | на табло Счетчика число 3
. --------
. Счетчик.начать работу
. Счетчик.установить в нуль
. Счетчик.увеличить на единицу
. Счетчик.увеличить на единицу
. Счетчик.увеличить на единицу
. Счетчик.показать значение
. Счетчик.кончить работу
конец программы

Конец примера 4.

ЗАДАЧИ И УПРАЖНЕНИЯ

1. Напишите программу для исполнителя «Резчик металла», вырезающую в листе металла номер Вашего дома.

2. Напишите программу для исполнителя «Стековый калькулятор», которая вычисляет значение формулы $5-(6+8/(7-5))+1$.

3. Напишите программу для Стекового калькулятора, которая показывает на табло год Вашего рождения.

4. Решите задачу 3 при условии, что в программе в предписаниях «добавить ⟨вх: число⟩ в стек» можно использовать только однозначные числа от 0 до 9.

5. Придумайте какого-нибудь исполнителя. Опишите его систему предписаний, глобальные объекты. Напишите две существенно разные программы для этого исполнителя.

2. Процесс выполнения программы. Управляющие конструкции и утверждения

Мы уже говорили, что программа — это формальное описание последовательности вызовов предписаний, и объясняли, что происходит при выполнении программы. До сих пор, однако, мы не уточняли, кто и как выполняет программу, т. е. переводит это формальное описание в конкретную последовательность вызовов. В программировании обычно считается, что эту работу выполняет некоторое *автоматическое* устройство — машина. Мы будем называть это

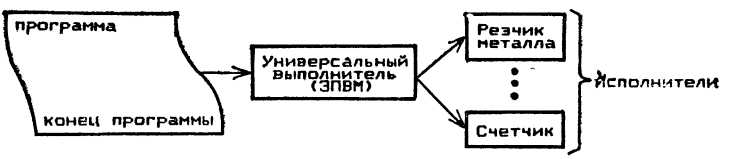


Рис. 2.1. Процесс выполнения программы

устройство Универсальным Выполнителем, или ЭПВМ — Электронно-Программной Выполняющей Машиной (рис. 2.1).

Для того чтобы программу могло выполнить *автоматическое* устройство, программа должна подчиняться строгим правилам, записываться на некотором понятном ЭПВМ формальном языке. Этот язык принято называть *языком программирования*. Разные ЭПВМ могут выполнять программы, вообще говоря, на различных языках. Таким образом, использованный нами язык программирования — лишь один из многих. В последних главах книги мы познакомимся с двумя другими языками (двумя другими Универсальными Выполнителями). Всего же в мире существуют тысячи языков программирования и продолжают появляться все новые.

Программы можно выполнять и без автоматического Универсального Выполнителя. Например, человек может читать текст программы и нажимать на соответствующие кнопки на соответствующих исполнителях. В дальнейшем нас будет интересовать, не кто выполняет наши программы, а существо дела — как записать программу и что произойдет при ее выполнении. Однако удобно объяснять смысл тех или иных форм записи программ, описывая поведение Универсального Выполнителя при их выполнении.

Например, выполнение строки «СК. добавить ⟨5⟩ в стек» в программе «вычисление» можно описать так: Универсальный Выполнитель (УВ), дойдя до этой строки в тексте программы, анализирует систему предписаний исполнителя СК, т. е. ищет строку, которая вне скобок ⟨ и ⟩ *в точности* совпадает со строкой программы. Обнаружив такую строку, т. е. строку «добавить ⟨вх: число⟩ в стек», УВ устанавливает входной объект «число» в состояние 5 и «нажимает нужную кнопку» — обращается с соответствующим предписанием к Стековому калькулятору.

В соответствии с принятой в программировании терминологией будем называть число 5 в тексте программы *фактическим параметром*, объект «число» в системе предписаний — *формальным параметром*, а акт установки объекта «число» в состояние 5 — *установкой соответствия формальных и фактических параметров*.

Управляющие конструкции. Приведенные в разд. 1 программы были весьма примитивными: каждая из них описывала фиксированную последовательность вызовов предписаний, совершенно не зависящую от состояния исполнителя, выполняющего эти предписания. Программа может, однако, описывать и более сложные последовательности, зависящие от тех или иных условий. Рассмотрим пример:

```
программа осторожный шаг
  дано      : | Путник где-то в квадранте
. получить:
. --------
. если Путник.впереди свободно
  .то
. .     Путник.сделать шаг
. конец если
конец программы
```

Эта программа содержит новую форму записи — управляющую конструкцию

```
если ⟨условие⟩
.то
.    ⟨действия⟩
конец если
```

Конструкция называется управляющей, так как сама по себе она не приводит ни к каким вызовам предписаний, а лишь управляет порядком таких вызовов. Хотя смысл конструкции **если — то — конец если** интуитивно ясен, поясним его, изобразив схему работы Универсального Выполнителя при выполнении этой конструкции (рис. 2.2). (Справа на рис. 2.2 изображена краткая форма, которой мы будем пользоваться в дальнейшем.)

Таким образом, при выполнении программы «осторожный шаг» Универсальный Выполнитель сначала анализирует условие между словами **если** и **то**, т. е. обращается к Путнику с предписанием

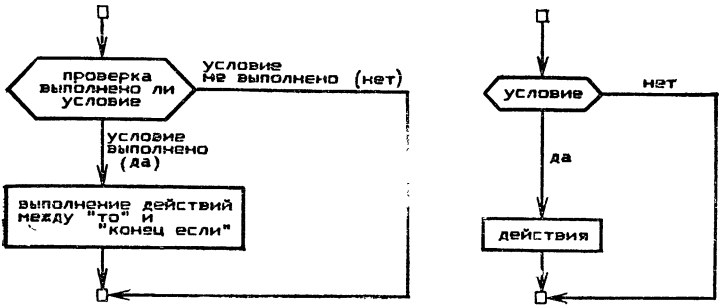


Рис. 2.2

«впереди свободно». Так как это — предписание, вырабатывающее значение типа да/нет, то Путник отвечает либо **да**, либо **нет**. Если условие выполнено (Путник ответил **да**), то Универсальный Выполнитель выполняет строки между **то** и **конец если** (т. е. командует Путнику «сделать шаг»). В противном случае текст между **то** и **конец если** пропускается.

Строго говоря, программа «осторожный шаг» описывает не одну, а две последовательности вызовов предписаний. Конкретная последовательность определяется лишь при выполнении этой программы. Таким образом, фразу «программа — формальное описание последовательности вызовов предписаний», следует понимать именно в таком динамическом смысле. В рамках метафоры Универсального Выполнителя мы можем понимать программу как программу его действий — формальный текст, полностью описывающий поведение Универсального Выполнителя с учетом всех тех условий, которые могут сложиться при выполнении программы.

Кроме конструкции **если — то — конец если** мы будем использовать и другие конструкции, управляющие порядком вызовов предписаний. Почти все эти конструкции представлены в табл. 2.1.

Семантику (смысл) этих конструкций мы будем объяснять, описывая работу Универсального Выполнителя при их выполнении.

Таблица 2.1

Основные управляющие конструкции

Четыре конструкции выбора

если ⟨условие⟩ . **то** . ⟨действия⟩ **конец если**	**если** ⟨условие⟩ . **то** ⟨действия1⟩ . **иначе** ⟨действия2⟩ **конец если**
выбор . **при** ⟨условие1⟩ ⇒ ⟨действия 1⟩ . **при** ⟨условие2⟩ ⇒ ⟨действия 2⟩ . . . . . **при** ⟨условие N⟩ ⇒ ⟨действияN⟩ . **конец выбора**	**выбор** . **при** ⟨условие1⟩ ⇒ ⟨действия1⟩ . **при** ⟨условие2⟩ ⇒ ⟨действия2⟩ . . . . . **при** ⟨условиеN⟩ ⇒ ⟨действияN⟩ . **иначе** ⇒ ⟨действия⟩ **конец выбора**

Шесть конструкций цикла

цикл . **выполнять** . ⟨действия⟩ **конец цикла**	**цикл пока** ⟨условие⟩ . **выполнять** . ⟨действия⟩ **конец цикла**
цикл K **раз** . **выполнять** . . ⟨действия⟩ **конец цикла**	**цикл** K **раз** . **пока** ⟨условие⟩ . **выполнять** . ⟨действия⟩ **конец цикла**
цикл для каждого X **из** M . **выполнять** . . ⟨действия⟩ **конец цикла**	**цикл для каждого** X **из** M . **пока** ⟨условие⟩ . **выполнять** . ⟨действия⟩ **конец цикла**

В каждой из этих шести конструкций цикла среди «действий» может встретиться одна или несколько конструкций

выход из цикла

Конструкции **утверждение, отказ и ничего не делать**

утверждение: ⟨условие⟩ или **утв:** ⟨условие⟩
отказ
ничего не делать

Конструкции выбора. Семантика конструкций выбора ясна из схем рис. 2.3.

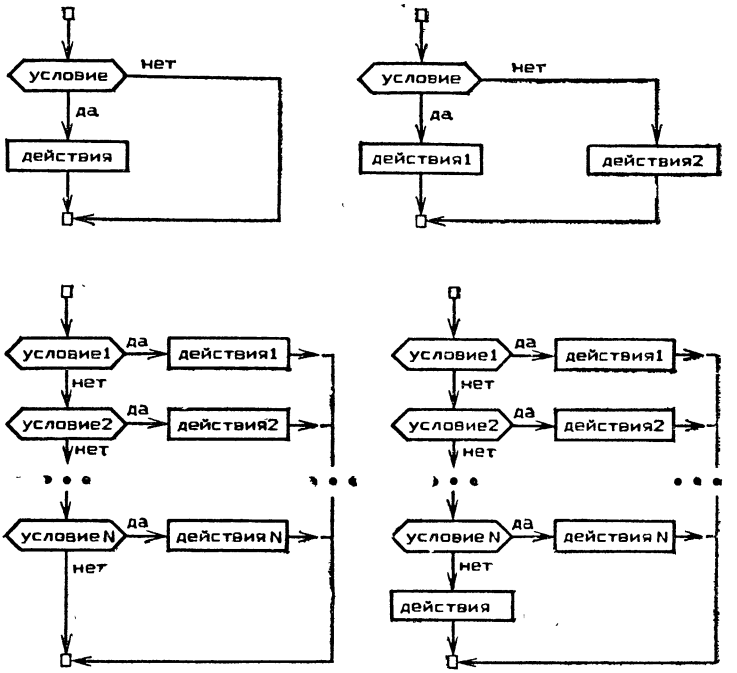


Рис. 2.3. Конструкции выбора: а) если — то — конец если; б) если — то — иначе — конец если; в) выбор — конец выбора; г) выбор — иначе — конец выбора

Циклы. Конструкция **цикл выполнять — конец цикла** вызывает повторение *тела цикла* (т. е. действий, записанных между словами **выполнять** и **конец цикла**), вообще говоря, до бесконечности. Выполнение такого цикла может закончиться лишь в двух случаях:

— возникла ситуация «отказ» при выполнении одного из действий (в этом случае выполнение программы вообще прекращается);

— встретилась конструкция **выход из цикла** (в этом случае текст до слов **конец цикла** пропускается и далее выполняются строки, записанные после конструкции цикла).

Конструкция **цикл пока — выполнять — конец цикла** выполняется в соответствии со схемой, изображенной на рис. 2.4.

Конструкция **цикл К раз выполнять — конец цикла** заставляет Универсальный Выполнитель выполнить тело цикла требуемое число раз (при $К \leqslant 0$ тело цикла не выполняется ни разу). Например, программу обхода квадранта по периметру можно записать так:

```
программа прогулка
. дано      : | препятствия не прилегают к стенам квадранта
. получить: | обойти квадрант по периметру
. --------
. Путник.начать работу
. цикл 4 раз выполнять
. . Путник.шагать до упора
. . Путник.повернуться направо
. конец цикла
. Путник.кончить работу
конец программы
```

Обратите внимание, что эта программа и программа «прогулка» из разд. 1 — это *два разных формальных описания* (т. е. *две разные программы*) *одной и той же последовательности вызовов предписаний* исполнителя Путник.

При выполнении конструкции **цикл К раз пока — выполнять — конец цикла** Универсальный Выполнитель пытается повторить тело цикла требуемое число раз. Однако в отличие от предыдущей конструкции *перед каждым* выполнением тела цикла Универсальный Выполнитель анализирует условие после слова **пока**. Если условие выполнено, то тело цикла очередной раз выполняется. Если же условие не выполнено, то выполнение конструкции цикла завершается и выполняются действия, записанные после слов **конец цикла**.

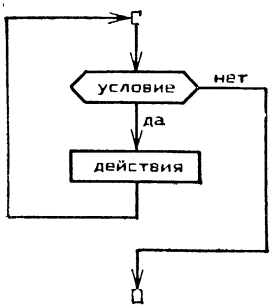


Рис. 2.4. Конструкция **цикл — пока — выполнять — конец цикла**

Конструкции **цикл для каждого...** и **цикл для каждого — пока...** приведены для полноты картины. Они встретятся и будут объяснены позже.

Наконец, встретив конструкцию **выход из цикла**, Универсальный Выполнитель пропускает текст до слов **конец цикла** и выполняет действия, следующие далее.

Утверждение, отказ и ничего не делать. Конструкция **утверждение**, или **утв**, приводит к проверке записанного в ней условия. Если условие выполнено, то выполнение программы нормально продолжается. Если условие не выполнено, то Универсальный Выполнитель считает, что возникла ситуация «отказ», и прекращает выполнение программы. Аналогичным образом он прекращает выполнение программы, если встречает конструкцию **отказ**.

Конструкция **ничего не делать**, как и следует из ее названия, не приводит ни к каким действиям Универсального Выполнителя. Ис-

пользуется эта конструкция исключительно для повышения понятности программы при чтении ее человеком. Например, мы можем записать программу «осторожный шаг» в виде:

```
программа осторожный шаг
. дано     : | Путник где-то в квадранте
. получить:
. --------
.  если Путник.впереди свободно
. . то
. .    Путник.сделать шаг
. . иначе
. .    ничего не делать
.  конец если
конец программы
```

Несколько примеров программ с управляющими конструкциями

```
программа повернуться спиной к препятствию
. дано     : | Путник в клетке около препятствия
. получить: | Путник там же, спиной к препятствию
. --------
.  выбор
. . при Путник.слева    занято ⇒ Путник.повернуться направо
. . при Путник.справа   занято ⇒ Путник.повернуться налево
. . при Путник.впереди  занято ⇒ Путник.повернуться налево
. .                              Путник.повернуться налево
. . иначе ⇒ ничего не делать
.  конец выбора
конец программы
```

```
программа в левый верхний угол стола
. дано     : | Резчик металла включен
. получить: | резак в левом верхнем углу стола
. --------
.  цикл пока РМ.резак не у верхнего края стола выполнять
. . РМ.шаг вверх
.  конец цикла
.    утв: РМ.резак у верхнего края стола
.  цикл пока РМ.резак не у левого края стола выполнять
. . РМ.шаг влево
.  конец цикла
.    утв: РМ.резак у верхнего края стола и
.         РМ.резак у левого края стола
конец программы
```

```
программа периметр квадранта
. дано      : | препятствия не прилегают к стенам квадранта
.             | Путник в левом верхнем углу квадранта
. получить: | Путник в той же клетке
.             | в Счетчике — периметр квадранта
. --------
. Путник.повернуться на восток
. Счетчик.установить в нуль
. цикл 4 раз выполнять
. . утв: | В Счетчике — сумма длин пройденных стен квадранта
. . Счетчик.увеличить на единицу
. . цикл пока Путник.впереди свободно выполнять
. . . Путник.сделать шаг
. . . Счетчик.увеличить на единицу
. . конец цикла
. . Путник.повернуться направо
. конец цикла
конец программы
```

Замечание. Теперь мы можем придать более точный смысл терминам «условие» и «действия».

Условие, как видно из примера программы «в левый верхний угол стола», может состоять из нескольких вызовов предписаний, вырабатывающих значения типа да/нет. Эти вызовы должны быть связаны между собой логическими связками **и**, **или**, **не** (сокращение от «неверно, что»). Допустимы также скобки для явного указания порядка выполнения. В условиях вида «А **и** В» и «А **или** В» вначале проверяется А. Если этого достаточно для проверки всего условия (в «А и В» А не выполнено, в «А или В» А выполнено), то В вообще не проверяется (предписание не вызывается).

Действия, как видно из примера программы «периметр квадранта», могут содержать не только вызовы предписаний, но и управляющие конструкции, действия которых в свою очередь также могут содержать управляющие конструкции, и т. д. Таким образом, конструкции можно вкладывать друг в друга и, вообще говоря, неограниченно. На практике, однако, стараются, чтобы уровень вложенности конструкций не превышал 2—3, так как, чем выше уровень вложенности, тем труднее понять программу.

И условия, и действия могут быть пустыми. Пустое условие считается выполненным. Обычно пустые условия используются вместе с комментариями, в которых записываются *неформальные условия* (см., например, программу «периметр квадранта»). Такие комментарии облегчают чтение и понимание программы человеком, но игнорируются Универсальным Выполнителем на этапе выполнения программы.

Наконец, заметим, что в отличие от программы «прогулка» в программе «периметр квадранта» мы не начинаем и не кончаем работы ни с Путником, ни со Счетчиком. Вообще говоря, информация о том, начата ли уже работа с исполнителем и надо ли кончать ее, должна записываться в дано/получить. Мы, однако, вместо этого установим следующее соглашение: если имя исполнителя или информация о состояниях его глобальных объектов фигурирует в дано/получить, то ни начинать, ни кончать работу с ним не надо.

ЗАДАЧИ И УПРАЖНЕНИЯ

Придумайте имена программ и напишите программы со следующими дано/получить:

1.	дано :	лист металла имеет прямоугольную форму, границы проходят по границам клеток, резак над металлом
	получить:	резак в левом верхнем углу листа металла
2.	дано :	Путник стоит лицом на восток в клетке над северо-западным углом прямоугольного препятствия, которое не прилегает к стенам квадранта
	получить:	положение и ориентация Путника не изменились, в Счетчике — периметр препятствия
3.	дано :	Путник где-то в квадранте
	получить:	Путник прошагал прямо до упора, в Счетчике — число пройденных Путником клеток (число шагов)
4.	дано :	лист металла занимает весь стол Резчика. В первом (верхнем) ряду какие-то клетки прорезаны, а какие-то нет. Резак в левом верхнем углу и поднят
	получить:	во втором ряду прорезаны те же клетки, что и в первом. Резак в самой левой клетке второго ряда
5.	дано :	лист металла занимает весь стол Резчика. В первом (верхнем) ряду какие-то клетки прорезаны, а какие-то нет. Резак в левом верхнем углу и поднят
	получить:	во всех рядах прорезаны те же клетки, что и в первом ряду. Резак в левой клетке последнего ряда
6.	дано :	лист металла занимает весь стол Резчика. Резак в левом верхнем углу и поднят
	получить:	в листе прорезано отверстие (трафарет) в форме числа 4444

3. Основная задача программирования и технология «сверху вниз»

Программы и их вызовы. Пусть требуется написать программу, которая переводит резак Резчика металла из произвольной точки над столом в левый верхний угол прямоугольного листа металла,

расположенного параллельно краям стола. Решая такую задачу, мы сначала, естественно, должны придумать общую стратегию перемещения резака, выбрать, как и куда его двигать. Первым неформальным описанием общей стратегии решения может быть, например, следующее: «Сначала переведем резак в левый верхний угол стола, а затем будем вести его вниз по первой колонке клеток и в каждой клетке проверять, есть под резаком металл или нет. Если мы найдем металл, то это и есть левый верхний угол листа металла, и больше ничего делать не надо. Если же в первой колонке мы металла не обнаружим, то проверим сверху вниз вторую колонку, затем третью и т. д., пока не найдем металл». Этот план мы можем записать более формально в виде

```
программа в левый верхний угол металла
. дано      : | на столе Резчика есть металл, резак поднят
. получить: | резак в левом верхнем углу листа металла
. --------
. в левый верхний угол стола
. цикл пока РМ.резак не над металлом выполнять
. . перейти в следующую клетку
. конец цикла
конец программы
```

Фактически мы выделили в исходной задаче две подзадачи («в левый верхний угол стола» и «перейти в следующую клетку») и записали решение задачи, считая, что подзадачи решены (или будут решены позже). Этот подход аналогичен использованию лемм в математике. Программа «в левый верхний угол стола» была приведена в предыдущем разделе. Напишем программу «перейти в следующую клетку»:

```
программа перейти в следующую клетку
. дано      :
. получить: | резак в следующей клетке, если перебирать
.           | их в порядке сверху вниз и слева направо
. --------
.  если РМ.резак не у нижнего края стола то РМ.шаг вниз
. . иначе
. . цикл пока РМ.резак не у верхнего края стола выполнять
. . . РМ.шаг вверх
. . конец цикла
. . РМ.шаг вправо
. конец если
конец программы
```

С чисто программистской точки зрения приведенный выше пример иллюстрирует новую возможность языка программирования —

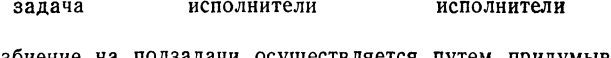
возможность *вызова* из одной программы других. Встретив, например, в программе «в левый верхний угол металла» строку «перейти в следующую клетку», Универсальный Выполнитель приостанавливает выполнение программы «в левый верхний угол металла», находит и выполняет программу «перейти в следующую клетку», после чего продолжает выполнение программы «в левый верхний угол металла».

Естественно, что для того, чтобы Универсальный Выполнитель смог действительно перевести резак в левый верхний угол металла, ему надо дать текст, содержащий все три программы, а также указать, какую именно программу мы хотим выполнить.

Основная задача программирования. Как видно из приведенного выше примера, при создании программ удобно разбить исходную задачу на подзадачи и записать программу, считая, что подзадачи решены. После этого можно для каждой подзадачи создавать программу отдельно (такие программы для решения подзадач часто называют *подпрограммами*). Естественно, что подзадачу можно в свою очередь разбивать на подподзадачи и т. д. Чем сложнее наша исходная задача, тем больше будет уровней иерархии и тем больше в конце концов получится отдельных программ. При современной методике программирования решение задачи среднего класса сложности может насчитывать несколько сотен таких программ. Чтобы среди них можно было ориентироваться, программы группируют. Для такого объединения программ в группы существует специальная конструкция **исполнитель — конец исполнителя**.

То, что эта конструкция называется исполнителем, отнюдь не случайное совпадение. Выполнение любой программы состоит в проведении каких-то действий над какими-то объектами (например, при выполнении программы «в левый верхний угол металла» изменяется положение резака у Резчика металла). Вызов программы — это приказ на выполнение этих действий. Таким образом, группу программ можно (и это удобно) считать исполнителем с некоторой системой предписаний. Про самого исполнителя в этом случае говорят, что он *реализован* группой программ *на базе* каких-то других исполнителей, где каждая программа *реализует* некоторое предписание.

Процесс разбиения исходной задачи на подзадачи с учетом группировки подзадач можно изобразить так:



т. е. разбиение на подзадачи осуществляется путем придумывания новых, *промежуточных* исполнителей. После этого отдельно решается задача а) — написание программы с использованием (или *на базе*)

этих вновь придуманных исполнителей и отдельно б) — *реализация* промежуточных исполнителей (т. е. написание соответствующих программ) с использованием заданных заранее *базовых* исполнителей. Каждая из этих задач может в свою очередь разбиваться на подзадачи и т. д.

Таким образом, за исключением, быть может, самого первого шага, перед нами будет стоять не задача написания одной отдельно взятой программы, а задача реализации одного исполнителя на базе каких-то других. Более того, на практике и исходная задача часто ставится не как написание отдельной программы, а как создание (реализация) того или иного исполнителя.

Заметим, что любого исполнителя можно рассматривать и описывать с двух точек зрения: *внешней* и *внутренней*. При использовании исполнителя нас интересуют его внешние свойства: какие у него есть предписания, что получается в результате их выполнения и т. д. Описание этих свойств, достаточное для использования исполнителя, называется *внешним описанием исполнителя*. Внешнее описание исполнителя позволяет им пользоваться, но ничего не говорит о его внутреннем устройстве (подобно тому как внешнее описание магнитофона содержит информацию о расположении кнопок «воспроизведение» и «запись», но не содержит информации о числе моторов).

В терминах исполнителей можно сформулировать *основную задачу программирования*:

Основная задача программирования:

дано : | Внешнее описание искомого исполнителя А,
| который надо создать, и
| внешнее описание базового исполнителя Е,
| который считается существующим

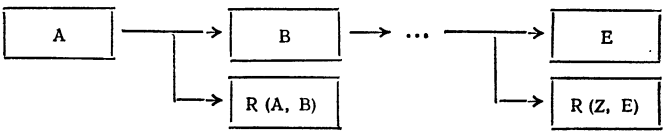
получить : | Реализацию R (A, E) — текст на языке
| программирования, описывающий
| функционирование
| А в терминах Е и позволяющий изготовить
| А при наличии Е

При решении производственных задач текст R(A, E) оказывается ужасающе длинным — от нескольких тысяч до нескольких сотен тысяч строк. Создание такого текста представляет собой трудную задачу, для решения которой в настоящее время разработано несколько подходов. Один из таких подходов — это технология «сверху вниз».

Технология программирования «сверху вниз». *Технология* (*греч.* ремесло + наука) — совокупность знаний о способах и средствах проведения производственных процессов. *Программирование* — про-

цесс создания (производства) текста, описывающего исполнителя с заданными свойствами.

Мы уже видели, что основную задачу программирования удобно решать, разбивая ее на подзадачи с помощью промежуточных исполнителей. Технология программирования «сверху вниз» предлагает такое разбиение вести от исполнителя А (сверху) к исполнителю Е (вниз), т. е. при придумывании промежуточных исполнителей руководствоваться скорее тем, *что* надо сделать, нежели тем, на какой базе. Технологическая цепочка программирования «сверху вниз» выглядит следующим образом:



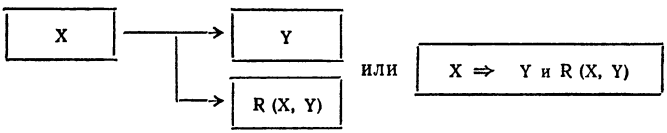
где B, ..., Z — промежуточные исполнители, придуманные в процессе решения задачи (а точнее, их внешние описания). Стрелки на схеме показывают последовательность действий человека.

При этом решение основной задачи программирования — текст R(A, E) — получается в виде *композиции реализаций*

$$R(A, E) = R(A, B) * R(B, C) * \ldots * R(Z, E).$$

Композиция реализаций — это объединение текстов R(A, B), R(B, C) и т. д. в указанном порядке.

Шаг декомпозиции. При использовании технологии «сверху вниз» производство текста R(A, E) состоит из последовательного применения операции



начиная с внешнего описания искомого исполнителя А и до тех пор, пока не дойдем до существующего базового исполнителя Е.

Эта операция называется *шагом декомпозиции*, является сердцем технологии «сверху вниз» и состоит в следующем:

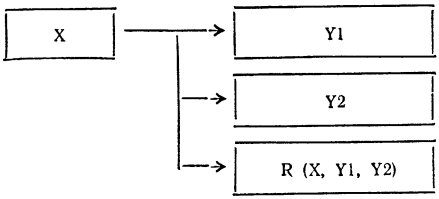
Шаг декомпозиции:
дано : | Внешнее описание исполнителя X
получить : | Реализацию X на базе придуманного нами
| исполнителя Y и одновременно внешнее
| описание этого исполнителя Y

При проведении шага декомпозиции (т. е. при придумывании Y) следует руководствоваться двумя требованиями:

1) л о к а л ь н о е: как можно большую часть работы надо переложить на плечи нового исполнителя — на Y;

2) г л о б а л ь н о е: шаг декомпозиции должен приближать нас к E (т. е. Y должен быть «ближе» к E, чем X).

З а м е ч а н и е 1. Как правило, при проведении шага декомпозиции исполнитель X реализуется на базе не одного исполнителя Y, а нескольких, например Y1, Y2. В результате шага декомпозиции возникают внешние описания этих исполнителей и одна реализация R(X, Y1, Y2):



Таким образом, в формуле для шага декомпозиции следует понимать Y как обозначение вектора (Y1, ..., YN). Если нарисовать технологическую цепочку без использования векторных обозначений, то стрелки, соединяющие внешние описания исполнителей, будут напоминать дельту большой реки, впадающей в «море» базовых исполнителей E = (E1, ..., EK) (рис. 3.1).

З а м е ч а н и е 2. При проведении шага декомпозиции можно использовать исполнителей, внешние описания которых либо были заданы (например, E1, ..., EK), либо получены ранее, на предыдущих шагах декомпозиции.

З а м е ч а н и е 3. В результате проведения очередного шага декомпозиции внешние описания придуманных ранее исполнителей могут быть уточнены или изменены. Более того, дойдя до какого-то места, мы можем увидеть, что придуманная иерархия промежуточных исполнителей неудачна, что какой-то исполнитель не реализуется или реализуется чересчур сложно. В этом случае приходится возвращаться на сколько-то шагов декомпозиции назад и начинать сначала. Таким образом, программирование является, вообще говоря, итеративным (повторяющимся) процессом. Приведенная выше технологическая цепочка является идеализацией и описывает процесс программирования без таких возвратов.

З а м е ч а н и е 4. Технология «сверху вниз» включает в себя еще и способы и средства контроля качества продукции (т. е. текстов R(A, B), R(B, C) и т. д.) на всех стадиях ее изготовления. Хотя на практике затраты на контроль качества — так называемые *тестирование* и *отладку* — зачастую соизмеримы с затратами на создание продукции, мы эти вопросы рассматривать не будем.

Замечание 5. Имея реализацию R(A, E) и базового исполнителя E, можно подключить E к Универсальному Выполнителю и дать Универсальному Выполнителю текст R(A, E). В результате получится реально функционирующий исполнитель A. Мы, однако,

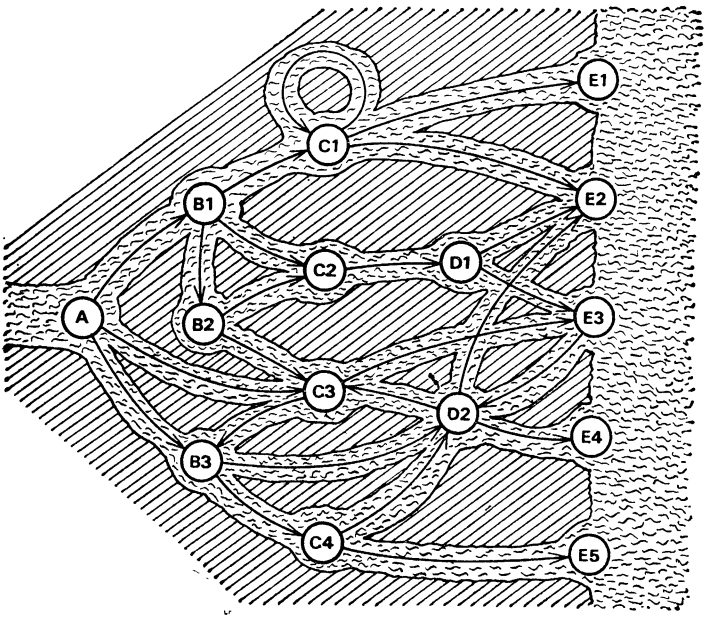


Рис. 3.1. Иерархия исполнителей в виде «дельты большой реки»

занимаясь программированием, будем интересоваться вопросами создания не самого исполнителя A, а лишь текста R(A, E).

Пример разработки программы по технологии «сверху вниз» *). Заставим теперь наш станок с ЧПУ — Резчика металла — заняться делом. Пусть требуется из листа металла нужного формата изготовить сито со структурой, изображенной на рис. 3.2.

Будем считать, что сито должен изготовить специальный исполнитель «Резчик сита» с одним-единственным предписанием «изготовить сито». Реализовать этого исполнителя (A) надо на базе базового исполнителя «Резчик металла» (E) — его описание было приведено в разд. 1.

Размеры каждой ячейки сита — 60*60 шагов (клеток) Резчика металла. В крайних ячейках верхнего и нижнего рядов проделы-

*) Термин «разработка программы» сложился исторически и означает реализацию программы или исполнителя на базе других исполнителей.

ваются четыре отверстия для крепления сита. В крайней правой ячейке нижнего ряда, кроме того, проделывается отверстие, которое обеспечивает правильную ориентацию сита при монтаже. Во

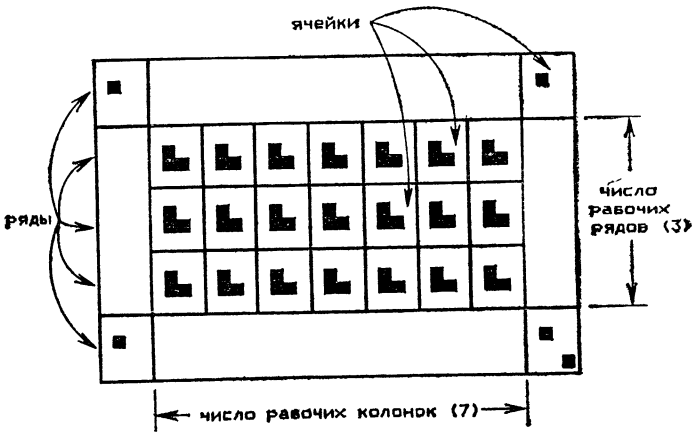


Рис. 8.2. Сито

всех внутренних ячейках проделываются рабочие отверстия. Таким образом, имеем три типа ячеек с отверстиями (рис. 3.3).



Рис. 8.3. Типы ячеек с отверстиями

Наша основная задача — получить реализацию (текст) R(Резчик сита, Резчик металла). Для получения этой реализации мы по технологии «сверху вниз» должны сделать шаг декомпозиции.

Резчик сита ⇒ X и R (Резчик сита, X).

Попробуем придумать X. Первое требование, которым надо руководствоваться при таком придумывании, — переложить как можно большую часть работы на X. Идеальным с этой точки зрения является исполнитель, который умеет делать сито целиком от начала до конца сам. Однако это и есть Резчик сита. Для того чтобы не получить тавтологии, надо помнить и про второе требование: X должен быть «ближе» к E, чем A_2.

Посмотрим на сито и постараемся найти в нем максимально крупные (первое требование) образования, структуры, которые, однако, были бы меньше, чем сито в целом (второе требование). Легко видеть, что сито состоит из ячеек, рядов ячеек и колонок ячеек. Поскольку надо искать максимально крупные образования, то мы должны выбирать между рядами и колонками ячеек, а сами ячейки, как более мелкие, не рассматривать. Для выбора между рядами и колонками у нас нет никаких аргументов «за» и «против», поэтому возьмем наобум первое, что придет в голову, скажем ряды.

Попробуем теперь изготовить сито, считая, что у нас есть исполнитель, обрабатывающий ряды целиком (назовем этого исполнителя «Обработчик рядов»). Как это делать? Разумно обрабатывать последовательно ряд за рядом сверху вниз, начиная с верхнего. Будем считать, что в начале обработки ряда резак находится в левой верхней клетке ряда (в *начале ряда*). Для того чтобы при переходе от ряда к ряду резак не надо было двигать, будем считать, кроме того, что в конце обработки ряда резак располагается в начале следующего ряда. Это положение — клетку под левой нижней клеткой ряда — назовем *концом ряда*. Заметим, что все наши «будем считать» — это продолжение придумывания исполнителя «Обработчик рядов», а именно уточнение семантики его будущих предписаний. Сами предписания будем вводить «по ходу» там, где они понадобятся.

```
исполнитель Резчик сита
. СП:
.  1. изготовить сито
.
. константы:
.  число рабочих рядов = 3
.
. используемые исполнители:
.  Обработчик рядов
конец описаний | ------------------------------
программа изготовить сито
. дано      : | резак в начале (левой верхней клетке) заготовки и
.             | поднят
. получить: | сито изготовлено, резак в конце заготовки
.             | (в клетке под левым нижним углом заготовки)
. --------
. Обработчик рядов.начать работу
. Обработчик рядов.обработать верхний ряд
. цикл число рабочих рядов раз выполнять
. . Обработчик рядов.обработать рабочий ряд
```

```
. конец цикла
. Обработчик рядов.обработать нижний ряд
. Обработчик рядов.кончить работу
конец программы
конец исполнителя
```

Мы здесь впервые встречаемся с конструкцией **исполнитель — конец исполнителя** и с понятием константы. Поэтому, прежде чем идти дальше, поясним их.

Конструкция **исполнитель — конец исполнителя** состоит из двух разделов. В первом разделе (между строками **исполнитель** и **конец описаний**) приводится описание исполнителя в целом. Второй раздел (между строками **конец описаний** и **конец исполнителя**) состоит просто из последовательно расположенных программ, реализующих предписания исполнителя. Раздел описаний может содержать различные подразделы. Подраздел **СП** (система предписаний) является обязательным. Смысл подразделов полностью соответствует их названиям. Подраздел **константы** используется для того, чтобы дать имена константам, используемым в исполнителе. После этого всюду ниже, вплоть до строки **конец исполнителя**, вместо констант можно использовать их имена. Например, встретив слова «число рабочих рядов» в процессе выполнения программы «изготовить сито», Универсальный Выполнитель заменит их на значение константы (3). Другие возможные подразделы в разделе описаний исполнителя будут пояснены по мере их появления.

Итак, вернемся к процессу разработки программы. Мы совершили первый шаг декомпозиции:

```
Резчик сита ⇒ Обработчик рядов и
              R (Резчик сита, Обработчик рядов)
```

и получили реализацию R(Резчик сита, Обработчик рядов) (это текст выше) и внешнее описание исполнителя «Обработчик рядов» (выпишем ниже). Поскольку Обработчик рядов еще не является базовым, то надо совершить очередной шаг декомпозиции:

```
Обработчик рядов ⇒ X и R (Обработчик рядов, X)
```

И опять, для придумывания X мы должны найти наиболее крупные образования, структуры, из которых состоят ряды. Это, несомненно, ячейки. Поэтому назовем следующего исполнителя «Обработчик ячеек». Будем обрабатывать ячейки в ряду слева направо и поэтому решим, что в начале работы всех предписаний Обработ-

ника ячеек резак будет находиться в *начале* (левой верхней клетке) *ячейки*, а в конце работы — в *конце ячейки* (рис. 3.4).

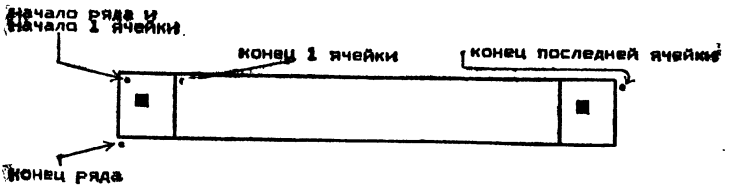


Рис. 3.4

Попробуем реализовать предписание «обработать верхний ряд»:

программа обработать верхний ряд
- **дано** : | резак в начале верхнего ряда
- **получить:** | верхний ряд обработан, резак в его конце
- ---------
- Обработчик ячеек.крепежная ячейка
- **цикл** число рабочих колонок **раз выполнять**
- Обработчик ячеек.пустая ячейка
- **конец цикла**
- Обработчик ячеек.крепежная ячейка

В этот момент верхний ряд обработан, а резак находится в конце его последней ячейки. Для того чтобы обеспечить выполнение записанного в **получить** условия, надо переместить резак из конца последней ячейки в конец ряда. Эта подзадача возникнет, конечно, и при обработке рабочих рядов, и при обработке последнего ряда. Поэтому мы реализуем такое перемещение отдельной программой, а здесь запишем вызов этой программы:

- встать в конец ряда

конец программы

Предписания «обработать рабочий ряд» и «обработать нижний ряд» реализуются аналогично. Запишем реализацию исполнителя «Обработчик рядов» в целом:

исполнитель Обработчик рядов
СП:
- 1. начать работу
- 2. обработать верхний ряд
- 3. обработать рабочий ряд
- 4. обработать нижний ряд

```
.  — встать в конец ряда
.
.  5. кончить работу
.
. константы:
.  число рабочих колонок = 7
.
. используемые исполнители:
.  Обработчик ячеек
конец описаний | ------------------------------

программа начать работу == Обработчик ячеек. начать работу
программа обработать верхний ряд
. дано      : | резак в начале верхнего ряда
. получить: | верхний ряд обработан, резак в его конце
. --------
.  Обработчик ячеек.крепежная ячейка
.  цикл число рабочих колонок раз выполнять
.  .   Обработчик ячеек.пустая ячейка
.  конец цикла
.  Обработчик ячеек.крепежная ячейка
.  встать в конец ряда
конец программы

программа встать в конец ряда
. дано      : | резак в конце последней ячейки ряда
. получить: | резак в конце ряда
. --------
.  Обработчик ячеек.вниз на ячейку
.  цикл число рабочих колонок + 2 раз выполнять
.  .   Обработчик ячеек.влево на ячейку
.  конец цикла
конец программы
```

Предписания «обработать рабочий ряд», «обработать нижний ряд» и «кончить работу» реализуйте самостоятельно.

```
конец исполнителя | ==============================
```

Минус перед «встать в конец ряда» в разделе описаний означает, что эта программа не является предписанием исполнителя «Обработчик рядов» и не может быть вызвана извне. Напомним, что программа «встать в конец ряда» возникла как решение подзадачи при реализации Обработчика рядов и используется локально только внутри этого исполнителя.

Программа «начать работу» записана в краткой форме, которая эквивалентна следующей:

```
программа начать работу
. дано :
. получить:
. --------
. Обработчик ячеек.начать работу
конец программы
```

Мы совершили еще один шаг декомпозиции:

> Обработчик рядов ⇒ Обработчик ячеек и
> R (Обработчик рядов, Обработчик ячеек)

и получили реализацию R(Обработчик рядов, Обработчик ячеек) и внешнее описание исполнителя «Обработчик ячеек» (сейчас выпишем). Поскольку исполнитель «Обработчик ячеек» еще не является базовым, надо совершить следующий шаг:

> Обработчик ячеек ⇒ X и R (Обработчик ячеек, X)

И опять, для придумывания X мы должны найти наиболее крупные образования, структуры, из которых состоят ячейки. Легко видеть, что это еще не клетки Резчика металла, а более крупные единицы, имеющие размеры 15*15 шагов Резчика металла; назовем их «квадратиками» (рис. 3.5).

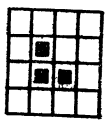


Рис. 3.5. Рабочая ячейка, разбитая на квадратики

Поскольку никакого явно выделенного порядка обработки квадратиков при обработке, например, рабочей ячейки не предвидится, то будем считать, что в начале и в конце обработки квадратика резак находится в его начале (левой верхней клетке).

```
исполнитель Обработчик ячеек
. СП:
.  1. начать работу
.  2. крепежная ячейка
.  3. монтажная ячейка
.  4. рабочая ячейка
.  5. пустая ячейка
.
.  6. вниз на ячейку
.  7. влево на ячейку
.  8. вправо на ячейку
```

```
. 9. вверх на ячейку
. 10. кончить работу
.
. обозначения:
. ширина ячейки == 4 | квадратика
. высота ячейки == 4 | квадратика
.
. используемые исполнители:
. Обработчик квадратиков
конец описаний | ------------------------------
```

Подраздел **обозначения** аналогичен подразделу **константы**. Встретив в тексте слова «ширина ячейки», Универсальный Выполнитель заменит их на текст, который записан после знака ==, в данном случае на 4. Таким образом, для Универсального Выполнителя обозначения и константы — суть одно и то же. Разница между этими понятиями существенна только для человека, который читает или изменяет текст программы. Запись

```
константы:
  число рабочих колонок = 7
```

подразумевает, что достаточно заменить 7 на другое число и *без каких-либо изменений текстов программ* исполнитель будет правильно работать с ситом новых размеров. Если это не так, т. е. если изменение значения может потребовать изменения текстов программ, то соответствующую величину надо задавать не в подразделе **константы**, а в подразделе **обозначения**.

```
программа начать работу==Обработчик квадратиков.начать работу
программа крепежная ячейка
. дано      : | резак в начале ячейки
. получить: | прорезано крепежное отверстие, резак в конце ячейки
. --------
. Обработчик квадратиков.вправо на квадратик
. Обработчик квадратиков.вниз на квадратик
. Обработчик квадратиков.прорезать квадратик | Положение реза-
                                               | ка не меняется
.
. Обработчик квадратиков.вверх на квадратик
. Обработчик квадратиков.вправо на квадратик
. Обработчик квадратиков.вправо на квадратик
. Обработчик квадратиков.вправо на квадратик
конец программы

программа вниз на ячейку
. дано      : | резак в начале ячейки
. получить: | резак сместился на ячейку вниз
```

```
. ---------
. цикл высота ячейки раз
. . выполнять
. .  Обработчик квадратиков.вниз на квадратик
. конец цикла
конец программы
```

Остальные предписания этого исполнителя реализуйте самостоятельно.

конец исполнителя | ==========

Мы совершили еще один шаг декомпозиции:

> Обработчик ячеек ⇒ Обработчик квадратиков и
> R (Обработчик ячеек, Обработчик квадратиков)

и получили реализацию R (Обработчик ячеек, Обработчик квадратиков) и внешнее описание исполнителя «Обработчик квадратиков» (выпишите его самостоятельно). Поскольку исполнитель «Обработчик квадратиков» еще не является базовым, то надо совершить следующий шаг декомпозиции:

> Обработчик квадратиков ⇒ X и R (Обработчик квадратиков, X)

И опять, для придумывания X мы должны найти наиболее крупные образования, структуры, из которых состоят квадратики. Это уже, конечно, клетки Резчика металла.

Проведите этот последний шаг декомпозиции и получите R (Обработчик квадратиков, Резчик металла) самостоятельно.

После этого задача будет полностью решена и будет получена реализация

> R (Резчик сита, Резчик металла) =
> R (Резчик сита, Обработчик рядов)
> * R (Обработчик рядов, Обработчик ячеек)
> * R (Обработчик ячеек, Обработчик квадратиков)
> * R (Обработчик квадратиков, Резчик металла)

Напомним, что под композицией реализаций (знак *) понимается объединение текстов в один в указанном порядке.

ЗАДАЧИ И УПРАЖНЕНИЯ

1. Реализуйте нереализованные предписания Обработчика рядов и Обработчика ячеек (т. е. напишите соответствующие программы).

2. Реализуйте исполнителя «Обработчик квадратиков».

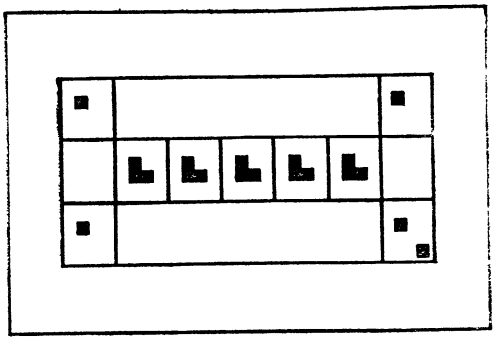


Рис. 3.6. Сито, окаймленное пустыми клетками

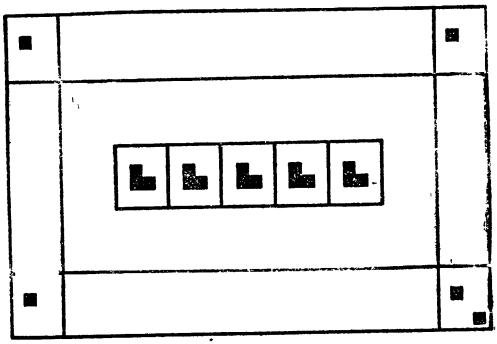


Рис. 3.7. Сито, рабочие клетки которого окаймлены пустыми

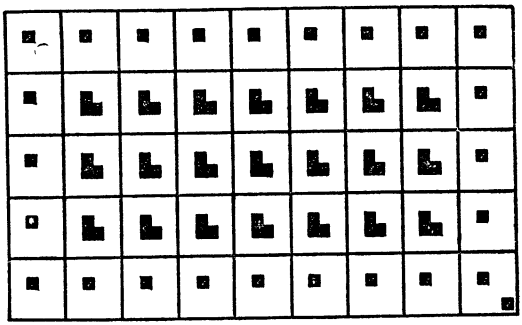


Рис. 3.8. Сито с крепежными отверстиями по всему краю

3. Модифицируйте R(Резчик сита, Резчик металла) так, чтобы из заготовки тех же размеров изготовить сито, аналогичное приведенному выше, но такое, что

а) все сито вокруг окаймлено полосой пустых ячеек (рис. 3.6);

б) рабочие ячейки окаймлены полосой пустых ячеек (рис. 3.7);

в) крепежные отверстия прорезаны в каждой граничной ячейке (рис. 3.8).

4. Напишите программы со следующими дано/получить:

а) **дано** : | резак над металлом, края листа совпадают с краями клеток
получить: | резак в левой верхней клетке металла

б) **дано** : | резак над металлом, края листа совпадают с краями клеток, лист лежит далеко от краев стола
получить: | из листа вырезана заготовка для сита максимального размера (содержащая целое число ячеек)

в*) (задача уровня сложности международных математических олимпиад):

дано : | на столе Резчика металла очень далеко от краев стола лежит заготовка для сита. В некоторых ячейках верхнего ряда и левой колонки прорезаны крепежные отверстия. Известны константы «число рабочих рядов» и «число рабочих колонок».
| Пользоваться можно только Резчиком металла, управляющими конструкциями и подпрограммами
получить: | прорезаны рабочие отверстия на пересечениях тех рядов и колонок, в которых есть крепежные отверстия

4. Процесс разработки программ. Рекурсия, итерация, проектирование цикла с помощью инварианта

Разделение труда — характерная черта современного производства. Самолет, например, проектируется и конструируется одними специалистами, изготавливается вторыми, испытывается третьими, эксплуатируется четвертыми и ремонтируется пятыми. Результатом работы проектировщиков и конструкторов является не реальный самолет, а всего лишь его описание.

Что же является результатом работы программиста? Разберемся в этом на примере программы для изготовления сита. Конечный продукт — изготовленное в металле сито — получается в результате работы исполнителя «Резчик сита». Сам Резчик сита получается в результате соединения нашей программы, Универсального Выполнителя и исполнителя «Резчик металла». Таким образом, ни изготовленное в металле сито, ни сам исполнитель «Резчик сита» не

являются прямым результатом труда программиста. *Результатом труда программиста является программа — текст R(Резчик сита, Резчик металла)*.

Естественно программист — «поставщик» этого текста — должен гарантировать его качество, в частности гарантировать, что изготовленный с его помощью Резчик сита будет резать то, что надо. Но это не единственное требование к качеству текста R(A, E). Другими требованиями являются понятность текста для человека (в частности, возможность быстро убедиться в его правильности), а также возможность его модификации. Последнее требование вызвано тем, что на практике редко удается ограничиться написанием одного варианта R(A, E). Рано или поздно оказывается, что требования к A изменились. Для Резчика сита несколько возможных изменений описаны в упражнении 3 предыдущего раздела. Конечно, можно для каждого нового варианта задачи создавать R(A, E) заново с нуля. Однако, если новый вариант близок к старому, выгоднее взять старый текст R(A, E) и изменить его. Необходимость изменения внешнего описания или реализации исполнителя часто возникает и по внутренним причинам. В процессе решения задачи мы можем, например, придумать промежуточного исполнителя X, реализовать его на базе Y, а в дальнейшем по каким-то соображениям решить, что надо было использовать не X, а близкий к нему исполнитель X′. Тогда придется изменить R(X, Y) и все реализации, использующие X.

Таким образом, программа как минимум должна быть *правильной, понятной* и *легко изменяемой*. Методам эффективного создания таких программ посвящена вся эта книга. Разработка программы «в целом» описана в предыдущем разделе (технология «сверху вниз»). В этом разделе мы займемся более узкой задачей — реализацией одной отдельно взятой программы.

Разбиение задачи на подзадачи. При работе по технологии «сверху вниз» к моменту реализации программы уже известны ее имя и дано/получить. Задача состоит в написании текста, который из любых условий, удовлетворяющих дано, переводит каких-то исполнителей в состояния, удовлетворяющие получить.

Обычно реализация одной программы — это не очень длинный текст (20—30 строк). Было бы замечательно, если бы можно было придумать и записать этот текст единым махом, в результате некоего «озарения». Было бы также замечательно, если бы уже написанный текст можно было мгновенно понять. На практике этого обычно, увы, не происходит: мы придумываем, записываем и понимаем текст программы не сразу, а по частям, в результате целенаправленной деятельности. Качество текста программы во многом определяется тем, в какой степени ключевые моменты нашего обду-

мывания находят отражение (пусть неполное) в окончательном тексте программы.

Рассмотрим еще раз задачу написания программы «в левый верхний угол металла». Самая первая идея решения состояла в том, чтобы сначала перевести резак в левый верхний угол стола, а затем уже попадать в левый верхний угол металла.

Фраза «сначала перевести резак в левый верхний угол стола, затем...», однако, относится к процессу выполнения программы, а не к ее тексту. Если рассуждать в терминах текста программы, то нужно говорить так: программа будет состоять из двух фрагментов — *А* и *Б*. Фрагмент *А* будет обладать следующим свойством: если перед выполнением фрагмента будут выполнены условия, указанные в дано, то после его выполнения будет выполнено условие «резак в левом верхнем углу стола». Фрагмент *Б* будет обладать тем свойством, что если перед его выполнением выполнено условие «резак в левом верхнем углу стола», то после будет выполнено условие, записанное в **получить**:

```
программа в левый верхний угол металла
. дано    : | на столе Резчика есть металл, резак поднят
. получить: | резак в левом верхнем углу листа металла
. --------
.  фрагмент А
.   утв: РМ.резак у левого края стола и
.        РМ.резак у верхнего края стола
.        | т. е. резак в левом верхнем углу стола
.  фрагмент Б
конец программы
```

Тем самым исходная задача реализации программы «в левый верхний угол металла» оказалась разбитой на две подзадачи — реализации фрагментов *А* и *Б*. Для каждой подзадачи (фрагмента программы) условие, которое должно быть достигнуто после выполнения фрагмента, называется *постусловием*, а условие, которое известно до выполнения, — *предусловием*. Таким образом, для фрагмента программы *А* выписанное выше утверждение является постусловием, а дано — предусловием. Для фрагмента программы *Б* получить является постусловием, а утверждение — предусловием. Если всю программу рассмотреть как единый фрагмент, то ее постусловием является получить, а предусловием — дано.

Фрагменты *А* и *Б* можно теперь обдумывать и реализовывать независимо, по очереди. Само разбиение на фрагменты *А* и *Б* зафиксировано в тексте программы в виде строк

утв: РМ.резак у левого края стола **и**
РМ.резак у верхнего края стола
| т. е. резак в левом верхнем углу стола

Это утверждение имеет двоякое назначение. Во-первых, при чтении программы человеком оно дает представление об общем ходе решения, описывает состояние резака, которое будет к этому моменту получено. Во-вторых, формальная часть этого утверждения будет проверяться всякий раз после выполнения фрагмента *А*. Если фрагмент *А* реализован неправильно и в какой-то ситуации утверждение окажется невыполненным, то Универсальный Выполнитель немедленно прекратит выполнение программы и сообщит об ошибке.

Итак, первый вывод состоит в том, что при разработке программы ее следует разбить на основные этапы (подзадачи) и сформулировать условия, которым будут удовлетворять промежуточные между подзадачами положения. Эти условия затем надо рассматривать как пост- и предусловия при разработке частей (фрагментов) программы. Другими словами, *разработка программы в целом состоит в разработке последовательности утверждений (начиная с* ***дано*** *и кончая* ***получить***), *служащих пост- и предусловиями ее частей.*

К реализации каждого из фрагментов можно, в свою очередь, применить тот же подход, т. е. разбить их на подподзадачи и т. п. Разумеется, процесс разбиения на подзадачи должен заканчиваться. Решение очередной подзадачи при этом записывается либо в виде вызова какой-то другой программы, либо непосредственно с помощью управляющих конструкций и иных возможностей Универсального Выполнителя.

Правильность программы в целом зависит от 1) корректности сведения программы к подзадачам и 2) правильности реализации подзадач. Этими вопросами можно заниматься независимо: при сведении задачи к подзадачам можно не думать об их реализации, при проверке реализаций можно не думать, как они используются.

По существу, это — полный аналог использования лемм и теорем в математике. При использовании теоремы надо смотреть на ее формулировку, но отнюдь не на доказательство. Наоборот, при доказательстве теоремы абсолютно неважно, где и как она используется. Понятие корректности можно пояснить так. Пусть при доказательстве теоремы была использована лемма, а в доказательстве леммы есть ошибки. Тогда лемма неверна, теорема в целом неверна, но доказательство теоремы самой по себе — без леммы — корректно. Если лемму в той же формулировке удастся доказать без ошибок, то теорема в целом станет верной. При этом доказательство самой теоремы можно заново не проводить и не проверять: достаточно знать, что оно и раньше было корректным.

Вызовы программ и дано/получить. В случае, если подзадачу *Ф* с известными пост- и предусловиями можно решить с помощью

вызова уже существующей программы или предписания *П*, то фрагмент *Ф* сводится к одной строчке — вызову этой программы. Слова «можно решить» означают, что программа *П* обладает следующими двумя свойствами (назовем их «условия корректности вызова»):

1) предусловие *Ф* ⇒ **дано** *П*, т. е. *П* применима в ситуации, когда выполнено предусловие *П*;

2) **получить** *П* ⇒ постусловие *Ф*, т. е. выполнение *П* гарантирует постусловие.

Возможен, однако, случай, когда для решения подзадачи *Ф* готовой программы нет, но желательно ее создать. Такое выделение подзадачи в отдельную программу обычно происходит, если подзадача: а) достаточно сложна; б) используется при решении нескольких задач; в) естественно относится к какому-то исполнителю (например, любую программу, оперирующую с рядами ячеек в целом, разумно отнести к Обработчику рядов). В этом случае надо не только записать вызов программы, но и придумать ее имя и **дано/получить**. Рассмотрим, как это делается на примере фрагмента *Б*.

Выбор имени. Прежде всего для этой программы нужно придумать имя, коротко описывающее назначение и условия применения программы. Проблема выбора имени тем актуальнее, чем больше размер программы. Если имена предписаний промежуточных исполнителей не будут достаточно полно отражать их назначение, то придется постоянно перечитывать их **дано/получить**, и мы не сможем эффективно создавать и модифицировать тексты программ. Посмотрите, например, как будет выглядеть программа «в левый верхний угол стола» (разд. 2), если заменить все имена на однобуквенные и убрать все комментарии:

```
программа А
. дано :
. получить:
. --------
. цикл пока X.не B выполнять
. . X.C
. конец цикла
. утв: X.B
. цикл пока X.не D выполнять
. . X.E
. конец цикла
. утв: X.B и X.D
конец программы
```

Поэтому выбираемое имя должно достаточно точно описывать программу. Назовем, например, программу, решающую подзадачу *Б*,

«из левого верхнего угла стола в левый верхний угол металла».

Формулировка дано/получить. Программа, решающая подзадачу *Б*, должна удовлетворять условиям

1) предусловие $Б \Rightarrow$ **дано**,
2) **получить** $\Rightarrow$ постусловие *Б*.

Поэтому проще всего взять в качестве **дано/получить** пред- и постусловия *Б*, после чего мы придем к задаче реализации программы с известными **дано/получить**:

программа из левого верхнего угла стола
• в левый верхний угол металла
• **дано** : РМ.резак у левого края стола **и**
РМ.резак у верхнего края стола
• | на столе у Резчика есть металл
• **получить**: РМ.резак над металлом | в левом верхнем углу листа
• --------

Обычно, однако, одна и та же программа *П* вызывается в разных местах для решения, вообще говоря, разных подзадач *Ф*, *Ф′* и *Ф″* и т. д. Условия корректности должны выполняться для всех вызовов. Даже если программа *П* используется в одном месте, то при изменениях в программе может понадобиться использовать ее где-то в другом. Об этом тоже лучше позаботиться заранее. Наконец, попадем ли мы при решении подзадачи *Ф* в ситуацию, когда подходящая программа уже есть, или в ситуацию, когда ее надо создать, решающим образом зависит от **дано/получить** уже существующих программ.

Тем самым желательно, чтобы одна и та же программа *П* была применима для решения возможно большего числа подзадач — как и тех, которые уже возникли, так и тех, которые гипотетически могут возникнуть потом. Для этого нужно, чтобы **дано** *П* следовало из как можно большего числа различных предусловий, а из **получить** *П* следовало как можно большее число различных постусловий. Или, говоря математическим языком, *надо максимально ослабить дано и максимально усилить получить.*

Замечание. Для того чтобы человеку было удобно работать с программой, ее **дано/получить** должны быть возможно более общими, естественными и легко запоминающимися. Как правило, ослабление упрощает, а усиление усложняет условие. Кроме того, на практике одна и та же программа обычно используется для решения подзадач с совпадающими (или почти совпадающими) постусловиями, но с довольно различными предусловиями. Поэтому усиление **получить** разумно лишь до тех пор, пока это не приводит к появлению и нагромождению деталей.

Р е з ю м е. Выделив решение подзадачи Φ в отдельную программу, придумав ей имя и сформулировав черновой вариант ее **дано/получить** (**дано** = предусловие Φ, **получить** = постусловие Φ), надо попробовать максимально ослабить **дано** и, быть может, усилить **получить** так, чтобы программа была применима при возможно более общих, естественных и легко запоминающихся условиях. Образно говоря, программа должна давать максимум отдачи при минимуме затрат.

П р и м е р. Пусть возникла следующая подзадача Φ:

предусловие: | Путник над северо-западным углом препятствия
| лицом на юг, препятствие не прилегает к стенам
| квадранта
постусловие: | Путник под южным краем препятствия

Попробуем придумать имя программе, решающей эту подзадачу, скажем «от северного края препятствия к южному». Это имя, однако, никак не отражает того факта, что вначале Путник находится в определенной точке у северного края препятствия, к тому же еще и с определенной ориентацией. Попробуем не включать эту информацию в **дано** (т. е. ослабить его):

программа от северного края препятствия к южному
. **дано** : | Путник над северным краем препятствия,
. | препятствие не прилегает к стенам квадранта
. **получить:** | Путник под южным краем препятствия
. --------

Такое решение упрощает **дано** и не очень усложняет реализацию. Заметим, что из имени этой программы не следует, что она правильно работает только для препятствий, не прилегающих к стенам квадранта. Это может ввести в заблуждение, поэтому надо либо еще более ослабить **дано**, либо изменить имя программы. Отбрасывание требования о расположении препятствия внутри квадранта, конечно, упростит **дано**, но сделает невозможной реализацию: например, если препятствие прилегает к нижнему краю квадранта, то встать под ним невозможно. Поэтому изменим имя программы так, чтобы оно не вводило в заблуждение: например, назовем ее «от северного края внутреннего препятствия к южному»:

программа от северного края внутреннего препятствия к южному
. **дано** : | Путник над северным краем препятствия,
. | препятствие не прилегает к стенам квадранта
. **получить:** | Путник под южным краем препятствия
. --------

Естественно, что ослаблять **дано**, усиливать **получить** и придумывать имя программы можно по-разному. В той же самой подзадаче

Ф, например, можно было оставить в **дано** только информацию о расположении Путника относительно препятствия:

```
программа на противоположный край внутреннего препятствия
. дано     : | Путник стоит лицом к препятствию,
.            | препятствие не прилегает к стенам квадранта
. получить: | Путник стоит у противоположной стороны
.            | препятствия
. --------
```

Выбор того или иного варианта ослабления **дано** и усиления **получить** определяется тем, какую из получающихся программ окажется проще применить для решения возможно большего количества подзадач, т. е. зависит от общей стратегии решения исходной задачи.

Конец примера.

Оставшаяся часть этого раздела посвящена случаям, когда для решения подзадачи нужно многократно повторять однотипные последовательности действий.

Рекурсия. *Рекурсией* называется ситуация, когда программа вызывает сама себя либо непосредственно, либо через другие программы. По сути, это означает, что мы некоторую подзадачу свели к точно такой же, но с другими исходными данными. Такое сведение является одной из базисных схем обработки информации и также называется *рекурсией.*

Рассмотрим пример. Пусть надо написать следующую программу:

```
программа расстояние до препятствия
. дано     : | Путник где-то, лицом на восток
. получить: | Путник там же, лицом на запад, в Счетчике —
.            | расстояние от Путника до ближайшего препятствия
.            | на восток (т. е. число свободных клеток между
.            | Путником и препятствием восточнее него)
. --------
```

Можно рассуждать следующим образом: если восточнее Путника находится стена или препятствие, то искомое расстояние равно нулю, и для того, чтобы было выполнено **получить**, достаточно написать

```
Счетчик.установить в нуль
Путник.повернуться на запад
```

Если же клетка восточнее Путника свободна, то можно сделать шаг на восток, после чего искомое в задаче расстояние можно вы-

числить как расстояние от нового положения Путника, увеличенное на 1. Таким образом, в этом случае мы можем написать:

Путник. сделать шаг
  утв: | Путник на шаг восточнее, лицом на восток
фрагмент А
  утв: | Путник там же, лицом на запад, в Счетчике —
  | расстояние до препятствия
Путник. сделать шаг
  утв: | Путник в исходном положении, лицом на запад
Счетчик. увеличить на единицу
  утв: | в Счетчике — расстояние от исходного положения
  | до препятствия на востоке

Обратите внимание на подзадачу *А*. Мы можем решать эту подзадачу с помощью вызова другой программы — с уже сформулированными дано/получить. Такая программа есть, и это программа «расстояние до препятствия». Действительно, так как предусловие $A \Rightarrow$ **дано** этой программы, а **получить** $\Rightarrow$ постусловие A, то вызов программы «расстояние до препятствия» для решения подзадачи *А* является корректным:

программа расстояние до препятствия
. **дано** : | Путник где-то, лицом на восток
. **получить:** | Путник там же, лицом на запад, в Счетчике —
. | расстояние от Путника до ближайшего препятствия
. | на восток (т. е. число свободных клеток между
. | Путником и препятствием восточнее него)
. ---------
. **если** Путник. впереди занято
. . **то**
. . Счетчик. установить в нуль
. . Путник. повернуться на запад
. . **иначе**
. . Путник. сделать шаг
. . **утв:** | Путник на шаг восточнее, лицом на восток
. . расстояние до препятствия
. . **утв:** | Путник там же, лицом на запад, в Счетчике —
. . | расстояние до препятствия
. . Путник. сделать шаг
. . **утв:** | Путник в исходном положении, лицом на запад
. . Счетчик. увеличить на единицу
. . **утв:** | в Счетчике — расстояние от исходного
. . | положения до препятствия на востоке
. **конец если**
конец программы

Эта программа корректна. Но является ли она правильной? Как мы знаем, корректная программа *П* является правильной, если все вызываемые из нее программы правильны. Но в данном случае среди вызываемых есть и сама программа *П*. Как же узнать, правильна она или нет?

Рассмотрим работу Универсального Выполнителя при вызове программы *В* из программы *А*. Если каждую программу представлять себе записанной на отдельном листке бумаги, то работа УВ состоит в том, что он помечает на листке *А* точку (место) приостановки, откладывает листок *А* в сторону и начинает выполнять программу *В*. Если в *В* встречается вызов программы *С*, то УВ помечает точку приостановки на листке *В*, кладет его сверху на листок *А* и начинает выполнять программу *С* и т. д. При завершении выполнения очередной программы, например программы *С*, УВ берет верхний листок из стопки отложенных и продолжает выполнение записанной на нем программы (в данном случае программы *В*), начиная с помеченной точки. При завершении выполнения программы *В* из стопки берется листок с программой *А* и т. д. до тех пор, пока вся стопка не исчерпается. Таким образом, стопка листков с приостановленными программами образует стек.

В случае рекурсивного вызова программой *Х* себя УВ также помечает точку приостановки и кладет листок с программой *Х* в стек приостановленных программ. После этого он берет другой точно такой же листок (мы будем говорить — другой *экземпляр* программы *Х*) и начинает выполнять то, что на нем написано, т. е. программу *Х* с начала. При следующем рекурсивном вызове УВ положит второй экземпляр *Х* в стек и начнет выполнять третий и т. д. до тех пор, пока при завершении выполнения очередной программы не окажется, что стек уже пуст.

Объяснение работы УВ в терминах стека листков с приостановленными программами, впрочем, не совсем точно. Реально нет необходимости помещать в стек программы целиком или переписывать их на новые листки при рекурсивных вызовах — достаточно запоминать в стеке лишь имена приостановленных программ и точки их приостановки.

Посмотрим теперь, что происходит при выполнении программы «расстояние до препятствия», если в начальный момент Путник находится за две клетки до препятствия (рис. 4.1).

Каждый раз программа вызывается с другими исходными данными (другим положением Путника). Вопрос о правильности работы программы при конкретных исходных данных сводится к вопросу о правильности работы программы при других исходных данных и т. д. вплоть до правильности работы программы в ситуации, когда препятствие находится непосредственно перед Путником.

В последнем случае вопрос о правильности решается прямо, и программа в целом оказывается правильной.

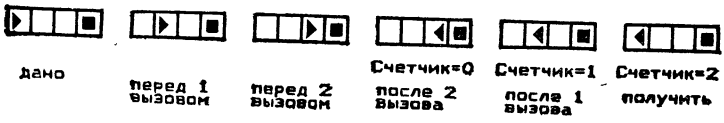


Рис. 4.1

Таким образом, единственная опасность рекурсивной реализации состоит в том, что программа потенциально может вызывать себя бесконечно. Чтобы этого не произошло, нужно придумать какое-то рассуждение, гарантирующее, что выполнение программы рано или поздно закончится. В данном случае каждый новый вызов происходит ближе к препятствию, чем предыдущий. Так как расстояние до препятствия конечно, то рано или поздно выполнение программы завершится.

Обратите внимание, что мы запрограммировали некоторый повторяющийся процесс, т. е. рекурсия является еще одним (наряду с циклами) средством программирования повторяющихся процессов.

Конструкции циклов. Программирование повторяющихся процессов, когда короткая программа описывает длинную последовательность действий, требует особого и пристального внимания. При возникновении такой задачи разумно попытаться свести ее к какой-нибудь из *базисных схем обработки информации.* Одна из таких схем — рекурсия — была описана выше. Сейчас мы приведем еще две: итерацию и схему проектирования цикла с помощью инварианта. Другие схемы — схема вычисления инвариантной функции и схема индуктивного вычисления функции на пространстве последовательностей — будут изложены в разд. 6 и 8.

Итерация. Изложение метода итераций начнем с математической модели. Пусть

M — некоторое множество,

$P: M \to \{\text{да, нет}\}$ — предикат на M, т. е. отображение M в множество из двух элементов: да и нет,

$M \setminus P = \{x \in M : P(x) = \text{нет}\}$ — множество тех элементов $x \in M$, для которых $P(x) = \text{нет}$.

Требуется найти элемент $x \in M$ такой, что $P(x) = \text{да}$.

Метод итераций (дословно — повторений) состоит в том, что строится некоторое преобразование (отображение) $T: M \setminus P \to M$ и это преобразование последовательно применяется, начиная с какого-то $x0 \in M$:

$$x1 = T(x0),\ x2 = T(x1),\ \ldots\ xn = T(xn-1),\ \ldots$$

до тех пор, пока мы впервые не получим xi такое, что P(xi) = да (естественно, должна быть уверенность, что рано или поздно такое xi найдется).

Графически метод представлен на рис. 4.2.

С программистской точки зрения, если в цикле меняются K объектов, то M = m1*m2* ... *mK, где mi — множество состояний i-го объекта, а знак * означает прямое (декартово) произведение множеств. Соответственно x0, x1, ..., xn, ... — это какие-то конкретные наборы состояний объектов, а P — описание состояния, которого мы хотим достичь (наша цель).

Конечно в такой формулировке в терминах метода итераций можно описать *любой* цикл: всегда есть какие-то объекты (x),

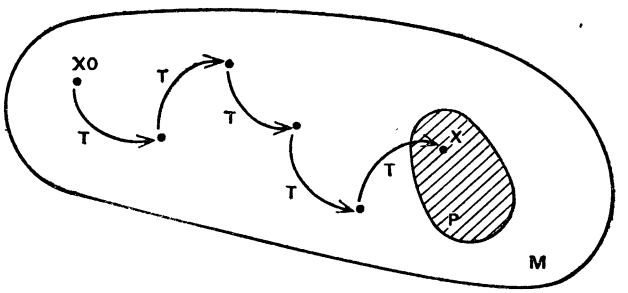


Рис. 4.2. Метод итераций

которые как-то меняются в результате выполнения тела цикла (T), и всегда есть цель, которой мы такими изменениями хотим достичь (P). Поэтому в чистом виде метод итераций столь же бесполезен, сколь и общ, — ведь задача состоит не в том, чтобы описать цикл в математических терминах, а в том, чтобы, используя математическую модель, строить программы. Поэтому на практике в чистом виде метод итераций применяется лишь в простейших случаях, когда в цикле меняется лишь один объект.

Обратите внимание, что для решения одной и той же задачи можно совершенно по-разному выбирать M, P и T еще до какого-либо написания программы.

Пример 1. Пусть требуется написать фрагмент программы, эквивалентный «Путник.шагать до упора», не используя само это предписание. В этом случае M — это множество свободных клеток впереди Путника, x0 — начальное положение, P(x) — условие «впереди занято» (наша цель), T — преобразование «сделать шаг», и соответствующий фрагмент программы выглядит так:

цикл пока Путник. впереди свободно
выполнять

Путник. сделать шаг
конец цикла

Пример 2. Пусть в прямоугольном квадранте произвольным образом расположены препятствия любой формы и есть хотя бы одна свободная клетка. Свободную клетку будем называть *угловой*, если в ней можно расположить Путника так, что впереди и справа от него будут две занятые клетки. Требуется написать программу, которая переводит Путника из произвольной свободной клетки в угловую.

Приведем два решения этой задачи методом итераций с разными M, P и T.

Решение 1. В качестве M возьмем множество всех свободных клеток квадранта. Для клетки $m \in M$ положим $P(m) =$ да тогда и только тогда, когда клетки севернее и восточнее m заняты. Таким образом, из $P(m)$ следует, что клетка m является угловой. Преобразование $T: M \setminus P \to M$ определим так:

$$T(m) = \begin{cases} \text{клетка севернее } m, \text{ если она свободна,} \\ \text{клетка восточнее } m \text{ в противном случае.} \end{cases}$$

Если теперь отождествить клетку $m \in M$ с положением Путника в этой клетке с ориентацией на север, то предикат P можно записать в виде «Путник.впереди занято и Путник.справа занято»; получение какой-нибудь точки $m0 \in M$ — в виде «Путник.повернуться на север»; а преобразование T — в виде:

если Путник. впереди свободно **то** Путник. сделать шаг
. **иначе**
. Путник. повернуться на восток
. Путник. сделать шаг
. Путник. повернуться на север
конец если

Программа в целом, таким образом, запишется так:

программа поиск угла
. **дано** : | Путник где-то в квадранте
. **получить**: | Путник в одной из угловых клеток
. --------
. Путник. повернуться на север
. **цикл пока** Путник. впереди свободно **или**
. Путник. справа свободно
. . **выполнять**
. . **если** Путник. впереди свободно **то** Путник. сделать шаг
. . . **иначе**
. . . Путник. повернуться на восток

. . . Путник. сделать шаг
. . . Путник. повернуться на север
. . **конец если**
. **конец цикла**
конец программы

Завершение этой программы за конечное число шагов гарантируется тем, что сумма расстояний от Путника до северной и восточной стен квадранта уменьшается на 1 при каждом выполнении тела цикла.

Решение 2. В качестве M возьмем множество таких свободных клеток, севернее или восточнее которых есть занятая клетка. Предикат P (нашу цель) оставим таким же, как в предыдущей задаче. Клетку $m \in M$ отождествим с положением Путника в этой клетке (независимо от ориентации). Преобразование T зададим так:

Путник.повернуться на север
Путник.шагать до упора
Путник.повернуться на восток
Путник.шагать до упора

Получение $m0 \in M$ можно записать так:

Путник.повернуться на север
Путник.шагать до упора

Написание самой программы и доказательство ее завершаемости за конечное число шагов оставляется читателю.

Схема проектирования цикла с помощью инварианта. У метода итераций есть подсхема с меньшей областью применения и соответственно более полезная на практике — это схема проектирования цикла с помощью инварианта. Основной идеей этой схемы является выражение взаимосвязи между меняющимися в теле цикла объектами в виде неизменного условия — *инварианта*, — которое выполнено для начальных, промежуточных и конечных состояний объектов. Польза инварианта состоит в том, что такое статическое описание связей между изменяющимися объектами легко понимаемо и позволяет, зная, как меняются одни объекты, *выводить*, как должны изменяться другие.

Более формально, пусть по-прежнему

M — некоторое множество,

P: $M \to$ {да, нет} — предикат на M, т. е. отображение M в множество из двух элементов: да и нет,

$M \setminus P = \{x \in M : P(x) = \text{нет}\}$ — множество тех элементов $x \in M$, для которых $P(x) =$ нет,

T: $M \setminus P \to M$ — некоторое преобразование,

И пусть существуют отображения (предикаты)

I: M → {да, нет} (инвариант),

Q: M → {да, нет} (условие окончания)

такие, что

1) I(x) и Q(x) ⇒ P(x),

2) I(x0) = да (инвариант выполнен для начального состояния объектов),

3) I(x) = да ⇒ I(T(x)) = да (инвариант сохраняется при преобразовании T).

Тогда в методе итераций в качестве условия окончания вместо P(x) можно взять Q(x).

Графически эта схема представлена на рис. 4.3.

На практике решение задачи с заданными пост- и предусловиями по схеме проектирования цикла с помощью инварианта состоит

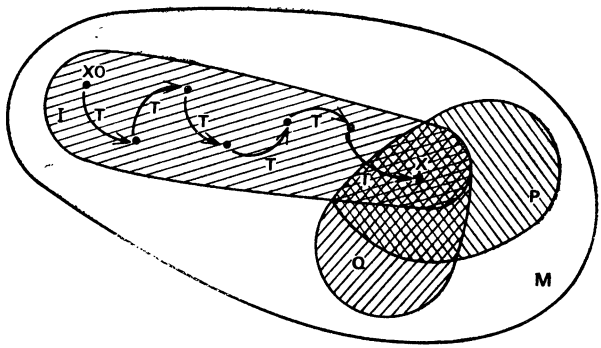


Рис. 4.3. Схема проектирования цикла с помощью инварианта

в следующем. Прежде всего надо придумать общую стратегию решения задачи, т. е. неформально решить, что будет происходить в цикле, а также описать объект или объекты, которые будут меняться при его выполнении. Далее надо:

1) сформулировать условие окончания Q — условие, при выполнении которого цикл должен закончиться;

2) попытаться описать взаимосвязи между изменяющимися в ходе выполнения цикла объектами в виде неизменного условия — инварианта I.

Если это удалось, то решение задачи можно записать в виде:

утв: предусловие
фрагмент A
цикл
. инв: I

. **пока не** Q
. **выполнять**
. *фрагмент Б*
конец цикла
фрагмент В
утв: постусловие

После чего останется:

А) обеспечить выполнение инварианта перед началом цикла;

Б) придумать тело цикла, сохраняющее инвариант, и убедиться, что цикл рано или поздно закончится;

В) обеспечить выполнение постусловия после окончания цикла.

(Запись **инв** между **цикл** и **пока** является синонимом **утв** и означает, что утверждение I будет проверяться каждый раз перед



Рис. 4.4

проверкой условия в **пока**, т. е. что I выполнено перед и после каждого шага цикла, включая первый и последний.)

Пример. Пусть известно (это предусловие), что Путник находится у западного края квадранта, лицом на восток, и что горизонтальный ряд клеток, в котором находится Путник, свободен от препятствий (рис. 4.4).

Над Путником находится полоса препятствий, т. е. несколько прямоугольных препятствий одинаковой высоты, но, быть может,



Рис. 4.5

разной ширины. Препятствия могут прилегать, а могут и не прилегать к западной и восточной стенам квадранта. В полосе может и не быть ни одного препятствия или, наоборот, не быть ни одного прохода.

Надо (это постусловие), чтобы Путник оказался у восточной стены, а в Счетчике получилось число препятствий в полосе (в ситуации, изображенной на рис. 4.5, в Счетчике должно быть число 5).

Остановитесь на минутку и, прежде чем читать дальше, попробуйте решить эту задачу самостоятельно — сначала как-нибудь, а

ватем применив схему проектирования цикла с помощью инварианта. Достаточно будет, если Вы сделаете это в уме, прикинув, какое у Вас будет условие окончания цикла, каково будет начальное, конечное и промежуточные положения Путника и т. д. Напомним, что пользоваться можно только управляющими конструкциями разд. 2 и предписаниями Путника и Счетчика.

Применим схему проектирования цикла с помощью инварианта. Первое, что надо сделать, это сформулировать общую стратегию и описать меняющиеся объекты. Общая стратегия очевидна — идти на восток и «по дороге» считать число пройденных препятствий. Оставим стратегию пока в таком виде, никак ее не уточняя, и опишем меняющиеся объекты. Таких объектов два: положение Путника и значение Счетчика.

Второе — надо попытаться описать цикл в терминах условия окончания Q и инварианта I. С условием окончания просто — остановиться надо у восточной стены квадранта, т. е. как только клетка впереди Путника окажется занятой. Обратите внимание, что мы не только сформулировали условие окончания, но и *формально выразили его* через предписания Путника. Если бы это оказалось невозможным, то от этого условия окончания пришлось бы отказаться и придумывать другое.

Итак, дело за инвариантом. Как связаны между собой положение Путника и значение Счетчика? Стратегия подсказывает: в Счетчике должно быть число препятствий в пройденной части полосы. Но включать или нет в пройденную часть клетку над Путником? Для решения этого вопроса надо посмотреть на конечное состояние (рис. 4.5). В этот момент в Счетчике должно оказаться общее число препятствий в полосе, и потому клетку над Путником в пройденную часть включать надо.

Мы сформулировали I и Q. Легко видеть, что в данном случае постусловие = (I и Q), и решение задачи можно записать в виде:

утв: предусловие
фрагмент А
цикл
. **инв:** | в Счетчике число препятствий в пройденной
. | части полосы
. **пока** Путник. впереди свободно
. **выполнять**
. *фрагмент Б*
конец цикла
утв: | в Счетчике число препятствий в пройденной
| части полосы и вся полоса пройдена

Мы, впрочем, упустили одну очень важную вещь — не описали начальное и промежуточные положения Путника при выполнении цикла. Одно из решений могло бы быть таким: внутри цикла будем двигаться от препятствия к препятствию так, как изображено на рис. 4.6. Тогда начальное положение — Путник под первой клеткой первого препятствия, промежуточные — под первой клеткой очередного препятствия, конечное — под первой клеткой конечного препятствия. Однако как при этом отличить конечное положение (т. е. как формально выписать условие окончания), если последнее препятствие не прилегает к восточной стене или прилегает, но имеет не



Рис. 4.6

единичную ширину? И что значат эти состояния, если препятствий в полосе вообще нет?

Система предписаний Путника такова, что условием окончания цикла может быть только одно — «впереди занято», т. е. Путник в клетке у восточной стены. Препятствие над ним в этот момент может быть, а может и не быть — конечное положение вообще никак не связано с препятствиями. Значит, решение двигаться от препятствия к препятствию является неверным.

Попробуем двигаться просто от клетки к клетке. Тогда начальное положение Путника — у западной стены, конечное — у восточной, а промежуточные — последовательно по всем клеткам слева направо (с запада на восток).

Подзадача *А* становится тривиальной — надо поместить в Счетчик число препятствий в пройденной части полосы, т. е.

```
Счетчик. установить в нуль
если Путник. слева занято
то
  Счетчик. увеличить на единицу
конец если
```

Внутри цикла надо переместить Путника на шаг вправо и при этом сохранить инвариант. Посмотрим, когда при таком перемещении число препятствий в пройденной части полосы изменяется. Это происходит в единственном случае — если после шага Путник оказывается под первой (левой) клеткой нового препятствия, т. е. если *до шага* слева было свободно, а *после шага* — занято. Так как наши действия зависят от того, было ли *до шага* слева занято или

свободно, то надо решать эту задачу с помощью конструкции выбора:

```
выбор
. при Путник. слева занято ⇒ фрагмент Б1
. при Путник. слева свободно ⇒ фрагмент Б2
конец выбора
```

Мы получили две новые подзадачи: *Б1* — переместить Путника на шаг вправо и сохранить инвариант при условии, что до шага слева было занято, и *Б2* — то же самое, но при условии, что до шага слева было свободно.

Решаем задачу *Б1*. Если до шага было занято, то число препятствий в пройденной части полосы после шага измениться не может. Таким образом, *Б1* это просто

```
Путник.сделать шаг
```

В задаче *Б2* число препятствий может увеличиться на 1, если после шага Путник окажется под первой клеткой нового препятствия, т. е. если слева окажется занято. Таким образом, задача *Б2* решается так:

```
Путник. сделать шаг
если Путник. слева занято
. то
.  Счетчик. увеличить на единицу
конец если
```

Окончательное решение задачи выглядит так:

```
Счетчик.установить в нуль
если Путник.слева занято
. то
.  Счетчик. увеличить на единицу
конец если
цикл
. инв: | в Счетчике — число препятствий в пройденной
.      | части полосы
. пока Путник. впереди свободно
. выполнять
. выбор
. .    при Путник. слева занято ⇒ Путник. сделать шаг
. .    при Путник. слева свободно ⇒
. .          Путник. сделать шаг
. .          если Путник. слева занято
.            . то
```

```
. .          . Счетчик. увеличить на единицу
. .          конец если
. конец выбора
конец цикла
```

Замечание 1. Выше описано, как примерно должен думать человек при создании программы и какой текст при этом он должен писать. Нужно понимать, что цель при этом преследуется сугубо практическая — научиться писать несложные программы быстро и без ошибок, а более сложные с минимальным количеством ошибок. При применении изложенной выше методики на практике не следует впадать в крайности и доказывать очевидные вещи или разбивать задачу на мельчайшие подзадачи. Если промежуточные утверждения, инварианты циклов и прочее помогают писать программу быстро и без ошибок, их следует применять. Если же они только загромождают текст программы и ничего не проясняют, надо обходиться без них. По мере накопления опыта многие вещи начинают делаться в уме, и это совсем не плохо. Вместе с тем даже в самых простых задачах часто обнаруживается масса «подводных камней». Поэтому при малейших сомнениях в правильности программы следует пытаться взглянуть на нее с более формальных, чем обычно, позиций.

Замечание 2. Выписываемые в программе **дано/получить**, инварианты и утверждения, как правило, являются неполными. Какие-то детали всегда считаются известными. При учете всех деталей утверждения становятся громоздкими, плохо понимаемыми и начинают не помогать, а мешать.

Замечание 3. В конструкциях **дано**, **получить** и **утверждение** записываются как комментарии, предназначенные для человека, так и формальные условия, проверяемые Универсальным Выполнителем при выполнении программы. Включение формальных условий в текст программы сильно снижает вероятность того, что программа с ошибками покажется работающей правильно. Поэтому формальные условия включают в программу часто даже в ущерб краткости и ясности текста.

ЗАДАЧИ И УПРАЖНЕНИЯ

Придумайте название и напишите программу со следующими **дано/получить**:

```
1. дано     : | Путник в северо-западном углу квадранта,
              | в квадранте одно прямоугольное препятствие,
              | это препятствие не прилегает к стенам квадранта
  получить  : | в Счетчике — ширина препятствия
```

2. **дано** :	Путник в северо-западном углу квадранта, в квадранте одно прямоугольное препятствие, это препятствие не прилегает к стенам квадранта
получить :	в Счетчике — периметр препятствия
3. **дано** :	в квадранте одно прямоугольное препятствие
получить :	в Счетчике — число препятствий, прилегающих к стенам квадранта
4. **дано** :	в квадранте два одноклеточных препятствия
получить :	в Счетчике — периметр их объединения
5. **дано** :	в квадранте несколько прямоугольных препятствий, все они прилегают к южной стене квадранта и не прилегают друг к другу и к другим стенам квадранта
получить :	в Счетчике — число препятствий
6. **дано** :	в квадранте несколько прямоугольных препятствий, все они прилегают к южной стене квадранта и не прилегают друг к другу и к другим стенам квадранта
получить :	в Счетчике — сумма ширин всех препятствий
7. **дано** :	в квадранте несколько прямоугольных препятствий, все они прилегают к южной стене квадранта и не прилегают друг к другу и к другим стенам квадранта
получить :	в Счетчике — сумма высот всех препятствий
8. **дано** :	в квадранте одно прямоугольное препятствие, не прилегающее к стенам квадранта, Путник над его северным краем
получить :	Путник в той же клетке, в Счетчике — ширина препятствия
9. **дано** :	в квадранте одно прямоугольное препятствие, не прилегающее к стенам квадранта, Путник у западной стороны препятствия лицом на север
получить :	Путник на той же горизонтали у восточной стороны препятствия лицом на юг
10. **дано** :	в квадранте несколько прямоугольных препятствий, не прилегающих друг к другу и к стенам квадранта
получить :	в Счетчике — число препятствий, пересекающих горизонтальный отрезок из клеток от Путника до восточной стены квадранта

Будем называть препятствие *шарообразным*, если при обходе его по часовой стрелке начиная с положения над северным краем препятствия сначала надо идти только на восток и на юг, потом

только на юг и на запад, потом только на запад и на север и наконец, только на север и на восток.

11. дано : | в квадранте одно шарообразное препятствие, не прилегающее к стенам квадранта, Путник у верхнего края препятствия лицом на восток
получить : | Путник где угодно, в Счетчике число северо-восточных углов препятствия, т. е. таких углов, клетки севернее и восточнее которых свободны

12. дано : | в квадранте одно шарообразное препятствие, не прилегающее к стенам квадранта, Путник у западного края препятствия лицом на восток
получить : | Путник на той же горизонтали с восточной стороны препятствия лицом на юг

5. Процесс разработки программ. Один пример

Формулировка задачи. Пусть известно, что квадрант, в котором действует Путник, состоит из нечетного числа горизонтальных полос различной ненулевой высоты. Первая (верхняя) и последняя (нижняя) полосы свободны от препятствий. В каждой полосе с нечётным

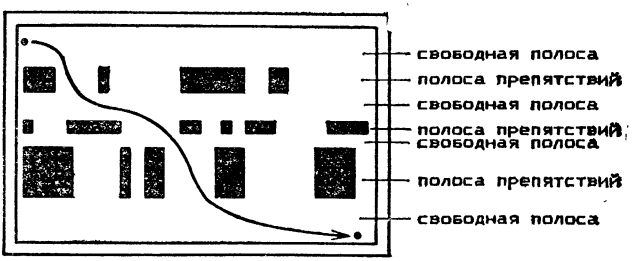


Рис. 5.1

номером имеются некоторое ненулевое число прямоугольных препятствий, высота которых равна высоте полосы, и хотя бы один проход (рис. 5.1).

Задача Путника состоит в том, чтобы пройти по квадранту из левого верхнего угла в правый нижний и по дороге подсчитать некоторые характеристики квадранта, например площадь всех препятствий, число полос с препятствиями, периметр проходов, у которых высота больше ширины, и т. п. (проходами называются прямоугольные промежутки между препятствиями или между препятствием и стеной в полосе препятствий).

Для проведения необходимых вычислений и запоминания информации в наше распоряжение предоставлен базовый исполнитель

«Бортовой вычислитель» (БВ), предназначенный для подсчета характеристик квадранта. Его система предписаний такова:

1. начать работу
2. число установить в нуль
3. число увеличить на единицу
4. число показать значение
5. общее число установить в нуль
6. общее число увеличить на единицу
7. общее число показать значение
8. общее число увеличить на число

Предписания 9—36 аналогичны предписаниям 2—8 и получаются заменой пары (число, общее число) на соответственно (ширина, общая ширина), (высота, общая высота), (периметр, общий периметр), (площадь, общая площадь). Так, например, есть предписание

36. общая площадь увеличить на площадь

Кроме того, Бортовой вычислитель умеет выполнять перекрестные операции:

37. периметр увеличить на ширину
38. периметр увеличить на высоту
39. площадь увеличить на произведение ширины на высоту
40. ширина меньше высоты : да/нет
41. ширина равна высоте : да/нет
42. ширина больше высоты : да/нет

Наконец, имеются предписания, позволяющие запомнить некоторую информацию, а потом узнать, какая информация была запомнена:

43. запомнить что край
44. запомнить что ряд свободен
45. запомнить что есть препятствия
46. край : да/нет
47. ряд свободен : да/нет
48. есть препятствия : да/нет
49. запомнить что флаг поднят
50. запомнить что флаг опущен
51. флаг поднят : да/нет
52. флаг опущен : да/нет
53. кончить работу

Бортовой вычислитель можно представлять в виде специализированного калькулятора — коробочки с кнопками, соответствующими перечисленным выше предписаниям. Внутри Бортового вычислителя имеется 12 объектов (его память). Это 10 встроенных

счетчиков (число, общее число, ширина, общая ширина, высота, общая высота, периметр, общий периметр, площадь, общая площадь), а также два объекта для запоминания информации — один на три положения (край/ряд свободен/есть препятствия), а другой на два (флаг поднят/флаг опущен). Состояния всех этих объектов и, в частности, ответы на вопросы типа «край» или «ряд свободен» зависят лишь от «нажатий на кнопки» Бортового вычислителя и физически никак не связаны с положением Путника в квадранте.

Кроме кнопок на передней панели Бортового вычислителя имеется табло, на котором по предписаниям «...показать значение» изображается состояние того или иного счетчика.

Итак, пусть перед нами стоит задача написать программу:

```
программа подсчет площади всех препятствий
. дано      : | Путник в северо-западном (левом верхнем) углу
.             | квадранта. Препятствия в квадранте расположены
.             | горизонтальными полосами и не прилегают к
.             | северной и южной стенам квадранта
. получить: | Путник в юго-восточном углу, лицом на восток,
.             | на табло Бортового вычислителя — суммарная
.             | площадь всех препятствий в квадранте
. --------
```

Решение. В соответствии с технологией «сверху вниз» следует выделить наиболее крупные образования — структуры, которые были бы меньше, чем квадрант целиком, и придумать промежуточного исполнителя, который с этими образованиями будет работать. Легко видеть, что такими наиболее крупными образованиями являются полосы. Поэтому примем решение, что у нас будет специальный исполнитель «Обработчик полос» (ОП), выполняющий всю работу с полосами целиком.

Мы определили уровень подзадач, на которые будем разбивать исходную задачу. Поскольку число полос неизвестно, то решение будет содержать какой-то повторяющийся процесс. А так как есть по крайней мере два объекта, которые при этом меняются (положение Путника и счетчик Бортового вычислителя, в котором мы будем считать общую площадь всех препятствий), то попробуем применить схему проектирования цикла с помощью инвариаита.

Прежде всего надо в терминах полос сформулировать общую стратегию решения задачи. Будем спускаться вдоль левого (западного) края квадранта и по дороге считать площадь препятствий в пройденных полосах (рис. 5.2).

Теперь надо описать начальное, промежуточные и конечное состояния изменяющихся в цикле объектов. Пусть конечное положение Путника — в левом нижнем углу, промежуточные — в левом

нижнем углу очередной свободной полосы, а начальное — в левом нижнем углу первой (верхней) свободной полосы. В счетчике «общая площадь» в этих положениях должна быть суммарная площадь препятствий в пройденной части квадранта. Таким образом, инвариант цикла можно записать так: «Путник в юго-западном углу свободной полосы, в счетчике «общая площадь» содержится площадь препятствий в пройденной части квадранта».

Условие окончания цикла — Путник в левом нижнем (юго-западном) углу квадранта. Однако само это положение отличить от промежуточных можно, лишь проанализировав весь ряд под Путником. Если прохода нет, то это — край (стена) квадранта, если

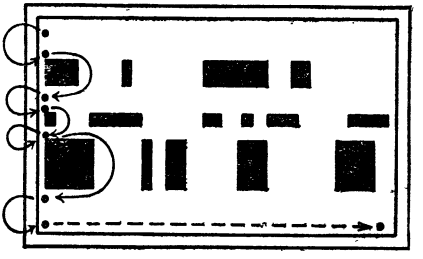


Рис. 5.2

же проход есть, то это — начало полосы препятствий. Назовем эти действия разведкой ряда южнее Путника и будем запоминать результат разведки в объекте «край»/«ряд свободен»/«есть препятствия» Бортового вычислителя. Например, если Путник находится у южной стены квадранта, то результат разведки — установка этого объекта в состояние «край».

Таким образом, искомую программу можно записать в виде

```
программа подсчет площади всех препятствий
. дано      : | Путник в северо-западном (левом верхнем) углу
.             | квадранта. Препятствия в квадранте расположены
.             | горизонтальными полосами и не прилегают к
.             | северной и южной стенам квадранта
. получить: | Путник в юго-восточном углу, лицом на восток,
.             | на табло Бортового вычислителя — суммарная
.             | площадь всех препятствий в квадранте
. --------
. БВ. общая площадь установить в нуль
. ОП. проход свободной полосы с разведкой следующего ряда
. цикл
. . инв: | Путник в юго-западном углу свободной полосы,
. .      | разведан ряд южнее Путника, в счетчике "общая
. .      | площадь" —
```

```
. .        | площадь препятствий в пройденной части квадранта
. . пока не БВ. край
. . выполнять
. .  ОП. проход полосы препятствий с подсчетом площади
. .  БВ. общая площадь увеличить на площадь
. .  ОП. проход свободной полосы с разведкой следующего ряда
. конец цикла
.  утв: | общая площадь = площадь препятствий в пройденной
.       | части квадранта. А так как пройден весь квадрант,
.       | то общая площадь = площади препятствий во всем
.       | квадранте
. БВ. общая площадь показать значение
. Путник. повернуться на восток
. Путник. шагать до упора
конец программы
```

Итак, мы совершили первый шаг декомпозиции и получили реализацию программы «подсчет площади всех препятствий» на базе Обработчика полос, Путника и Бортового вычислителя. Обработчик



Рис. 5.3

полос не является базовым, и, следовательно, мы должны сделать следующий шаг декомпозиции — реализовать этого исполнителя.

```
исполнитель Обработчик полос (ОП)
. СП:
.  1. проход свободной полосы с разведкой следующего ряда
.  2. проход полосы препятствий с подсчетом площади
```

Начнем с предписания «проход свободной полосы с разведкой следующего ряда»:

```
программа проход свободной полосы с разведкой следующего ряда
. дано    : | Путник в северо-западном углу свободной полосы
. получить: | Путник в юго-западном углу той же свободной
.           | полосы. Разведан ряд южнее Путника
. --------
```

Поскольку высота свободной полосы неизвестна, то при реализации этой программы будет какой-то цикл. Опишем конечное, начальное и промежуточные положения Путника (рис. 5.3). Таким образом, общая стратегия состоит в том, чтобы спускаться вдоль западной стены квадранта до тех пор, пока не дойдем до полосы

препятствий или до южного края квадранта. Для того чтобы отличить конечное положение от промежуточных, нужно разведать ряд южнее Путника. Поскольку разведка ряда южнее Путника достаточно сложна, выделим ее в отдельную программу:

```
программа проход свободной полосы с разведкой следующего ряда
. дано      : | Путник в северо-западном углу свободной полосы
. получить: | Путник в юго-западном углу той же свободной
.           | полосы. Разведан ряд южнее Путника
. --------
. разведка ряда южнее Путника
. цикл
. . инв: | Путник у западного края в свободном ряду,
. .      | ряд южнее Путника разведан
. . пока БВ. ряд свободен
. . выполнять
. . Путник. повернуться на юг
. . Путник. сделать шаг
. . разведка ряда южнее Путника
. конец цикла
.   утв: | Путник у западного края в свободном ряду,
.        | ряд южнее Путника разведан и не свободен
конец программы
```

Реализуем теперь программу «разведка ряда южнее Путника», используя, как и выше, схему проектирования цикла с помощью инварианта:

```
программа разведка ряда южнее Путника
. дано      : | Путник у западного края в свободном ряду
. получить: | Путник там же, ряд южнее Путника разведан,
.           | результат разведки запомнен в Бортовом вычислителе
. --------
. Путник. повернуться на восток
. выбор
. .  при Путник.справа занято ⇒ БВ.запомнить что край
. .  при Путник.справа свободно ⇒
. .                          БВ.запомнить что ряд свободен
. конец выбора
. цикл
. . инв: | разведана часть ряда южнее Путника от западной
. .      | стены и до клетки, севернее которой стоит Путник,
. .      | включительно. Результат разведки запомнен в БВ
. . пока Путник.впереди свободно и не БВ.есть препятствия
. . выполнять
. . Путник.сделать шаг
```

```
. . если (БВ.край и Путник.справа свободно) или
. . .      (БВ.ряд свободен и Путник.справа занято)
. . . то
. . .  БВ.запомнить что есть препятствия
. . конец если
. конец цикла
. Путник.повернуться на запад
. Путник.шагать до упора
конец программы
```

Осталось реализовать предписание «проход полосы препятствий с подсчетом площади». Будем делать это в два этапа — сначала пройдем полосу препятствий и подсчитаем ее высоту, а затем пройдем под полосой препятствий и, уже зная высоту препятствий, подсчитаем их площадь:

```
программа проход полосы препятствий с подсчетом площади
. дано     : | Путник у западного края в ряду над полосой
.            | препятствий
. получить: | Путник у западного края в ряду под полосой
.           | препятствий, в счетчике "площадь" Бортового
.           | вычислителя — площадь препятствий в
.           | пройденной полосе
. --------
. проход полосы препятствий с подсчетом ее высоты
.   утв: | Путник у западного края под полосой препятствий,
.        | в счетчике "высота" — высота полосы
```

Вторую задачу — проход под полосой препятствий и подсчет площади — будем решать так: сначала найдем суммарную ширину всех препятствий в полосе (используя схему проектирования цикла с помощью инварианта), а затем установим площадь в произведение ширины на высоту. Собственно суммарную ширину будем считать, шагая по клеточке с запада на восток:

```
. Путник.повернуться на восток
. БВ. ширина установить в нуль
. если Путник.слева занято
. . то
. .  БВ. ширина увеличить на единицу
. конец если
. цикл
. . инв: | в счетчике "ширина" — суммарная ширина препятствий в
. .      | пройденной части полосы (включая клетку над Путником)
. . пока Путник.впереди свободно
. . выполнять
```

```
. .  Путник.сделать шаг
. .  если Путник.слева занято то
. .  .  БВ.ширина увеличить на единицу
. .  конец если
. конец цикла
.  утв: | Путник у восточного края под полосой препятствий,
.       | в счетчике "ширина" — суммарная ширина всех
.       | препятствий в полосе
```

Осталось вычислить площадь препятствий и вернуться к западному краю:

```
. БВ.площадь установить в нуль
. БВ. площадь увеличить на произведение ширины на высоту
. Путник.повернуться на запад
. Путник.шагать до упора
конец программы
```

Итак, мы реализовали предписание «проход полосы препятствий с подсчетом площади», использовав при этом вызов программы «проход полосы препятствий с подсчетом ее высоты». Теперь надо сделать следующий шаг и реализовать эту программу:

```
программа проход полосы препятствий с подсчетом ее высоты
. дано     : | Путник у западного края над полосой препятствий
. получить: | Путник у западного края под полосой препятствий,
.           | в счетчике "высота" — высота полосы
. ---------
. БВ.высота установить в нуль
```

При проходе полосы препятствий могут быть две существенно различные ситуации — препятствие может прилегать или не прилегать к западной стенке квадранта. В первом случае можно просто обойти препятствие и на спуске подсчитать его высоту



Рис. 5.4

(рис. 5.4). Сам спуск и подсчет высоты естественно программировать, используя схему проектирования цикла с помощью инварианта:

```
. Путник.повернуться на юг
. выбор
.  при Путник.впереди занято => | у западного края препятствие
.       Путник.повернуться на восток
```

```
. .      цикл пока Путник.справа занято выполнять
. .      . Путник.сделать шаг
. .      конец цикла
. .      Путник.повернуться на юг
. .      Путник.сделать шаг
. .      цикл
. .      . инв: | в счетчике "высота" — высота пройденной
. .      .      | части препятствия (клетка правее
. .      .      | Путника пройденной не считается)
. .      . пока Путник.справа занято
. .      . выполнять
. .      . Путник.сделать шаг
. .      . БВ.высота увеличить на единицу
. .      конец цикла
```

Во втором случае нужно сначала найти какое-нибудь препятствие, чтобы вдоль него спуститься и подсчитать высоту (рис. 5.5).



Рис. 5.5

```
. . при Путник.впереди свободно => | у западного края свободно
. .      Путник.сделать шаг
. .      Путник.повернуться на восток
. .      Путник.шагать до упора
. .      Путник.повернуться на юг
. .      цикл
. .      . инв: | в счетчике "высота" — высота пройденной
. .      .      | части препятствия (клетка левее
. .      .      | Путника пройденной не считается)
. .      . пока Путник.слева занято
. .      . выполнять
. .      . Путник.сделать шаг
. .      . БВ.высота увеличить на единицу
. .      конец цикла
. конец выбора
```

В обоих случаях после спуска с подсчетом высоты надо еще вернуться к западной стене квадранта:

```
. Путник.повернуться на запад
. Путник.шагать до упора
конец программы
```

В заключение соберем все тексты воедино, чтобы создать представление о том, как выглядит полное решение задачи:

```
программа подсчет площади всех препятствий
. дано      : | Путник в северо-западном (левом верхнем) углу
.             | квадранта. Препятствия в квадранте расположены
.             | горизонтальными полосами и не прилегают к
.             | северной и южной стенам квадранта
. получить: | Путник в юго-восточном углу, лицом на восток, на
.             | табло Бортового вычислителя — суммарная площадь
.             | всех препятствий в квадранте
. --------
. БВ.общая площадь установить в нуль
. ОП.проход свободной полосы с разведкой следующего ряда
. цикл
. . инв: | Путник в юго-западном углу свободной полосы,
. .      | разведан ряд южнее Путника, в счетчике "общая
. .      | площадь" — площадь препятствий в пройденной
. .      | части квадранта
. . пока не БВ.край
. . выполнять
. .  ОП.проход полосы препятствий с подсчетом площади
. .  БВ.общая площадь увеличить на площадь
. .  ОП.проход свободной полосы с разведкой следующего ряда
. конец цикла
.  утв: | общая площадь = площадь препятствий в пройденной
.       | части квадранта. А так как пройден весь
.       | квадрант, то общая площадь = площади препятствий
.       | во всем квадранте
. БВ.общая площадь показать значение
. Путник.повернуться на восток
. Путник.шагать до упора
конец программы | ==============================

исполнитель Обработчик полос (ОП)
. СП:
. 1. проход свободной полосы с разведкой следующего ряда
. — разведка ряда южнее Путника
. 2. проход полосы препятствий с подсчетом площади
. — проход полосы препятствий с подсчетом ее высоты
.
. используемые исполнители:
. Путник
. Бортовой вычислитель (БВ)
конец описаний | ------------------------------
```

```
программа проход свободной полосы с разведкой следующего ряда
. дано      : |Путник в северо-западном углу свободной полосы
. получить: |Путник в юго-западном углу той же свободной
.           |полосы. Разведан ряд южнее Путника
. --------
. разведка ряда южнее Путника
. цикл
. . инв: |Путник у западного края в свободном ряду,
. .      |ряд южнее Путника разведан
. . пока БВ.ряд свободен
. . выполнять
. .  Путник.повернуться на юг
. .  Путник.сделать шаг
. .  разведка ряда южнее Путника
. конец цикла
.   утв: |Путник у западного края в свободном ряду,
.        |ряд южнее Путника разведан и не свободен
конец программы

программа разведка ряда южнее Путника
. дано      : |Путник у западного края в свободном ряду
. получить: |Путник там же, ряд южнее Путника разведан, резуль-
.           |таты разведки запомнены в Бортовом вычислителе
. --------
.  Путник.повернуться на восток
.  выбор
. . при Путник.справа занято ⇒ БВ.запомнить что край
. . при Путник.справа свободно ⇒ БВ.запомнить что ряд свободен
.  конец выбора
.  цикл
. . инв: |разведана часть ряда южнее Путника от западной
. .      |стены и до клетки, севернее которой стоит Путник,
. .      |включительно. Результат разведки запомнен в БВ
. . пока Путник.впереди свободно и не БВ.есть препятствия
. . выполнять
. .  Путник.сделать шаг
. .  если (БВ.край и Путник.справа свободно) или
. .  .    (БВ.ряд свободен и Путник.справа занято)
. .  . то
. .  . БВ.запомнить что есть препятствия
. .  конец если
.  конец цикла
.  Путник.повернуться на запад
.  Путник.шагать до упора
конец программы
```

```
программа проход полосы препятствий с подсчетом площади
. дано      : | Путник у западного края в ряду над полосой
.             | препятствий
. получить: | Путник у западного края в ряду под полосой
.             | препятствий, в счетчике "площадь" Бортового
.             | вычислителя — площадь препятствий в пройденной
.             | полосе
. --------
. проход полосы препятствий с подсчетом ее высоты
.  утв: | Путник у западного края под полосой препятствий,
.       | в счетчике "высота" — высота полосы
. Путник. повернуться на восток
. БВ. ширина установить в нуль
. если Путник. слева занято
. . то
. .  БВ. ширина увеличить на единицу
. конец если
. цикл
. .  инв: | в счетчике "ширина" — суммарная ширина препятствий
. .       | в пройденной части полосы (включая клетку над
. .       | Путником)
. .  пока Путник. впереди свободно
. .  выполнять
. .  Путник. сделать шаг
. .  если Путник. слева занято то
. .  . БВ. ширина увеличить на единицу
. .  конец если
. конец цикла
.  утв: | Путник у восточного края под полосой препятствий,
.       | в счетчике "ширина" — суммарная ширина всех
.       | препятствий в полосе
. БВ. площадь установить в нуль
. БВ. площадь увеличить на произведение ширины на высоту
. Путник. повернуться на запад
. Путник. шагать до упора
конец программы
```

```
программа проход полосы препятствий с подсчетом ее высоты
. дано      : | Путник у западного края над полосой препятствий
. получить: | Путник у западного края под полосой препятствий,
.             | в счетчике "высота" — высота полосы
. --------
. БВ. высота установить в нуль
. Путник. повернуться на юг
. выбор
```

```
. . при Путник. впереди занято ⇒ | у западного края препятствие
. .       Путник. повернуться на восток
. .       цикл пока Путник. справа занято
. .       . выполнять Путник. сделать шаг
. .       конец цикла
. .       Путник. повернуться на юг
. .       Путник. сделать шаг
. .       цикл
. .       . инв: | в счетчике "высота" — высота пройденной
. .       .      | части препятствия (клетка правее
. .       .      | Путника пройденной не считается)
. .       . пока Путник. справа занято
. .       . выполнять
. .       .  Путник. сделать шаг
. .       .  БВ. высота увеличить на единицу
. .       конец цикла
. . при Путник. впереди свободно ⇒ | у западного края свободно
. .       Путник. сделать шаг
. .       Путник. повернуться на восток
. .       Путник. шагать до упора
. .       Путник. повернуться на юг
. .       цикл
. .       . инв: | в счетчике "высота" — высота пройденной
. .       .      | части препятствия (клетка левее
. .       .      | Путника пройденной не считается)
. .       . пока Путник. слева занято
. .       . выполнять
. .       .  Путник. сделать шаг
. .       .  БВ. высота увеличить на единицу
. .       конец цикла
. конец выбора
. Путник. повернуться на запад
. Путник. шагать до упора
конец программы
конец исполнителя |=========================
```

Напомним, что минусы перед именами программ «разведка ряда южнее Путника» и «проход полосы препятствий с подсчетом ее высоты» в описателе исполнителя «Обработчик полос» означают, что эти программы являются локальными: они не входят в систему предписаний, появились как решения каких-то подзадач и не могут быть использованы вне исполнителя «Обработчик полос».

ЗАДАЧИ И УПРАЖНЕНИЯ

Модифицируя решение задачи «подсчет площади всех препятствий», получите решение следующих задач.

1. Число (периметр) всех препятствий.

2. Число (периметр, площадь) препятствий не у восточного края.

3. Число (периметр, площадь) препятствий, не прилегающих к стенам квадранта.

4. Число (периметр, площадь) препятствий, прилегающих к стенам квадранта.

5. Число (периметр, площадь) квадратных препятствий.

6. Число (периметр, площадь) препятствий, у которых ширина больше высоты.

7. Число (периметр, площадь) препятствий размером в одну клетку (1*1).

8. Число (периметр, площадь) препятствий размером 2*2.

9—16. Решите задачи 1—8, но не для препятствий, а для проходов.

17. Известно, что в квадранте несколько прямоугольных препятствий, которые прилегают к южной стене квадранта и не прилегают к другим стенам и друг к другу. Напишите программу, которая находит максимальную высоту препятствий и число препятствий максимальной высоты.

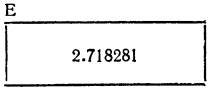
6. Объекты, параметры, типы. Схема вычисления инвариантной функций

Мы уже говорили, что исполнители могут иметь память (глобальные объекты), а предписания исполнителей могут иметь входные и выходные параметры. С тех пор, однако, наша точка зрения на исполнителей несколько изменилась: кроме «железных» базовых исполнителей вроде Резчика металла бывают, как мы теперь знаем, и исполнители, реализованные программно, например такие, как Резчик сита или Обработчик рядов. Естественно, встает вопрос: если мы реализуем исполнителя программно, как описать его глобальные объекты, параметры его предписаний, как с ними оперировать и что вообще в этом случае значит слово память? Ответу на этот вопрос в основном и посвящен настоящий раздел.

Объекты. Будем считать, что наши программы по-прежнему выполняет гипотетический Универсальный Выполнитель (УВ). УВ, несомненно, обладает зачатками интеллекта — во всяком случае он понимает управляющие конструкции и соответствующим образом выполняет их. УВ обладает и памятью — уже при вызовах подпрограмм ему приходится запоминать, какая программа приостановлена и с какого места ее надо продолжить после того, как подпрограмма будет полностью выполнена. Память УВ можно, однако, задействовать и явно для хранения, изменения и получения любой другой информации. В разных частях памяти УВ и в разное время можно хранить разную информацию. Отдельную часть памяти УВ мы и

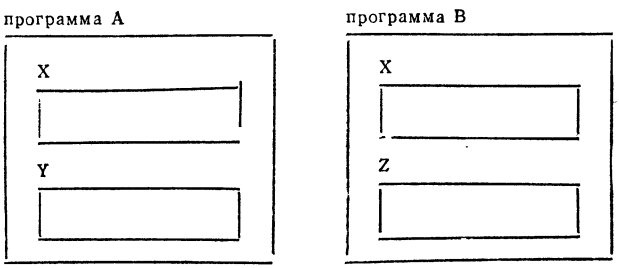
будем называть *объектом*, а информацию, которая в данный момент в этой части хранится, — *состоянием* или *значением* объекта. Для того чтобы в программе можно было указать, какой именно объект мы имеем в виду, объектам даются *имена*, позволяющие отличить их друг от друга. Еще одной характеристикой объекта является *тип* информации, которая хранится в объекте. Например, хранится ли в объекте информация типа да/нет, буква русского алфавита или действительное число.

Можно представлять себе память УВ в виде доски, а объекты в виде прямоугольников, которые ограничивают части доски и внутри которых записывается нужная информация, например:



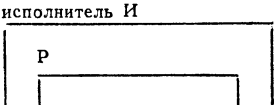
Имена объектов надписываются сверху прямоугольников. Состояние объекта — это то, что записано внутри прямоугольника. Изменение состояния означает стирание старой информации и запись новой.

Имена, которые даются объектам, на самом деле позволяют различать объекты *лишь внутри одной программы*. В разных программах имена, вообще говоря, могут и совпадать. Поэтому правильнее считать, что, начав выполнять какую-то программу, УВ рисует для этой программы на доске большой прямоугольник, а уже внутри него рисует отдельные объекты:



По окончании выполнения программы УВ стирает с доски весь ее прямоугольник со всеми объектами внутри. Таким образом, объект X программы A и объект X программы B — это, вообще говоря, два разных объекта, никак между собой не связанных.

Аналогичным образом УВ заводит область памяти («большой прямоугольник») и при начале работы с реализованным программно исполнителем:



В целом, таким образом, «доска» УВ в каждый момент времени разбита на большие прямоугольники исполнителей и программ, которые разбиты уже на отдельные объекты. Тексты программ обычно также записываются не на отдельных листках, а на этой «доске». В процессе выполнения УВ работает только с доской — читает программы; помечает, где какая программа приостановлена; заводит и стирает прямоугольники объектов, программ и исполнителей; меняет содержимое прямоугольников-объектов и т. п.

Объекты, расположенные внутри прямоугольника программы, называются *локальными объектами программы*, а расположенные внутри прямоугольника исполнителя — *глобальными объектами исполнителя*.

Подчеркнем, что имя локального объекта программы можно использовать только в данной программе (то же имя в другой программе означает совершенно другой объект) и что локальные объекты программы возникают в момент начала выполнения программы и исчезают, когда выполнение программы заканчивается. Про эти свойства локальных объектов принято говорить, что *локальные объекты программы локализованы в программе как по тексту, так и по времени существования*.

Соответственно *глобальные объекты исполнителя локализованы в исполнителе как по тексту, так и по времени существования*. Локализация по тексту означает, что имя глобального объекта исполнителя можно использовать только в тексте данного исполнителя (от слова **исполнитель** до слов **конец исполнителя**) — то же имя в другом исполнителе означает совершенно другой объект. Локализация по времени существования означает, что глобальные объекты исполнителя возникают в момент начала работы с ним (предписание «начать работу») и исчезают после конца (предписание «кончить работу»).

Замечание. Описание объектов в терминах доски и прямоугольников не более, чем модель. По сути дела нас интересуют формы записи программ и смысл этих форм, т. е. то, что происходит при их выполнении. Как это происходит, не очень важно. Мы можем, например, считать, что УВ сначала читает программу целиком, а потом «берет в руки инструменты» и изготавливает устройство с кнопкой (или кнопками), при нажатии на которую система рычагов или электронных схем производит требуемые действия (в частности, в нужной последовательности нажимает на кнопки базовых исполнителей). Таким образом, к модели с доской нужно относиться лишь как к одному из возможных описаний семантики форм записи программ, а не как к объективной реальности.

Локальные объекты программ. Рассмотрим простейший пример. Пусть требуется написать программу, которая с помощью Стекового калькулятора вычисляет произведение чисел от 1 до 10:

программа произведение
дано

```
. получить: | на табло Стекового калькулятора произведение
.           | чисел от 1 до 10
.   объекты :
.   N : число
. --------
.   N := 1
.   СК. добавить ⟨1⟩ в стек
.   цикл
.   . инв: | в вершине стека калькулятора находится
.   .      | произведение чисел от 1 до N
.   . пока N≠10
.   . выполнять
.   .   N. увеличить на 1
.   .   СК. добавить ⟨N⟩ в стек
.   .   СК. умножить
.   конец цикла
.     утв: | в вершине стека калькулятора находится
.          | произведение чисел от 1 до N и N = 10
.   СК. показать результат
конец программы
```

Запись «объекты: N: число» означает, что в программе будет использоваться локальный объект с именем N типа число (т. е. внутрь «прямоугольника» N будут записываться числа). В начале выполнения этой программы УВ «нарисует большой прямоугольник» для этой программы, а внутри него — объект с именем N. Кроме того, он запомнит, что N — это объект типа число.

Строка «N := 1» означает, что объект N надо установить в состояние 1 (т. е. в прямоугольник с именем N надо записать число 1). По историческим причинам такая установка состояния (значения) объекта называется *присваиванием*, знак := называется *операцией присваивания*, а строка в целом читается как «N присвоить 1».

Имена объектов можно использовать в арифметических и логических выражениях, например «N ≠ 10». Встретив имя объекта в выражении, УВ заменяет его на значение этого объекта (подобно тому, как он заменяет на значения имена констант) и только после этого вычисляет значение всего выражения целиком.

Имена объектов могут фигурировать и в качестве фактических параметров при вызовах предписаний и программ. Например, при вызове «СК.добавить ⟨N⟩ в стек» в качестве фактического параметра будет использовано значение объекта N.

Кроме того, над объектами, расположенными в памяти УВ, можно выполнять некоторые операции. Для объектов разных типов

набор операций различен. Скажем, объекты типа **число** можно увеличивать и уменьшать, например «N.**увеличить на 1**». Заметим, что то же самое действие можно было выразить и по-другому — с помощью оператора присваивания: «N := N + 1».

Описание локального объекта программы (т. е. запись «**объекты:** N: **число**») обычно можно опускать. УВ все равно заведет объект с именем N, так как это имя встречается в программе, а его тип постарается определить из контекста. Зная, например, что в вызове «СК.добавить ⟨N⟩ в стек» параметр должен быть числом, УВ сделает вывод, что и объект N имеет тип **число**. Естественно, что если УВ обнаружит противоречия или не сможет определить тип объекта по контексту, то программа выполняться не будет.

Глобальные объекты исполнителей. Подобно тому как локальные объекты программы возникают в момент начала выполнения программы и исчезают, когда выполнение программы заканчивается, глобальные объекты возникают при начале работы с исполнителем и исчезают после конца работы. Изменяя состояния этих объектов, разные программы исполнителя могут передавать информацию друг другу. Глобальные объекты исполнителя описываются в подразделе **объекты** раздела описаний исполнителя (между словами **исполнитель** и **конец описаний**). Форма описаний ничем не отличается от описания локальных объектов программ:

```
исполнитель И
. СП:
.    ...
.
. объекты:
.  P, R : число
.    ...
конец описаний
```

После такого описания во всех программах данного исполнителя (т. е. в программах, расположенных между словами **конец описаний** и **конец исполнителя**) можно использовать имена P и R для указания на соответствующие глобальные объекты исполнителя (оба этих объекта будут иметь тип **число**). Таким образом, внутри программы можно использовать как имена локальных объектов данной программы, так и имена глобальных объектов того исполнителя, к которому относится программа. Естественно, что эти имена не могут совпадать, т. е. ни в одной программе исполнителя и не может быть локальных объектов с именами P и R.

Опускать описания глобальных объектов исполнителя нельзя: если объект не описан ни в исполнителе, ни в программе, то он

считается локальным. Можно, однако, опустить описание типа, например написать

объекты:
P, R : ...

В этом случае УВ будет считать объекты P и R глобальными, а тип постарается определить по контексту.

Входные и выходные параметры программ. Рассмотрим пример:

```
программа сделать ⟨вх : t : Z+⟩ шагов прямо
. дано      : | Путник где-то в квадранте
. получить: | Путник сделал t шагов вперед
. --------
. цикл t раз                                          (*)
. . выполнять
. . Путник. сделать шаг
. конец цикла
конец программы
```

Запись «⟨вх : t : Z+⟩» в имени программы означает, что объект с именем t является входным (вх) формальным параметром (т. е. объектом) программы и имеет тип Z+ (**неотрицательное целое**). Если в какой-то другой программе написать «сделать ⟨5⟩ шагов прямо», то при выполнении этого вызова объект t программы (*) будет иметь состояние 5. Если написать «сделать ⟨7⟩ шагов прямо», то t будет иметь значение 7. Если «сделать ⟨N⟩ шагов прямо», то t будет иметь значение объекта N. А если «сделать ⟨K + 3⟩ шагов прямо», то t будет иметь значение выражения K + 3 (это значение будет вычислено УВ в момент вызова). Аналогично в программе:

```
программа идти до препятствия ⟨вых : расстояние : Z+⟩
. дано      : | Путник где-то в квадранте
. получить: | Путник прошагал прямо до упора, в выходном
.           | объекте — число сделанных Путником шагов
.           | (расстояние до препятствия)
. --------
. расстояние := 0                                     (**)
. цикл
. . утв: | расстояние = число сделанных Путником шагов
. . пока Путник. впереди свободно
. . выполнять
. .  Путник. сделать шаг
. .  расстояние. увеличить на 1
. конец цикла
конец программы
```

запись «⟨**вых** : расстояние : **Z+**⟩» означает, что объект с именем «расстояние» является выходным формальным параметром (объектом) программы и имеет тип **Z+** (**неотрицательное целое**). Если теперь написать «идти до препятствия ⟨**вых** : K⟩», то после выполнения этого вызова в объекте K будет число сделанных Путником шагов (расстояние до препятствия). Если «идти до препятствия ⟨**вых** : M : **Z+**⟩», то число шагов Путника окажется в объекте M и т. п.

Обратите внимание, что для повышения выразительности текста в вызове можно указать, каким именно является параметр (входным, выходным, входно-выходным), а также указать его тип.

Входно-выходные формальные параметры сочетают в себе свойства и входных, и выходных: в момент начала выполнения программы входно-выходной формальный параметр имеет значение соответствующего фактического параметра, указанного в вызове, а после окончания выполнения программы фактический параметр имеет значение, полученное в программе. Естественно, что выражение в качестве фактического выходного или входно-выходного параметра указать нельзя.

Параметры программ не случайно называются «формальными». По сути дела, они являются не объектами, а обозначениями для соответствующих фактических параметров, указанных в вызове. В программе (*), например, объект t не создается («прямоугольник» для t вообще не рисуется) — УВ прямо использует значение фактического параметра (5, 7, N, K + 3). Аналогично при выполнении вызова «идти до препятствия ⟨**вых** : M : **Z+**⟩» УВ не создает объекта «расстояние» в программе (**), а прямо работает с объектом M (вместо «расстояние := 0», например, выполняет «M := 0»). Именно поэтому «расстояние» называется формальным, а M — фактическим параметром программы.

Если параметр является входным, то это значит, что его значение используется в программе, но не меняется в результате ее выполнения. Другими словами, значения входного параметра до и после выполнения программы должны совпадать. Если параметр выходной, то в результате выполнения программы его значение может быть изменено, однако начальное (перед выполнением программы) значение объекта несущественно и в программе не используется. Если параметр входно-выходной, то это значит, что его начальное значение может быть использовано в программе, а в результате выполнения это значение может измениться.

Наконец, заметим, что имена формальных параметров, как и имена локальных объектов программы, локализованы в тексте программы. В качестве обозначений конкретных фактических

параметров формальные параметры действуют от момента вызова программы (начала ее выполнения) и до окончания ее выполнения. В этом смысле можно сказать, что, как и локальные объекты, *формальные параметры локализованы в программе как по тексту, так и по времени существования.*

Программы, вырабатывающие значения. Программу (**), имеющую один выходной параметр, можно записать и в виде программы, вырабатывающей значение:

```
программа расстояние : Z+
. дано     : | Путник где-то в квадранте
. получить: | Путник прошагал прямо до упора, в ответе --
.           | число сделанных Путником шагов
.           | (расстояние до препятствия)
. --------
. ответ := 0
. цикл
. . утв   : | ответ = число сделанных Путником шагов
. . пока Путник.впереди свободно
. . выполнять
. .  Путник.сделать шаг
. .  ответ.увеличить на 1
. конец цикла
конец программы
```

Тогда, написав «М := расстояние», можно получить расстояние до препятствия в объекте М, а написав «N := расстояние», — в объекте N. При выполнении, например, оператора присваивания «М := расстояние» УВ выполнит программу «расстояние» и полученное при этом в ответе значение присвоит объекту М.

Программы, вырабатывающие значение, являются, по сути дела, программами с одним выходным параметром. Запись «программа расстояние : Z+» можно понимать как сокращение от «программа расстояние ⟨вых : ответ : Z+⟩». Основные отличия заключаются в форме *использования*: если программу с одним выходным параметром надо вызывать отдельно, то программу, вырабатывающую значение, можно использовать непосредственно в арифметических и логических выражениях. В целом особенности описания и использования программ, вырабатывающих значения, иллюстрируются табл. 6.1. Выбрав из двух приведенных в этой таблице путей использования программы тот, который нам больше нравится (это дело вкуса), мы можем соответственно реализовать ее либо как программу с одним выходным параметром, либо как программу, вырабатывающую значение.

Таблица 6.1

Программы, вырабатывающие значение	Программы с одним выходным параметром
Описание	
программа G: да/нет	**программа** G (**вых**: q: да/нет)
Использование	
если G **то** ...	G (**вых**: успех) **если** успех **то** ...
Описание	
программа F (**вх**: X) : R	**программа** F (**вх**: X, **вых**: Y: R)
Использование	
B:=F (A)	F (A, B)
	F (X1, Y1) F (X2, Y2)
Z:=(F (X1)+F (X2))/2.0	Z:=(Y1+Y2)/2.0

Типы объектов. Для объектов разных типов УВ может выполнять разные наборы операций. Таким образом,

> тип характеризует множество состояний, которые может принимать объект, и множество операций, которые над ним допустимы.

Например, состоянием (значением) объекта типа **число** может быть любое действительное число (множество состояний — множество действительных чисел). Над этим объектом допустимы любые арифметические операции и операции сравнения (при использовании имени объекта в выражениях), а кроме того, операции присваивания, увеличения и уменьшения на некоторую величину (т. е. на значение некоторого выражения) и др. Естественно, возникает вопрос: какие еще типы, кроме типа **число**, могут иметь объекты в программе? Ответ на этот вопрос звучит так:

> Объекты программы либо могут быть некоторых предопределенных типов (эти типы мы сейчас перечислим), либо построены с помощью способов конструирования из других объектов (подобъектов). Подобъекты в свою очередь могут быть либо предопределенными, либо построены из подподобъектов и т. п. В конечном счете любой объект либо имеет предопределенный тип, либо строится из объектов предопределенных типов с помощью, быть может неоднократного, применения различных способов конструирования.

Способы конструирования объектов и типов будут изложены в следующем разделе. Предопределенными типами являются следующие:

— **Q** (да/нет),
— **S** (символ),
— **Z** (целое),
— **Z+** (неотрицательное целое),
— **R** (действительное число или просто число).

Значки **Q**, **S**, **Z**, **Z+**, **R** и слова в скобках являются синонимами, т. е. запись «**объекты : N : число**» эквивалентна записи «**объекты : N : R**». Таким образом, существует всего пять предопределенных типов. Опишем их.

Множество значений для каждого из типов **Z**, **Z+**, **R** ясно следует из названия типа. Объект типа **да/нет** может принимать лишь одно из двух значений — либо **да**, либо **нет**. Значением объекта типа **символ** является символ: либо буква, либо цифра, либо специальный значок (например, "+", "—", "≠", "≥", "∈" и т. п.). Существует также символ " ", который называется пробелом и используется для разделения слов в последовательности символов. Множество различных символов содержит всего 256 элементов и некоторым образом упорядочено. Этот порядок сохраняет порядок букв в алфавите ("a"<"b"<"c") и порядок цифр ("0"<"1"<"2").

Объект любого типа может иметь значение **неопр** (т. е. быть неопределенным). В множество значений любого упорядоченного типа (**S**, **Z**, **Z+**, **R**) входят, кроме того, два идеальных значения $-\infty$ и $+\infty$, которые могут использоваться только в операторе присваивания и в операциях сравнения.

Операции. Над объектами любого предопределенного типа допустимы операции присваивания (:=), сравнения на равенство (=) и неравенство (≠). Если тип упорядоченный (**S**, **Z**, **Z+**, **R**), то допустимы также сравнения на <, ≤, >, ≥. В операциях сравнения для упорядоченных типов наряду с обычными могут участвовать и идеальные значения $-\infty$ и $+\infty$. Значение **неопр** может участвовать только в операциях присваивания, сравнения на равенство и неравенство.

Объекты и различные функции от них могут входить в арифметические и логические выражения, которые подчиняются общепринятым правилам записи. В число функций входят максимум, минимум, остаток, знак, модуль (абсолютная величина) и др. (см. приложение 1). При записи арифметических выражений знак — означает вычитание, + — сложение, * — умножение, / — деление и ** — возведение в степень.

Если в операторе присваивания слева стоит имя объекта типа **Z** или **Z+**, а в результате вычисления значения выражения в правой

части получилось нецелое число, то присваивается *целая часть* полученного значения. При попытке присвоить объекту типа **Z+** отрицательное число происходит отказ.

Наконец, имеется некоторое количество других операций, например операции **увеличить на** и **уменьшить на** для числовых типов, операции **следующий** и **предыдущий** для упорядоченных дискретных типов и т. п. Мы будем вводить их по мере надобности.

Пусть есть объекты:

```
q : Q
s : S
n : Z+
z, A, B, C, D : Z
r : R
```

Приведем несколько примеров фрагментов программ с использованием этих объектов, арифметических и логических выражений:

```
q := нет
s := "a"
n := 0
z := — 137
r := 2.71828
q := (n = 3 или r ⩾ z)
если s = " " или не q то ...
если "a" ⩽ s ⩽ "я" или
"0" ⩽ s ⩽ "9" то ...
цикл пока s ≠ " " выполнять ...
A := 7
B := — 49
C := (A + 5)/15 — B
если A + B < 4*C — 1 ⩽ D то
D := максимум (A, B) + частное (B + 3, C)
```

Описания типов. Как мы видели, тип локальных и глобальных объектов можно указать в разделе **объекты**, расположенном соответственно в программе или в исполнителе, в заголовке программы можно указать тип формальных, а в вызове — тип фактических параметров. Описание типа всюду можно опускать, тогда тип будет определен по контексту.

Кроме этого, тип любого объекта можно указать в **дано**/**получить** или в утверждении. Программу (**), например, можно записать в виде:

```
программа идти до препятствия ⟨вых : расстояние⟩
. дано      : | Путник где-то в квадранте
. получить: расстояние: Z+ | расстояние до препятствия
```

```
.              | (число сделанных Путником шагов прямо)
.              | Путник прошагал прямо до упора
. --------
. расстояние := 0
. цикл
. . утв  : | расстояние = число сделанных Путником шагов
. . пока Путник.впереди свободно
. . выполнять
. .  Путник.сделать шаг
. .  расстояние.увеличить на 1
. конец цикла
конец программы
```

Таким образом, на запись «расстояние: **Z+**» можно смотреть как на утверждение о типе объекта. Таких утверждений, естественно, может быть много, т. е. тип объекта «расстояние» можно указать и в заголовке программы, и в дано/получить, и в утверждениях внутри программы. УВ лишь контролирует непротиворечивость таких описаний и их соответствие использованию объекта (например, из строки «расстояние := 0» следует, что «расстояние» — объект числового типа). Хотя описания типа всюду можно опускать, в программе должно быть достаточно информации для определения типа по контексту, поэтому тип каких-то объектов где-то описывать надо. Мы обычно будем описывать тип глобальных объектов исполнителя и тип формальных параметров программ (в заголовке или в дано/получить). Наоборот, тип локальных объектов программ, как правило, описываться не будет.

Описания типов в дано/получить, вызовах и утверждениях являются как контекстом, исходя из которого УВ может определить тип объекта, так и средством контроля, уменьшающим вероятность ошибки в программе. Кроме того, они могут быть использованы для того, чтобы сделать программу более понятной и легче читаемой.

Примеры программ, использующих объекты предопределенных типов.

```
программа число корней квадратного уравнения (вх: P, Q): Z+
. дано     : P, Q: число | коэффициенты уравнения
             x** 2 + P*x + Q = 0
. получить: | число действительных корней этого уравнения
. --------
. D := P*P — 4*Q
. выбор
. . при D > 0 ⇒ ответ := 2
. . при D = 0 ⇒ ответ := 1
```

```
. . при D < 0 ⇒ ответ := 0
. конец выбора
конец программы

программа переставить циклически влево ⟨вх/вых : A, B, C : число⟩
. дано      :
. получить: | значения A, B, C циклически сдвинуты влево
. --------
.  R := A
.  A := B
.  B := C
.  C := R
конец программы

программа число различных ⟨вх : A, B, C : число⟩ : Z+
. дано      : | три действительных числа
. получить: | количество различных среди них
. --------
. выбор
. . при A = B = C      ⇒ ответ := 1
. . при A ≠ B ≠ C ≠ A  ⇒ ответ := 3
. .      иначе         ⇒ ответ := 2
. конец выбора
конец программы
```

Обратите внимание на условие «A = B = C». Это условие в более развернутой форме можно записать так: «A = B и B = C». Аналогично «A ≠ B ≠ C ≠ A» означает «A ≠ B и B ≠ C и C ≠ A», «a ⩽ x ⩽ b» означает «a ⩽ x и x ⩽ b» и т. д.

```
программа старшая цифра ⟨вх : n0⟩ : Z+
. дано      : n0 : Z+ | неотрицательное целое число
. получить: | старшую цифру десятичного разложения n0
. --------
.  n := n0
.  цикл пока n ⩾ 10 выполнять
.  .  n := целое (n/10)
.  конец цикла
.  ответ := n
конец программы
```

При выполнении этой программы с n0 = 3271 объект n будет последовательно принимать состояния 3271, 327, 32, 3. Последнее значение и будет присвоено ответу (т. е. выработано в качестве значения программы). Стандартная функция «целое» получает целую часть аргумента, в данном случае выражения n/10.

```
программа две старшие цифры ⟨вх: n0, вых: c1,c2⟩
. дано      : n0: Z+ | неотрицательное целое число
. получить: c1,c2: Z+ | две старшие цифры числа n0. Если n0
.           | однозначное, то c1 должно быть равно 0,
.           | а c2 = n0
. --------
. n := n0
. цикл пока n ⩾ 100 выполнять
. .  n := целое (n/10)
. конец цикла
. c1 := целое (n/10)
. c2 := остаток (n,10)
конец программы
```

Схема вычисления значения инвариантной функции. В разд. 4 были описаны базисные схемы обработки информации «рекурсия», «итерация» и «проектирование цикла с помощью инварианта». Сейчас мы приведем еще одну базисную схему — схему вычисления инвариантной функции. Пусть M — некоторое множество, $F: M \to Y$ — заданная функция и требуется определить значение $F(x0)$ функции F в некоторой точке $x0 \in M$.

Рассмотрим предикат $P: M \to \{\text{да, нет}\}$, заданный следующим образом:

$P(x) = $ **да** $\Leftrightarrow$ $F(x)$ легко вычислить.

Если существует преобразование $T: M \setminus P \to M$ такое, что F является T-инвариантной функцией (т. е. $F(T(x)) = F(x)$ для любого $x \in M \setminus P$), то можно применить метод итераций (разд. 4) и, получив $P(x) =$ **да**, вычислить $F(x)$ вместо $F(x0)$:

```
x := x0
цикл пока не  P(x)
. выполнять
. применить к ⟨x⟩ преобразование T
конец цикла
ответ  := F(x)
```

Графически это изображено на рис. 6.1, где

$$I = \{x \in M: F(x) = F(x0)\}$$

T-инвариантность F обеспечивает постоянство значений функции на траектории $\{x \in M: x = T^n(x0), n \in Z+\}$. Именно это и позволяет вычислять значение функции не в точке $x0$, а в любой точке x этой траектории.

Рассмотрим пример. Пусть требуется определить наименьший общий делитель двух чисел a и b, где $a, b \in Z+$, $a + b > 0$. Рас-

смотрим эту задачу как задачу вычисления значения функции $F: M \to Y$. Тогда $M = Z+ * Z+ \setminus \{0, 0\}$, а $Y = N$ (множество натуральных чисел).

Сформулируем предикат P, т. е. опишем множество пар $(a, b) \in M$, для которых НОД вычисляется легко. Очевидно, что

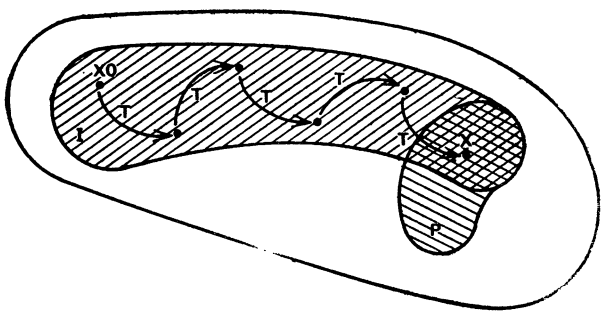


Рис. 6.1. Схема вычисления инвариантной функции

если $a = 0$, то НОД$(a, b) = b$, и, наоборот, если $b = 0$, то НОД$(a, b) = a$. Поэтому положим

$$P(a, b) = (a = 0 \text{ или } b = 0).$$

Таким образом, если $P(a, b) =$ да, то НОД$(a, b) = a + b$.

Осталось придумать преобразование T, которое бы сохраняло НОД и «приближало» к множеству $\{x \in M: P(x) = \text{да}\}$. Так как нужно сохранять НОД, то посмотрим на его свойства. Поскольку

$$\text{НОД}(a, b) = \text{НОД}(a - b, b) = \text{НОД}(a, b - a),$$

определим преобразование $T: M \setminus P \to M$ следующим образом:

$$T(a, b) = \begin{cases} (a - b, b), & \text{если } a \geqslant b > 0, \\ (a, b - a), & \text{если } b \geqslant a > 0. \end{cases}$$

Очевидно, что для любого $x0 \in M$ существует $n \in Z+$ такое, что $P(x) =$ да, где $x = T^n(x0)$. Итак, решение задачи можно записать в виде

```
программа НОД (вх: a0, b0: Z+): Z+
. дано      : a0 + b0 > 0
. получить: ответ > 0 | ответ = НОД(a0, b0)
. --------
. a := a0; b := b0
. цикл
```

```
. . инв: | НОД(a,b) = НОД(a0,b0)
. . пока a > 0 и b > 0
. . выполнять
. .  выбор
. .  . при a ⩾ b ⇒ a := a − b
. .  . при b ⩾ a ⇒ b := b − a
. .  конец выбора
. конец цикла
.  утв: a = 0 или b = 0 | и НОД(a,b) = НОД(a0,b0)
. ответ := a + b
конец программы
```

ЗАДАЧИ И УПРАЖНЕНИЯ

В задачах 1—13 напишите программы, которые находят:

1. Количество максимальных чисел среди A, B, C.
2. Количество разных чисел среди A, B, C.
3. Количество цифр в десятичной записи числа n.
4. Количество цифр 7 в десятичной записи числа n.
5. Сумму цифр в десятичной записи числа n.
6. Первую цифру дробной части положительного числа n.
7. Количество корней биквадратного уравнения $x^4 + Px^2 + Q = 0$.
8. Количество целых чисел, удовлетворяющих неравенству $x^2 + Ax + B < 0$.
9. Количество целых точек в круге $(x - a)^2 + (y - b)^2 \leqslant r^2$.
10. Длину интервала отрицательности функции f(x) = максимум(a − x, b − 2x, x + c).
11. sin(sin(... sin(x) ...)) — всего N раз.
12. a*b*a*b*a*b*... — всего N знаков умножения.
13. Расстояние от точки плоскости с координатами x0, y0 до
    а) круга радиусом r > 0 с центром в точке x1, y1;
    б) прямой y = ax + b;
    в) отрезка с концами в точках (ax, ay) и (bx, by).
14. Напишите программу, которая определяет, пересекаются ли два отрезка на плоскости, заданные координатами своих концов.
15. С помощью схемы вычисления инвариантной функции напишите программу, находящую НОД двух чисел на основе тождества НОД(2x, 2y + 1) = НОД(x, 2y + 1).
16. С помощью схемы проектирования цикла с помощью инварианта напишите программу, которая находит сумму ряда для sin(x) с заданной точностью.
17. С помощью схемы проектирования цикла с помощью инварианта напишите программу, которая находит n-е число Фибоначчи.

7. Способы конструирования типов, объектов и исполнителей

Перечисление. Простейшим способом конструирования объектов является перечисление:

объекты:
А: **перечисление** (пн, вт, ср, чт, пт, сб, вс)

Перечисление — это способ конструирования объектов «из ничего»: мы явно перечисляем (откуда и название способа) все возможные состояния данного объекта. Например, описанный выше объект А может принимать ровно семь разных «обычных» значений (эти значения обозначаются словами пн, вт, ср, чт, пт, сб, вс). Кроме того, как и любой другой объект любого упорядоченного типа, А может принимать три идеальных значения: $-\infty$, $+\infty$ и **неопр**. Значения считаются упорядоченными в порядке их перечисления:

$$-\infty < \text{пн} < \text{вт} < \text{ср} < \text{чт} < \text{пт} < \text{сб} < \text{вс} < +\infty.$$

Можно, например, написать

А := пн
если А = вс **то** ...
если пн $\leqslant$ А $\leqslant$ пт **то** ...

Таким образом, «пн» для объекта А является таким же обозначением состояния, как и **да** для объекта типа **да/нет** или 1 для объекта типа **число**.

Для объекта, заданного перечислением, определены все стандартные операции с упорядоченным дискретным типом: присваивание, сравнения на равенство и неравенство, сравнения по порядку, операции **следующий**, **предыдущий**:

А := А.**предыдущий**
цикл пока А.**следующий** $\neq +\infty$ **выполнять** ...

С помощью способов конструирования можно конструировать не только объекты некоторых новых типов, но и сами эти новые типы:

типы:
день недели = **перечисление** (пн, вт, ср, чт, пт, сб, вс)
объекты:
А: день недели

Для типов, построенных с помощью перечисления, как и вообще для любого упорядоченного типа с конечным множеством значений, существует специальная управляющая конструкция — цикл

для каждого, — позволяющая перебрать все значения данного типа. В общем виде эта конструкция записывается как

цикл для каждого x **из** ***типа*** T
. **выполнять**
. ⟨действия⟩
конец цикла

или

цикл для каждого x **из** ***типа*** T
. **пока** ⟨условие⟩
. **выполнять**
. ⟨действия⟩
конец цикла

Слово ***типа***, показанное курсивом, можно не писать. Вместо слов **для каждого** и **из** можно использовать значки $\forall$ и $\in$ соответственно. Например, в фрагменте программы

цикл $\forall$ A $\in$ день недели **выполнять**
. обработать ⟨**вх**: A⟩
конец цикла

программа «обработать» будет вызвана сначала при A = пн, затем при A = вт и т. д. до A = вс (всего семь раз). После окончания выполнения цикла объект A будет иметь значение **неопр.**

Элементы упорядоченного типа с конечным множеством значений можно перебрать и в обратном порядке. Для этого нужно после итератора «**для каждого** x **из типа** T» написать служебное слово **реверс** (от англ. reverse — обратный, противоположный). Например, в фрагменте программы

цикл $\forall$ A $\in$ день недели **реверс выполнять**
. обработать ⟨**вх**: A⟩
конец цикла

программа «обработать» будет вызвана сначала при A = вс, затем при A = сб и т. д. до A = пн (всего семь раз). После окончания выполнения цикла объект A будет иметь значение **неопр.**

Отрезок. Способ конструирования **отрезок** позволяет ограничить множество состояний любого дискретного типа. Например, запись

объекты:
x: 1..**31**
y: пн..**пт**

означает, что объект x может принимать целые значения в диапазоне от 1 до 31 включительно, а объект y — дни недели в диапазоне от пн до пт включительно. (Кроме этих «обычных» значений объекты x и y, как обычно, могут принимать идеальные значения $-\infty$,

$+\infty$ и **неопр**. Над сконструированными таким образом объектами допустимы те же операции, что и над объектами исходного типа, из которого «вырезался» отрезок: над x допустимы те же операции, что и над целыми числами, а над y — те же, что и над объектами типа «день недели». Единственное ограничение состоит в том, что при изменении состояния объекта новое значение должно лежать в указанном диапазоне (если y = пт, то **y. следующий** = $+\infty$).

Вместо конструирования конкретного объекта нового типа можно, как уже отмечалось выше, сначала сконструировать сам тип, а потом использовать объекты этого типа:

типы:
день месяца = 1..31
рабочий день = пн..пт
объекты:
x: день месяца
y: рабочий день

Поскольку множество значений сконструированного таким образом типа конечно и упорядоченно, можно использовать циклы **для каждого:**

цикл $\forall$x $\in$ день месяца **выполнять** ...
цикл $\forall$x $\in$ день месяца **реверс выполнять** ...

Кроме того, способом конструирования **отрезок** можно пользоваться и прямо в цикле **для каждого:**

цикл $\forall$x $\in$ 1..31 **выполнять** ...
цикл $\forall$y $\in$ пн..пт **реверс выполнять** ...
цикл $\forall$z $\in$ сб..вс **выполнять** ...

Запись. Способ конструирования **запись** позволяет объединить несколько объектов в один, более сложный:

объекты:
дата1, дата2: **запись** (год : Z+
месяц : 1..12
день : 1..31
день недели: день недели)

Графически запись можно представлять в виде прямоугольника, разбитого на некоторое число меньших прямоугольников:

дата1:

год	месяц	день	день недели

Каждый маленький прямоугольник (он называется *полем записи*) имеет имя, действующее только внутри большого прямоугольника (записи в целом). С каждым таким маленьким прямоугольником можно работать как с обычным объектом, только надо указывать, внутри какого большого прямоугольника он находится:

```
дата2. год := дата1. год + 100
если дата1. день = 31 то ...
если пн ≤ дата2. день недели ≤ пт то ...
```

Таким образом, запись является произвольным формальным объединением каких-то объектов. С каждым из этих объектов можно работать независимо. Кроме того, однако, допустимы некоторые операции над записью в целом: сравнение на равенство и неравенство, присваивание и передача в качестве параметра:

```
дата1 := дата2
если дата1 = дата2 то ...
обработать ⟨вх: дата1, вых: дата2⟩
```

В качестве еще одного примера работы с записями приведем программу, которая находит пересечение двух отрезков на прямой:

```
программа отрезок ⟨вх/вых: A⟩ пересечь с отрезком ⟨вх: B⟩
. дано: A, B : запись (начало, конец : R) | отрезки на прямой
. получить   : | в A пересечение отрезков A и B
.--------
. A. начало := максимум (A. начало, B. начало)
. A. конец := минимум (A. конец, B. конец)
конец программы
```

Структуры. Прежде чем двигаться дальше, рассмотрим исполнителей стек, дек, очередь, список, последовательность, множество, нагруженное множество, вектор, матрица и динамический вектор. Каждый из этих исполнителей работает с конечной совокупностью некоторых объектов (элементов) одного и того же типа E. Например, если тип E = **целое**, то это значит, что исполнитель работает с какой-то совокупностью целых чисел; если E = **да/нет**, то с совокупностью элементов типа да/нет и т. п. Между собой исполнители различаются тем, *как именно* они работают с элементами. Например, из стека элементы можно доставать только в порядке, обратном порядку помещения их в стек, а из очереди, наоборот, — только в том порядке, в котором элементы в очередь помещались. Далее мы будем называть эти совокупности элементов *структурами* и говорить о структуре стека или структуре очереди соответственно.

Все перечисленные выше структуры, за исключением вектора и матрицы, являются *динамическими*, т. е. число элементов в них

может изменяться в процессе работы. Исполнитель, соответствующий любой динамической структуре, имеет предписания

2. сделать пустым
3. пуст/не пуст :да/нет

где слово «пусто» означает структуру, не содержащую ни одного элемента.

Стек элементов типа E. Система предписаний:

1. начать работу
2. сделать *стек* пустым
3. *стек* пуст/*стек* не пуст :да/нет
4. добавить *элемент* ⟨вх : E⟩ *в стек*
5. взять *элемент из стека в* ⟨вых : E⟩
6. вершина *стека* ::E
7. удалить вершину *стека*
8. кончить работу

Курсивом показаны части имени предписания, которые можно опускать. Таким образом, «Стек.добавить ⟨E⟩», «Стек.добавить элемент ⟨E⟩ в стек», «Стек.добавить элемент ⟨E⟩» и «Стек.добавить ⟨E⟩ в стек» означают одно и то же. Структура стека в целом уже

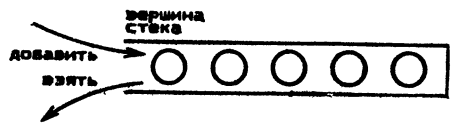


Рис. 7.1. Стек

знакома нам по Стековому калькулятору (разд. 1). Напомним, что стек можно представлять себе в виде трубы с одним запаянным концом, куда помещаются «бочонки»-элементы (рис. 7.1).

По предписанию «начать работу» стек начинает работу. После выполнения этого предписания, как и после предписания «сделать пустым», стек пуст, т. е. не содержит ни одного элемента. Предписания «пуст»/«не пуст» анализируют, есть ли в стеке элементы, и вырабатывают в качестве ответа соответствующее значение — либо «да», либо «нет». Предписания «добавить» и «взять» выполняются в соответствии со структурой стека: по предписанию «взять» из стека изымается элемент, который был добавлен последним, и его состояние присваивается фактическому выходному параметру. Предписание «удалить вершину» отличается от предписания «взять» только тем, что состояние изымаемого элемента (вершины стека) никуда не попадает, а безвозвратно теряется.

Наконец, запись ::E в предписании «вершина» означает, что это предписание в зависимости от вида использования *либо прини-*

мает, либо вырабатывает значение типа Е. Как и всякое предписание, вырабатывающее значение, «вершину» можно использовать в выражениях и в качестве фактического входного параметра. Кроме того, однако, это предписание можно использовать и в левой части оператора присваивания и в качестве выходных и входно-выходных фактических параметров. Например, можно написать «Стек.вершина := 0» или «Стек.вершина := Стек.вершина + 1».

Таким образом, запись «Стек.вершина» можно рассматривать как обозначение для объекта-вершины стека. Заметим, что при этом используется или меняется только состояние вершины стека, а число элементов в стеке не изменяется. Естественно, что если в стеке нет элементов, то предписания «взять», «удалить» и «вершина» приводят к отказу.

Очередь элементов типа Е. Система предписаний:

1. начать работу
2. сделать *очередь* пустой
3. *очередь* пуста/*очередь* не пуста : да/нет
4. добавить *элемент* ⟨вх : Е⟩ *в конец очереди*
5. взять *элемент из начала очереди в* ⟨вых : Е⟩
6. начало *очереди* ::Е
7. удалить начало *очереди*
8. кончить работу

Если стек можно представлять себе в виде трубы с одним запаянным концом, то очередь — это труба с двумя открытыми концами, движение в которой, однако, возможно лишь в одном направленни (рис. 7.2).

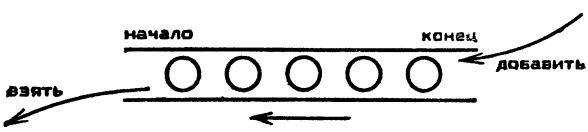


Рис. 7.2. Очередь

Очередь является аналогом стека, с той лишь разницей, что добавление элементов производится в конец очереди (на рис. 7.2 справа), а взять, посмотреть, изменить или удалить можно лишь начало (на рис. 7.2 слева).

Дек элементов типа Е. Система предписаний:

1. начать работу
2. сделать *дек* пустым
3. *дек* пуст/*дек* не пуст : да/нет
4. добавить *элемент* ⟨вх : Е⟩ в начало/конец *дека*
5. взять *элемент* из начала/конца *дека в* ⟨вых : Е⟩

6. начало/конец *дека* ::Е
7. удалить начало/конец *дека*
8. кончить работу

Дек является симбиозом стека и очереди — это та же труба с двумя открытыми концами, с которой на сей раз можно работать с обоих концов (рис. 7.3).

Таким образом, если мы работаем с деком только с левого края («добавить в начало», «взять из начала», «начало», «удалить начало»), то фактически получается стек. Аналогично мы получим стек, если будем работать только с правого края (конца дека). Пользуясь

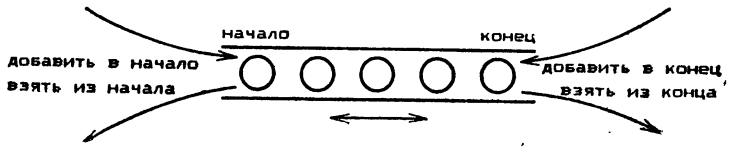


Рис. 7.3. Дек

предписаниями «добавить в конец» и «взять из начала», мы можем получить очередь, элементы которой будут продвигаться справа налево. С помощью противоположной пары предписаний можно получить очередь, элементы которой движутся слева направо. В целом же эта структура называется деком (сокращение от английского double ended queue — очередь с двумя концами).

Множество элементов типа Е. Система предписаний:

1. начать работу
2. сделать *множество* пустым
3. *множество* пусто/*множество* не пусто : да/нет
4. добавить *элемент* ⟨**вх** : Е⟩ *в множество*
5. удалить *элемент* ⟨**вх** : Е⟩ *из множества*
6. *элемент* ⟨**вх** : Е⟩ принадлежит *множеству* : да/нет
7. взять *какой нибудь элемент множества в* ⟨**вых** : Е⟩
8. кончить работу

В отличие от всех предыдущих структур множество является структурой, элементы которой не упорядочены. Предписание «взять», например, выдает какой-то элемент множества. Был ли он добавлен первым, вторым, *n*-м или последним, сказать нельзя. Как обычно, по предписанию «взять» элемент из множества забирается «материально», т. е. после этого предписания взятого элемента в множестве не будет. Предписания «добавить» и «удалить» следует понимать в теоретико-множественном смысле: если добавляемый элемент уже был в множестве или если удаляемого элемента в множестве нет, то отказа не возникает, а множество не изменяется.

Предписание «элемент ⟨вх : Е⟩ принадлежит множеству» берет состояние фактического параметра и проверяет, есть ли такой элемент в множестве. Множество при этом никак не меняется.

Нагруженное множество. Система предписаний:

1. начать работу
2. сделать *множество* пустым
3. *множество* пусто/*множество* не пусто : да/нет
4. добавить *элемент* ⟨вх : Е⟩ *в множество*
5. удалить *элемент* ⟨вх : Е⟩ *из множества*
6. *элемент* ⟨вх : Е⟩ принадлежит *множеству* : да/нет
7. взять *какой нибудь элемент множества в* ⟨вых : Е⟩
8. нагрузка ⟨вх : Е⟩ ::Y
9. кончить работу

Нагруженное миожество отличается от обычного тем, что каждый элемент можно «нагрузить» — сопоставить ему некоторое значение определенного типа Y. Это делается с помощью предписания «нагрузка», принимающего и вырабатывающего значения типа Y. Если этим предписанием не пользоваться, то внешне нагруженное множество не отличимо от обычного. Запись

М.нагрузка ⟨вх : е⟩: = у

устанавливает нагрузку элемента е множества М в состояние у независимо от того, какова была нагрузка этого элемента раньше. Если в предписании «нагрузка» указан элемент, не принадлежащий множеству, то возникает ситуация **отказ**. После добавления элемента к множеству независимо от того, был уже такой элемент в множестве или нет, была ли у него нагрузка определена или нет, состояние одинаково — новый элемент принадлежит множеству и его нагрузка имеет состояние **неопр**.

Последовательность элементов типа Е. Система предписаний:

1. начать работу
2. сделать *последовательность* пустой
3. *последовательность* пуста/не пуста : да/нет
4. добавить *элемент* ⟨вх : Е⟩ *в конец последовательности*
5. встать в начало *последовательности*
6. есть/нет непрочитанные *элементы* : да/нет
7. прочесть *очередной элемент последовательности в* ⟨вых : Е⟩
8. очередной *элемент последовательности* ::Е
9. Пропустить *очередной элемент последовательности*
10. кончить работу

Совокупность элементов, с которыми работает этот исполнитель, является линейно упорядоченной. Элементы последовательности в каждый момент времени разделены на две части — прочитанную и непрочитанную (рис. 7.4).

После предписаний «начать работу» и «сделать пустой» прочитанная и непрочитанная части последовательности пусты (не содержат ни одного элемента). По предписанию «добавить ⟨вх : E⟩ в конец» элемент добавляется в конец последовательности (непрочитанная часть при этом увеличивается, а прочитанная не изменяется).

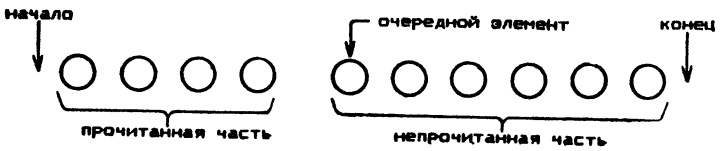


Рис. 7.4. Последовательность

По предписанию «встать в начало» прочитанная часть делается пустой, а непрочитанная совпадает со всей последовательностью. Предписание «есть непрочитанные элементы» отвечает да, если в непрочитанной части есть элементы, и нет, если эта часть пуста.

Непрочитанная часть последовательности является аналогом очереди. Очередным элементом (аналог начала очереди) называется первый элемент непрочитанной части. По предписаниям «прочесть» и «пропустить» (аналоги «взять» и «удалить» для начала очереди) очередной элемент перемещается из непрочитанной части в прочитанную (т. е. прочитанная часть увеличивается на этот элемент, непрочитанная уменьшается и очередным становится следующий элемент последовательности).

Л2-список элементов типа E. Система предписаний:

1. начать работу
2. сделать *список* пустым
3. *список* пуст/*список* не пуст : да/нет
4. установить *указатель* в начало/конец *списка*
5. *указатель* в начале/конце *списка* : да/нет
6. *передвинуть указатель списка* вперед/назад
7. добавить *элемент* ⟨вх : E⟩ до *указателя*/за *указателем списка*
8. взять *элемент списка* до *указателя*/за *указателем в* ⟨вых : E⟩
9. *элемент списка* до *указателя*/за *указателем* ::E
10. удалить *элемент списка* до *указателя*/за *указателем*
11. кончить работу

Л2-список — линейный двунаправленный список — можно представлять себе в виде перекидного календаря или своего рода бус из элементов типа E (рис. 7.5).

Элементы списка линейно упорядочены. Положение перед первым элементом списка называется *началом* списка, положение за последним элементом называется *концом* списка. Кроме элементов в списке имеется еще *указатель*, который можно представлять себе

стрелочкой, расположенной в начале, в конце или между элементами списка (в перекидном календаре указатель — это место, на котором раскрыт календарь).

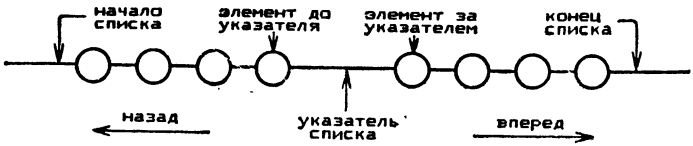


Рис. 7.5. Л1-список

После предписаний «начать работу» и «сделать список пустым» список становится пустым, т. е. не содержит элементов. Для пустого списка понятия «начало» и «конец» существуют и совпадают, а указатель расположен одновременно и в начале, и в конце списка (рис. 7.6).

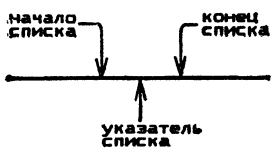


Рис. 7.6. Пустой Л2-список

По предписанию «передвинуть указатель вперед/назад» указатель смещается на один элемент соответственно вперед (от начала к концу списка) либо назад (от конца к началу).

Если пользоваться только предписаниями «добавить»/«взять»/«элемент»/«удалить», а сам указатель не двигать, то части списка до указателя и за указателем представляют собой полные аналоги стеков (рис. 7.7).

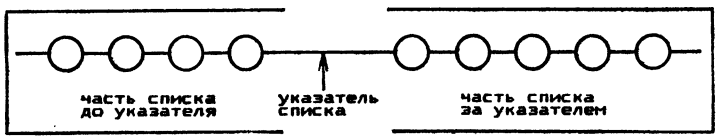


Рис. 7.7

Название списка «линейный двунаправленный» означает, что элементы списка упорядочены линейно, а указатель можно двигать и вперед, и назад.

Л1-список элементов типа E. Система предписаний:

1. начать работу
2. сделать *список* пустым
3. *список* пуст/*список* не пуст : да/нет
4. установить *указатель* в начало *списка*
5. *указатель* в конце *списка* : да/нет
6. *передвинуть указатель списка* вперед
7. **добавить** *элемент* ⟨вх : E⟩ за *указателем списка*

8. **взять** *элемент списка* **за** *указателем в* (**вых** : Е)
9. *элемент списка* **за** *указателем* ::Е
10. удалить *элемент списка* за *указателем*
11. кончить работу

Л1-список — линейный однонаправленный список — отличается от Л2-списка тем, что он обеспечивает последовательный просмотр элементов списка только от начала к концу (встать в начало, пока не в конце передвинуть вперед). Передвинуть указатель назад нельзя. Кроме того, в Л1-списке работать (добавлять/брать/получать/менять/удалять элементы) можно только за указателем, т. е. впереди по ходу движения. Образно говоря, в Л1-списке только одна «сторона» указателя является активной.

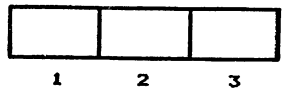


Рис. 7.8. Вектор с индексом типа 1 .. 3

Вектор элементов типа Е с индексом типа И. Система предписаний:

1. начать работу
2. *элемент вектора с индексом* (**вх**: И) ::Е
3. кончить работу

Вектор представляет собой совокупность элементов типа Е, «пронумерованную» значениями типа И (номер элемента называется его *индексом*). На рис. 7.8 изображен вектор при И = 1 .. 3.

Множество значений типа И должно быть конечно: тип И должен быть либо равен да/нет, либо задан перечислением, либо получен как отрезок из дискретного типа. Например, на рис. 7.9 изображен вектор при И = перечисление(пн, вт, ср, чт, пт, сб, вс).

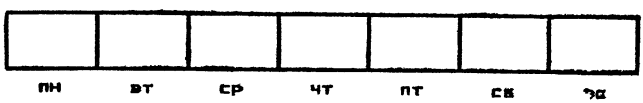


Рис. 7.9. Вектор с индексом типа «день недели»

После предписания «начать работу» все элементы вектора существуют и все имеют состояние **неопр**. С помощью предписания «элемент вектора с индексом (и)» можно работать с любым элементом вектора, используя его индекс. Если исполнитель имеет имя «вектор», то вместо «вектор.элемент вектора с индексом (и)» можно сокращенно писать «вектор(и)». Например, можно написать

вектор (пн) := Е
вектор (вс) := вектор (сб)

и т. п.

Матрица элементов типа Е с индексами типа И1, И2. Система предписаний:

1. начать работу
2. *элемент матрицы с индексами* ⟨вх : И1, И2⟩ ::Е
3. кончить работу

Матрица является полным аналогом вектора с той лишь разницей, что индексов у нее два. Соответственно представлять матрицу

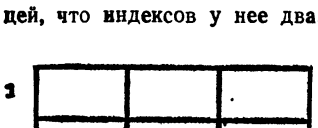


Рис. 7.10. Матрица с индексами типа 1 . . 2, 1 . . 3

надо не в виде линейной последовательности элементов, а в виде прямоугольной таблицы (рис. 7.10).

Динамический вектор элементов типа Е. Система предписаний:

1. начать работу
2. сделать *вектор* пустым
3. *вектор* пуст/*вектор* не пуст : да/нет
4. число *элементов вектора* : Z+
5. добавить *элемент* ⟨вх : Е⟩ *в конец вектора*
6. удалить *элемент из конца вектора*
7. *элемент вектора с индексом* ⟨вх : И⟩ ::Е
8. кончить работу

Диамический вектор отличается от обычного только тем, что

а) иидекс у него меняется всегда от 1 до числа элементов в векторе;

б) число элементов в векторе может динамически (т. е. в процессе работы) изменяться.

После «начать работу», как и после «сделать вектор пустым», динамический вектор пуст, т. е. не содержит ни одного элемента. Предписания «добавить» и «удалить» добавляют или удаляют элемент в конце вектора (т. е. справа, если рисовать вектор в порядке возрастания индексов).

Структуры как способы конструироваиия исполнителей. Перечисленные выше структуры являются не конкретными исполнителями, а способами их конструирования. Пользуясь этими способами, можно сконструировать, например, исполнителей «стек символов», «стек целых чисел», «стек элементов типа да/нет» и т. п. Такое конструирование записывается в виде

исполнитель А : стек *элементов типа* S
исполнитель В : стек *элементов типа* Z

Курсивом показаны части, которые можно опускать. Например, можно написать «**исполнитель** A: **стек** S».

Использование сконструированных таким образом исполнителей ничем не отличается от использования любых других исполнителей. Например, мы можем написать

A. добавить ⟨**вх**: "x"⟩ в стек
A. вершина := " "
B. вершина := B. вершина + 1
если B. стек пуст **то** ...

Структуры как способы конструирования объектов. Описанные выше структуры можно рассматривать и как способы конструирования объектов (множество состояний и операции задаются семантикой и системой предписаний соответствующей структуры). Например, можно написать

объекты:
C : **стек** ***элементов типа*** R
O : **очередь** ***элементов типа*** Z+
R3 : **вектор** ***элементов типа*** R ***с индексом*** (1 .. 3)
M : **множество** ***элементов типа*** Z+
MY : **множество** ***элементов типа*** Z+ ***с*** **нагрузкой** ***типа*** R

Курсивом показаны части, которые можно опускать, т. е. можно, например, писать «R3 : **вектор** R(1 .. 3)».

Использование сконструированных таким образом объектов практически ничем не отличается от использования соответствующих исполнителей. Мы также можем писать

C. вершина := C. вершина + 1
если C. стек пуст **то** ...
R3 (1) := 3.14

и т. п.

Разница состоит лишь в том, что исполнители глобальны — они существуют все время от начала до конца работы с исполнителем и их можно использовать повсюду в программе, а объекты локализованы внутри той программы или исполнителя X, где они описаны. Локальность объектов, как мы знаем, надо понимать двояко:

а) по тексту — их нельзя использовать вне X;

б) по времени существования — они не существуют вне периода от начала до конца работы X.

Именно этим и отличаются объекты от исполнителей. Вместо любого объекта A типа T можно использовать исполнителя **A типа** T, и при этом разница будет только в вопросах локализации.

Замечание. Структуры как способы конструирования объектов и исполнителей применимы не только к предопределенным, но

в к сконструированным объектам и типам. Можно, например, сконструировать стек, элементами которого являются последовательности символов, или даже рассмотреть множество таких стеков. Однако в этой книге такие «структуры структур» использоваться не будут — мы ограничимся лишь структурами, состоящими из объектов простых типов.

Несколько примеров программ.

```
программа инвертирование последовательности ⟨вх/вых: П⟩
. дано      : П : последовательность элементов типа Е
. получить  : | в П порядок элементов изменен на противоположный
. объекты   :
.   С : стек элементов типа Е
. --------
.   С. начать работу
.   П. встать в начало последовательности
.   цикл пока П. есть непрочитанные элементы выполнять
.   .   П. прочесть очередной элемент в ⟨вых: е⟩
.   .   С. добавить элемент ⟨вх: е⟩ в стек
.   конец цикла
.   П. сделать последовательность пустой
.   цикл пока С. стек не пуст выполнять
.   .   С. взять элемент из стека в ⟨вых: е⟩
.   .   П. добавить элемент ⟨вх: е⟩ в конец последовательности
.   конец цикла
.   С. кончить работу
конец программы

программа сжатие пробелов ⟨вх/вых: С⟩
. дано      : С : Л2-список элементов типа символ
. получить  : | в С каждая группа подряд идущих пробелов заме-
.             | нена на один пробел
. константы:
.  пробел = " "
. --------
.  если С. список пуст то ничего не делать иначе
.  .   С. установить указатель в начало списка
.  .   С. передвинуть указатель вперед
.  .   цикл пока С. указатель не в конце списка выполнять
.  .   .   если С. элемент списка до указателя = пробел и
.  .   .   .       С. элемент списка за указателем = пробел
.  .   .   то
.  .   .   .   С. удалить элемент списка за указателем
.  .   .   иначе
```

```
. . . . С. передвинуть указатель вперед
. . . конец если
. . конец цикла
. конец если
конец программы
```

```
программа корректировка ⟨вх/вых: П, вх: x : R⟩
. дано : П :  последовательность элементов типа R,
. получить : | в последовательности П все элементы, меньшие x,
.            | заменены на x
. --------
. П. встать в начало последовательности
. цикл нчв: | все меньшие x элементы прочитанной части П
. .         | заменены на x
. . пока П. есть непрочитанные элементы
. . выполнять
. .  если П. очередной элемент < x то
. .  .    П. очередной элемент := x
. .  конец если
. .  П. пропустить очередной элемент
. конец цикла
конец программы
```

```
программа в последовательности ⟨вх: П⟩ есть ⟨вх:e0⟩ : да/нет
. дано     : П : последовательность элементов типа E,
.            e0 : E
. получить: | ответ = (e0 ∈ П)
. --------
. П. встать в начало последовательности
. ответ := нет
. цикл пока П. есть непрочитанные элементы и ответ = нет
. . выполнять
. .  ответ := (П. очередной элемент = e0)
. .  П. пропустить очередной элемент
. конец цикла
конец программы
```

Циклы для каждого. При работе со структурами часто возникает необходимость перебрать все элементы структуры, или перебрать их до наступления какого-то условия. Для организации таких переборов в используемом нами языке программирования есть специальные конструкции — циклы **для каждого**, о которых мы упоминали в разд. 2. Например, приведенную выше программу «инвертирование последовательности» с помощью циклов **для каждого** можно записать так:

```
программа инвертирование последовательности (вх/вых: П)
. дано      : П: последовательность элементов типа Е
. получить  : | в П порядок элементов изменен на противоположный
. объекты   :
.   С : стек элементов типа Е
. --------
.   С. начать работу
.   П. встать в начало последовательности
.   цикл ∀ e ∈ непрочитанной части П выполнять
.   .   С. добавить элемент (вх: e) в стек
.   конец цикла
.   П. сделать последовательность пустой
.   цикл ∀e ∈ стека С выполнять
.   .   П. добавить элемент (вх: e) в конец последовательности
.   конец цикла
.   С. кончить работу
конец программы
```

Общая форма циклов для **каждого** и эквивалентные им фрагменты программ с использованием циклов **пока** приведены в табл. 7.1.

Полужирным курсивом выделены части, которые можно опускать. Обратите внимание, что:

а) перебор элементов происходит в соответствии со структурой, т. е. в последовательности перебираются только непрочитанные элементы; из стека, очереди и дека элементы при переборе изымаются. при переборе элементов множества их порядок неопределен и т. п.;

б) для векторов и матриц нет циклов, перебирающих элементы, — при необходимости надо перебирать индексы (**«для каждого И из** Имин..Имакс»);

в) в теле цикла **«для каждого** e **из** X» e является обозначением для доступного элемента структуры X (например, для очередного элемента последовательности, элемента списка до или за указателем, вершины стека и пр.).

Программу «корректировка» можно, например, записать так:

```
программа корректировка (вх/вых: П, вх : x: R)
. дано      :  П : последовательность элементов типа R
. получить  : | в последовательности П все элементы,
              | меньшие x, заменены на x
. --------
.   П. встать в начало последовательности
.   цикл ∀e ∈ непрочитанной части П выполнять
.     если e < x то e := x конец если
.   конец цикла
конец программы
```

Таблица 7.1

```
цикл ∀e∈стека X выполнять действие(e) конец цикла ==
    цикл пока X. стек не пуст выполнять
    . действие (X. вершина)
    . X.удалить вершину стека
    конец цикла
```

```
цикл ∀e∈очереди X выполнять действие(e) конец цикла ==
    цикл пока X.очередь не пуста выполнять
    . действие (X.начало)
    . X.удалить начало очереди
    конец цикла
```

```
цикл ∀e∈начала дека X выполнять действие(e) конец цикла ==
    цикл пока X.дек не пуст выполнять
    . действие (X.начало)
    . X.удалить начало дека
    конец цикла
```

```
цикл ∀e∈конца дека X выполнять действие(e) конец цикла ==
    цикл пока X.дек не пуст выполнять
    . действие (X.конец)
    . X.удалить конец дека
    конец цикла
```

```
цикл ∀e∈непрочитанной части последовательности X
выполнять действие(e) конец цикла ==
    цикл пока X.есть непрочитанные элементы выполнять
    . действие (X.очередной элемент)
    . X.пропустить очередной элемент
    конец цикла
```

```
цикл ∀e∈части списка X за указателем
выполнять действие(e) конец цикла ==
    цикл пока X.указатель не в конце списка выполнять
    . действие (X.элемент списка за указателем)
    . X.передвинуть указатель списка вперед
    конец цикла
```

```
цикл ∀e∈части списка X до указателя
выполнять действие(e) конец цикла ==
    цикл пока X.указатель не в начале списка выполнять
    . действие (X.элемент списка до указателя)
    . X.передвинуть указатель списка назад
    конец цикла
```

```
цикл ∀e∈множества X выполнять действие(e) конец цикла ==
    Y:=X | Y—это еще одно, вспомогательное множество,
         | которое нужно только для объяснения смысла
         | цикла ∀e∈множества.
    цикл пока Y.множество не пусто выполнять
    . Y.взять какой нибудь элемент в (выход)
    . действие (e)
    конец цикла
```

Таблица 7.2

цикл ∀e∈*стека* X **пока** *условие*(e)
выполнять *действие*(e) **конец цикла** ==
цикл пока X.стек **не пуст и** *условие* (X.вершина) **выполнять**
. *действие* (X.вершина)
. X.удалить вершину стека
конец цикла

цикл ∀e∈*очереди* X **пока** *условие*(e)
выполнять *действие*(e) **конец цикла** ==
цикл пока X.очередь **не пуста и** *условие*(e) **выполнять**
. *действие* (X.начало)
. X.удалить начало очереди
конец цикла

цикл ∀e∈**начала** *дека* X **пока** *условие*(e)
выполнять *действие*(e) **конец цикла** ==
цикл пока X.дек **не пуст и** *условие*(e) **выполнять**
. *действие* (X.начало)
. X.удалить начало дека
конец цикла

цикл ∀e∈**конца** *дека* X **пока** *условие*(e)
выполнять *действие*(e) **конец цикла** ==
цикл пока X.дек **не пуст и** *условие*(e) **выполнять**
. *действие* (X.конец)
. X.удалить конец дека
конец цикла

цикл ∀e∈*непрочитанной части последовательности* X
пока *условие*(e) **выполнять** *действие*(e) **конец цикла** ==
цикл пока X.есть непрочитанные элементы **и**
. *условие*(X.очередной элемент)
. **выполнять**
. *действие* (X.очередной элемент)
. X.пропустить очередной элемент
конец цикла

цикл ∀e ∈ *части списка* X **за** *указателем* **пока** *условие*(e)
выполнять *действие*(e) **конец цикла** ==
цикл пока X.указатель **не в конце** списка **и**
. *условие*(X.элемент списка за указателем)
. **выполнять**
. *действие* (X.элемент списка за указателем)
. X.передвинуть указатель списка вперед
конец цикла

Таблица 7.2 (*продолжение*)

```
цикл ∀e∈части списка X до указателя пока условие(e)
выполнять действие(e) конец цикла ==
      цикл пока X.указатель не в начале списка и
      .          условие(X.элемент списка до указателя)
      . выполнять
      . действие (X.элемент списка до указателя)
      . X.передвинуть указатель списка назад
      конец цикла
```

```
цикл ∀e∈множества X пока условие(e)
выполнять действие(e) конец цикла ==
      Y:=X | Y—это еще одно, вспомогательное множество,
           | которое нужно только для объяснения смысла
           | цикла ∀e∈множества.
      цикл пока Y.множество не пусто выполнять
      . Y.взять какой нибудь элемент в (вых:e)
      . если не условие(e) то выход из цикла конец если
      . действие(e)
      конец цикла
```

Для всех структур существуют также циклы **для каждого — пока**, перебирающие элементы лишь до тех пор, пока истинно некоторое условие (табл. 7.2).

Например, программу анализа наличия элемента e0 в последовательности с использованием цикла **«для каждого — пока»** можно записать так:

```
программа в последовательности (вх: П) есть (вх: e0) : да/нет
. дано      : П : последовательность элементов типа E,
.             e0 : E
. получить: | ответ = (e0 ∈ П)
. --------
. П. встать в начало последовательности
. ответ := нет
. цикл ∀e ∈ непрочитанной части П пока ответ = нет
. . выполнять
. .   ответ := (x = e0)
. конец цикла
конец программы
```

ЗАДАЧИ И УПРАЖНЕНИЯ

1. Напишите программу, которая определяет принадлежность точки (вх : t : R) отрезку (вх : A), где A: запись (начало, конец: R).

Во всех следующих задачах разрешается использовать только

те объекты-структуры, которые фигурируют в условии задачи. Кроме структур разрешается использовать произвольное количество объектов простых типов (не структур). Итак, напишите следующие программы:

2. программа удалить отрицательные элементы очереди ⟨вх/вых: О
   . дано : О : очередь элементов типа R
   . получить: | из очереди О удалены отрицательные элементы
   . объекты : О1 : очередь элементов типа R
   . --------

3. программа в деке ⟨вх: Д⟩ слово перевертыш : да/нет
   . дано : Д : дек символов
   . получить: | ответ = (в деке было слово-перевертыш),
   . Д. дек пуст
   . --------

4. программа пересечение множеств ⟨вх: А, В, вых: С⟩
   . дано : А, В : множество элементов типа Е
   . получить: С : множество элементов типа Е | $C = A \cap B$
   . --------

5. программа объединение множеств ⟨вх: А, В, вых: С⟩
   . дано : А, В : множество элементов типа Е
   . получить: С : множество элементов типа Е | $C = A \cup B$
   . --------

6. программа разность множеств ⟨вх: А, В, вых: С⟩
   . дано : А, В : множество элементов типа Е
   . получить: С : множество элементов типа Е | $C = A \setminus B$
   . --------

7. программа симметрическая разность множеств ⟨вх: А, В, вых:С⟩
   . дано : А, В : множество элементов типа Е
   . получить: С : множество элементов типа Е | $C = A \Delta B$
   . | $A \Delta B = (A \cup B) \setminus (A \cap B) = (A \setminus B) \cup (B \setminus A)$
   . --------

8. программа увеличить третий элемент ⟨вх/вых: П⟩ на единицу
   . дано : П : последовательность элементов типа R
   . получить: | в П третий элемент увеличен на 1
   . --------

9. программа слияние упорядоченных ⟨вх: П1, П2⟩ в ⟨вых: П3⟩
   . дано : П1, П2 : последовательность элементов типа Е
   . | последовательности П1 и П2 не убывают
   . получить: П3 : последовательность элементов типа Е
   . | П3 состоит из элементов П1 и П2 (с учетом
   . | кратности) и не убывает
   . --------

10. программа сумма элементов последовательности ⟨**вх:** П⟩ : **R**
. **дано** : П : **последовательность элемеитов типа R**
. **получить:** | **ответ** = сумма элементов последовательности П
. ---

11. программа произведение элементов последовательности ⟨**вх:** П⟩: **R**
. **дано** : П : **последовательность элементов типа R**
. **получить:** | **ответ** = произведение элементов П
. ---

12. программа удалить отрицательные элементы списка ⟨**вх/вых:** С⟩
. **дано** : С : **Л1-список элементов типа R**
. **получить:** | из списка С удалены все отрицательные элементы
. ---

13. программа число слов в списке ⟨**вх:** С⟩ : **Z** +
. **дано** : С : **Л2-список элементов типа символ**
. **получить:** | **ответ** = число слов в списке С.
. | Словом называется максимальная группа подряд
. | идущих символов, не содержащая пробелов
. ---

14. программа сумма векторов ⟨**вх:** a,b,**вых:**c⟩
. **дано** : a,b: **вектор R**(1..максинд)
. **получить:** c : **вектор R**(1..максинд) | c = a + b
. ---

15. программа скалярное произведение векторов ⟨**вх:** a,b⟩ : **R**
. **дано** : a,b : **вектор R**(1..максинд)
. **получить:** | **ответ** = скалярное произведение (a,b)
. ---

16. программа домножить матрицу ⟨**вх/вых:** a⟩ на число ⟨**вх:** x: **R**⟩
. **дано** : a : **матрица R**(1..n,1..m)
. **получить:** | a : = x * a
. ---

17. программа произведение матриц ⟨**вх:** a,b,**вых:** c⟩
. **дано** : a : **матрица R**(1..n,1..k),
. b : **матрица R**(1..k,1..m)
. **получить:** c : **матрица R**(1..n,1..m) | c = a * b
. ---

8. Индуктивное вычисление функций на пространстве последовательностей

Этот раздел состоит из двух частей — теоретической и практической. Для применения материала раздела на практике из теоретической части нужны только формулировки. Доказательства приведены для полноты картины и могут быть опущены.

Математика. Обозначения. Пусть

X — произвольный *алфавит*, т. е. какое-то конечное или бесконечное множество, например множество букв русского языка или множество действительных чисел;

$\Omega(X)$ или просто Ω — пространство всех конечных последовательностей над X, т. е. $\Omega = \{x_1x_2\ldots x_n\colon n \in Z+, x_i \in X\ \forall i \in 1..n\}$;

Δ — пустая последовательность, не содержащая ни одного элемента ($\Delta \in \Omega$);

$\Omega k(X)$, или просто Ωk, — пространство конечных последовательностей длины не менее k, т. е. $\Omega k = \{x_1x_2\ldots x_n\colon n \geqslant k, x_i \in X\}$; таким образом,

$\Omega = \Omega 0 \supset \Omega 1 \supset \Omega 2 \supset \ldots$

$*\colon \Omega * X \to \Omega$ — операция добавления (дописывания) элемента в конец последовательности, т. е. $x_1x_2\ldots x_n * x = x_1x_2\ldots x_nx$, $\Delta * x = x$ и т. п.; операция $*$ естественно сужается на $\Omega k\ \forall k$.

Кроме того, будем пользоваться стандартными математическими обозначениями:

Id_M — тождественное отображение произвольного множества M в себя: $Id_M(x) \equiv x\ \forall x \in M$,

N — множество натуральных чисел,

$Z+$ — множество неотрицательных целых чисел,

Z — множество целых чисел,

$R+$ — множество неотрицательных действительных чисел,

R — множество действительных чисел.

Через $\overline{N}$, $\overline{Z}+$ и т. д. будем обозначать множества, которые содержат кроме обычных еще и два идеальных элемента: $+\infty$ и $-\infty$.

Индуктивные функции.

Определение. Функция $F\colon \Omega \to Y$ называется *индуктивной*, если $F(\omega * x)$ можно вычислить, зная $F(\omega)$ и x, т. е. если

$$\exists G\colon Y * X \to Y : \forall \omega \in \Omega,\ x \in X\ F(\omega * x) = G(F(\omega), x). \qquad (1)$$

Примером индуктивной функции может служить функция $F =$ «число элементов последовательности», $F\colon \Omega \to Z+$. Действительно, $\forall \omega \in \Omega,\ x \in X\quad F(\omega * x) = F(\omega) + 1$, т. е. $\exists G\colon Z+ * X \to Z+$, а именно $G(y, x) = y + 1$, удовлетворяющее (1).

Индуктивные функции удобно вычислять. Если известно значение F_0 функции F на пустой последовательности $\omega_0 = \Delta$, то значение F_n функции F на последовательности $\omega_n = x_1x_2\ldots x_n$ можно найти следующим образом:

$$\begin{aligned} &\omega_1 = \omega_0 * x_1, && F_1 = F(\omega_1) = G(F_0, x_1),\\ &\omega_2 = \omega_1 * x_2, && F_2 = F(\omega_2) = G(F_1, x_2),\\ &\ldots\\ &\omega_n = \omega_{n-1} * x_n, && F_n = F(\omega_n) = G(F_{n-1}, x_n). \end{aligned} \qquad (2)$$

Если математическая последовательность $\omega \in \Omega$ является состоянием программистской последовательности P элементов типа X, то соответствующая (2) программа запишется в виде:

P. встать в начало
$F := F0$
цикл $\forall x \in$ **непрочитанной части** P **выполнять**
. $F := G(F, x)$ (2′)
конец цикла

Заметьте, что индексы в математическом описании (2) схемы вычисления F, т. е. значки x_i и F_i, относятся к этапам вычислений и обозначают *состояния объектов* x и F в (2′) в различные моменты времени. При программной реализации эти индексы исчезают.

Легко проследить аналогию между схемами (2)—(2′) и методом математической индукции. Роль базы индукции играет начальное значение функции F_0, а роль шага индукции — шаг вычислений, т. е. G. Именно поэтому удовлетворяющие (1) функции называются индуктивными.

Определение. *Действием* алфавита X на множестве Y называется правило, сопоставляющее каждому символу $x \in X$ отображение $Y \to Y$, т. е. отображение $G: Y*X \to Y$.

Вместо $G(y, x)$ мы будем часто писать $y \circ x$, подчеркивая аналогию с операцией $*$ — действием алфавита X на Ω.

Таким образом, определение индуктивной функции можно переформулировать так: функция $F: \Omega \to Y$ называется индуктивной, если существует действие $\circ: Y*X \to Y$ такое, что диаграмма

$$\begin{array}{ccc} \Omega * X & \xrightarrow{*} & \Omega \\ {\scriptstyle F*\mathrm{Id}X}\downarrow & & \downarrow{\scriptstyle F} \\ Y * X & \xrightarrow{\circ} & Y \end{array}$$

коммутативна, т. е. $\forall \omega \in \Omega,\ x \in X \quad F(\omega * x) = F(\omega) \circ x$. Допуская вольность речи, можно сказать, что F индуктивна, если она является гомоморфизмом $\Omega \to Y$, согласованным с действием X.

Замечание. В этом определении требуется существование какого-нибудь действия. Однако если F индуктивна, то действие X на $F(\Omega)$ определено однозначно, а на $Y \setminus F(\Omega)$ оно не существенно.

Утверждение (критерий индуктивности). F индуктивна $\Leftrightarrow$

$\forall a, b \in \Omega: F(a) = F(b)$, $\forall x \in X$ выполнено $F(a * x) = F(b * x)$

т. е. F индуктивна тогда и только тогда, когда из равенства значений F на последовательностях a и b следует равенство и на любых «одинаково удлиненных» последовательностях $a*x$ и $b*x$.

Доказательство. То, что это условие является необходимым ($\Rightarrow$), очевидно следует прямо из определения индуктивной функции. Докажем достаточность ($\Leftarrow$). Для этого определим действие X на $F(\Omega)$ формулой $y \circ x = F(a * x)$, где a — любой элемент множества $F^{-1}(y)$. Это определение корректно, так как если b — другой элемент множества $F^{-1}(y)$, то $F(a) = F(b)$ и, значит, $F(a * x) = F(b * x)$. Для $y \in Y \setminus F(\Omega)$ определим действие X как-нибудь, например формулой $y \circ x = y$ $\forall x \in X$. Очевидно, что для так определенного действия $\circ$ $\forall \omega \in \Omega$, $x \in X$ $F(\omega * x) = F(\omega) \circ x$. ■

На практике для того, чтобы выяснить, является ли F индуктивной, разумно действовать по определению: рассмотреть $F(\omega * x)$ и попытаться выразить эту величину через $F(\omega)$. Если это удалось и в полученных выражениях фигурируют лишь $F(\omega)$, x и константы, то F индуктивна и, более того, найденные выражения и есть G. Если же эти выражения содержат что-то еще, то обычно следует вывод, что информации, заключенной в $F(\omega)$ и x, недостаточно для вычисления $F(\omega * x)$, т. е. F не является индуктивной. Критерий индуктивности позволяет доказать это строго.

Рассмотрим, например, функцию f = «число максимальных элементов последовательности», $f: \Omega(R) \to Z+$, $f(\Delta) = 0$. Выразим $f(\omega * x)$:

$$f(\omega * x) = \begin{cases} 1, & \text{если } x > \text{максимального элемента } \omega, \\ f(\omega), & \text{если } x < \text{максимального элемента } \omega, \\ f(\omega) + 1, & \text{если } x = \text{максимальному элементу } \omega. \end{cases}$$

В правой части фигурируют $f(\omega)$, x и «максимальный элемент ω». Таким образом, можно сделать вывод, что f неиндуктивна — информации, заключенной в $f(\omega)$ и x, недостаточно для вычисления $f(\omega * x)$. Хотя в данном случае и очевидно, что максимальный элемент ω не может быть найден по $f(\omega)$, т. е. по числу максимальных элементов в ω, можно доказать неиндуктивность f и строго: $\exists a, b \in \Omega$, $x \in X$, а именно $a = 1$, $b = 2$, $x = 2$ такие, что $f(a) = f(b) = 1$, но $f(a * x) = 1$, а $f(b * x) = 2$, т. е. $f(a * x) \neq f(b * x)$. Следовательно, по критерию индуктивности f не является индуктивной.

Стационарные значения. Пусть $F: \Omega \to Y$ — индуктивная функция и $\circ: Y * X \to Y$ — соответствующее ей действие X на Y. Значение $s \in Y$ называется *стационарным*, если

$$\forall x \in X \quad s \circ x = s.$$

Если при вычислении $F(\omega)$ какой-нибудь индуктивной функции обнаруживается, что значение F_1 стационарно, то вычисления можно прервать, ибо $F(\omega) = F_1$. Схему вычисления значения индуктивной функции (2′) с учетом стационарных значений можно переписать в виде:

Р.встать в начало
F := **F0**

цикл $\forall x \in$ **непрочитаниой части** P
. **пока** ⟨F⟩ не стационарно
. **выполнять**
. F := G (F, x)
конец цикла

Пример. Пусть

X — произвольный алфавит из двух или более символов, $x0 \in X$,
Y = {да, нет},
F = «все элементы последовательности равны x0», $F: \Omega \to Y$.

Определим $\circ$ (действие X на Y) следующим образом:

да $\circ$ x0 = да,
да $\circ$ x = нет, для любого $x \neq x0$,
нет $\circ$ x = нет, для любого x.

Легко видеть, что $F(\omega * x) = F(\omega) \circ x$, т. е. F индуктивна, а значение нет $\in$ Y является стационарным. Таким образом, вычисление $F(\omega)$ можно записать так:

F := да
цикл $\forall x \in$ **непрочитаниой части** P
. **пока** $F \neq$ нет
. **выполнять**
. $F := (x \neq x0)$
конец цикла

Обычно программа, учитывающая наличие стационарных значений, более понятна и проста, чем программа, перебирающая элементы последовательности до конца даже после того, как ответ уже не может измениться. Кроме того, учет стационарных значений в пока обычно позволяет упростить тело цикла (что и сделано в примере выше). Разумеется, в целях оптимизации можно игнорировать существование стационарных значений.

Индуктивные расширения функций. Многие функции на пространстве последовательностей не являются индуктивными. Это означает, что информации, заключенной в $F(\omega)$ и x, недостаточно для вычисления $F(\omega * x)$. Можно, однако, посмотреть, какой именно информации нам не хватает, и рассмотреть более сложную функцию, включив в нее и эту информацию тоже.

Определение. Функция $F: \Omega \to Y$ называется *индуктивным расширением* функции $f: \Omega \to Y_f$, если

а) F индуктивна,
б) $\exists \pi: Y \to Y_f$ такое, что $\forall \omega \in \Omega \quad f(\omega) = \pi(F(\omega))$.

Пример. Рассмотрим функцию f = «число максимальных элементов последовательности». Мы видели, что эта функция не

является индуктивной — для вычисления $f(\omega * x)$ надо знать максимальный элемент ω. Рассмотрим новую функцию:

$$F : \Omega \to Z + *\overline{R},\ F(\omega) = (f(\omega), \max(\omega)),$$

где max: $\Omega \to \overline{R}$ — функция, сопоставляющая каждой последовательности максимум ее элементов, $\max(\Delta) = -\infty$.

Зная $f(\omega)$, $\max(\omega)$ и x, можно найти и $f(\omega * x)$, и $\max(\omega * x)$, т. е. F — индуктивная функция. Очевидно также, что $\forall \omega \in \Omega$, зная $F(\omega)$, можно найти $f(\omega)$ — достаточно просто «забыть» $\max(\omega)$. Таким образом, F — индуктивное расширение f, и f можно вычислить так:

```
Р.встать в начало
y := 0; m := — ∞
цикл ∀x ∈ непрочитанной части Р выполнять
.  выбор
.  . при x < m ⇒ ничего не делать
.  . при  x = m ⇒ y := y + 1
.  . при  x > m ⇒ y := 1; m := x
.  конец выбора
конец цикла
ответ := y
```

Минимальное индуктивное расширение. Для одной и той же функции f можно придумать разные индуктивные расширения. Например, тождественное отображение Id_Ω: $\Omega \to \Omega$ является индуктивным расширением для любой функции f. Для этого расширения $\pi \equiv f$, т. е. задача получения $f(\omega)$ по $F(\omega)$ есть в точности исходная задача. Таким образом, это индуктивное расширение абсолютно бесполезно.

Наибольший практический интерес представляет такое индуктивное расширение, *которое содержит минимум информации об* ω. Мы покажем сейчас, что такое минимальное расширение существует и что оно единственно с точностью до изоморфизма.

Определение. Пусть F: $\Omega \to Y$ и $\hat{F}$: $\Omega \to \hat{Y}$ — две функции на Ω. Скажем, что $F \geqslant \hat{F}$, если $\exists p$: $Y \to \hat{Y}$ такое, что $\forall \omega \in \Omega$ $\hat{F}(\omega) = p(F(\omega))$, т. е. если $\hat{F}(\omega)$ можно найти по $F(\omega)$.

Определение. Индуктивное расширение $\hat{F}$: $\Omega \to \hat{Y}$ функции f называется *минимальным*, если

а) $\hat{F}(\Omega) = \hat{Y}$,

б) для любого другого индуктивного расширения F функции f $F \geqslant \hat{F}$.

Утверждение. Минимальное расширение единственно с точностью до изоморфизма.

Доказательство. Пусть $F_1\colon \Omega\to Y_1$ и $F_2\colon \Omega\to Y_2$ — два минимальных индуктивных расширения f. Тогда

$$F_1(\Omega)=Y_1,\quad F_2(\Omega)=Y_2,\quad F_1\geqslant F_2,\quad F_2\geqslant F_1.$$

Так как $F_1\geqslant F_2$ и $F_2\geqslant F_1$, то существуют p_{12} и p_{21} такие, что

$$\forall\omega\in\Omega\ F_2(\omega)=p_{21}(F_1(\omega))\ \text{и}\ F_1(\omega)=p_{12}(F_2(\omega)).$$

Докажем, что $p_{12}\circ p_{21}=\mathrm{Id}_{Y_1}$. Возьмем любое $y_1\in Y_1$. Так как $F_1(\Omega)=Y_1$, то существует $\omega\in\Omega$ такое, что $F_1(\omega)=y_1$. Так как $\forall\omega\in\Omega\ F_1(\omega)=p_{12}(p_{21}(F_1(\omega)))$, то $p_{12}\circ p_{21}(y_1)=y_1$. Поскольку y_1 любое, то $p_{12}\circ p_{21}=\mathrm{Id}_{Y_1}$. Аналогично доказывается, что $p_{21}\circ p_{12}=\mathrm{Id}_{Y_2}$, и, значит, p_{12} и p_{21} — изоморфизмы. ■

Теорема. Минимальное расширение существует.

Доказательство. Доказательство этой теоремы состоит из двух этапов. Сначала строится некоторое вполне конкретное индуктивное расширение (назовем его *каноническим*), а затем доказывается его минимальность.

Каноническое индуктивное расширение функции f. Обозначим через $\omega_1*\omega_2$ результат добавления (приписывания) последовательности $\omega_2\in\Omega$ в конец последовательности $\omega_1\in\Omega$ (для одноэлементной последовательности ω_2 это согласуется со старым смыслом знака $*$). Введем на Ω бинарное отношение $\approx$ следующим образом:

$$a\approx b\iff\forall\omega\in\Omega\ f(a*\omega)=f(b*\omega).$$

Ясно, что

1) $\approx$ есть отношение эквивалентности на Ω,

2) $a\approx b\Rightarrow\forall x\in X\ a*x\approx b*x$,

3) $a\approx b\Rightarrow f(a)=f(b)$.

Рассмотрим фактор-множество $\widehat{Y}=\Omega/\approx$ по отношению эквивалентности $\approx$ и обозначим через $\widehat{F}\colon\Omega\to\widehat{Y}$ естественную проекцию Ω на фактор-множество. Тогда 2) можно переписать в виде

$$\forall a,b\in\Omega\colon\ \widehat{F}(a)=\widehat{F}(b)\quad\forall x\in X\quad\widehat{F}(a*x)=\widehat{F}(b*x),$$

т. е. для $\widehat{F}$ выполняется критерий индуктивности.

Из 3) вытекает, что f постоянна на множествах уровня функции $\widehat{F}$ и, следовательно, можно определить проекцию $\widehat{\pi}\colon\widehat{Y}\to Y_f$ формулой $\widehat{\pi}(\widehat{\omega})=f(\omega)$, где ω — какой-нибудь представитель класса эквивалентности $\widehat{\omega}$. Наконец, из определений $\widehat{F}$ и $\widehat{\pi}$ вытекает, что $f=\widehat{\pi}\circ\widehat{F}$.

Таким образом, $\widehat{F}$ — индуктивное расширение f и $\widehat{F}(\Omega)=\widehat{Y}$.

Докажем теперь минимальность $\widehat{F}$, т. е. покажем, что для любого $F\colon\Omega\to Y$ индуктивного расширения f существует $p\colon Y\to\widehat{Y}$ такое, что $\forall a\in\Omega\ \widehat{F}(a)=p\circ F(a)$. Так как $\widehat{F}(a)$ — это класс эквивалентности a, то нам надо показать, что по значению $F(a)$ класс эквивалентности a определяется однозначно. Действительно, пусть $F(b)=F(a)$ $(b\in\Omega)$. В силу индуктивности F $\forall\omega\in\Omega\ F(a*\omega)=F(b*\omega)$, а так как F — расширение f, то и $f(a*\omega)=f(b*\omega)$, т. е. $a\approx b$. Таким образом, для элементов a и b из разных классов эквивалентности $F(a)\neq F(b)$ и по значению $F(a)$ класс эквивалентности a (т. е. $\widehat{F}(a)$) определяется однозначно. ■

Утверждение (критерий минимальности индуктивного расширения). Индуктивное расширение $F\colon\Omega\to Y$ функции $f\colon\Omega\to Y_f$ является минимальным тогда и только тогда, когда

1) $F(\Omega)=Y$,

2) $\forall a,b\in\Omega\ F(a)\neq F(b)\Rightarrow a\not\approx b$, где $\not\approx$ — отношение эквивалентности, введенное выше $(a\not\approx b\iff\exists\omega\in\Omega\ f(a*\omega)\neq f(b*\omega))$.

Доказательство. Необходимость очевидна. Докажем достаточность. Рассмотрим каноническое минимальное расширение $\hat{F}_c: \Omega \to \hat{Y}$. Так как $F \geqslant \hat{F}$, то существует $p: Y \to \hat{Y}$ такое, что $\forall \omega \in \Omega\ \hat{F}(\omega) = p \circ F(\omega)$. Так как $F(\Omega) = Y$, а $\hat{F}(\Omega) = \hat{Y}$, то $p(Y) = \hat{Y}$, т. е. p является отображением «на». Для того чтобы доказать, что p — изоморфизм, осталось показать, что p не склеивает точек из $Y = F(\Omega)$. Действительно, рассмотрим две разные точки из $F(\Omega)$:

$$y_a, y_b \in F(\Omega),\quad y_a \neq y_b,\quad y_a = F(a),\quad y_b = F(b).$$

Так как $F(a) \neq F(b)$, то, согласно 2), $\hat{F}(a) \neq \hat{F}(b)$, т. е. $p(y_a) \neq p(y_b)$. ■

Пример использования критерия минимальности. Очевидно, что функция

$$F: \Omega \to \mathbf{Z} + * \overline{\mathbf{R}},\qquad F(\omega) = (f(\omega), \max(\omega))$$

не является минимальным индуктивным расширением функции f = = «число максимальных элементов последовательности», так как $F(\Omega) \neq \mathbf{Z} + * \overline{\mathbf{R}}$. Например, $\exists \omega \in \Omega: F(\omega) = (0, x)$ при $x \neq -\infty$.

Однако если ограничить область значений функции, т. е. рассматривать ее как отображение $F: \Omega \to Y = (0, -\infty) \cup \mathbf{N} * \mathbf{R}$, то такая функция будет минимальным индуктивным расширением f. Докажем это, пользуясь критерием минимальности.

1) $F(\Omega) = Y$. Действительно, $\forall (n, x) \in \mathbf{N} * \mathbf{R}\ \ \exists \omega = xx \ldots x$ (n элементов) такая, что $F(\omega) = (n, x)$, а $(0, -\infty) = F(\Delta)$.

2) Пусть $a, b \in \Omega$ и $F(a) \not\approx F(b)$. Покажем, что $a \not\approx b$. Пусть $F(a) = (n_a, x_a)$, $F(b) = (n_b, x_b)$. Если $n_a \neq n_b$, то это просто значит, что $f(a) \neq f(b)$ и, следовательно, $a \not\approx b$. Если же $n_a = n_b$, но $x_a \neq x_b$, то без ограничения общности можно считать, что $x_a < x_b$. Тогда $f(a * x_a) = n_a + 1$, $f(b * x_a) = n_a$, т. е. $f(a * x_a) \neq f(b * x_a)$ и, значит, $a \not\approx b$.

Разные пространства и доопределения. На практике типичны ситуации, когда сама функция f и ее *удобное* индуктивное расширение F определены на разных пространствах. Например, для функции f = «среднее арифметическое элементов последовательности», определенной только на $\Omega 1$, можно использовать индуктивное расширение $F = (\Sigma, n)$, где $\Sigma: \Omega \to \mathbf{R}$ — сумма элементов последовательности ($\Sigma(\Delta) = 0$), а $n: \Omega \to \mathbf{Z}+$ — число элементов последовательности. Таким образом, в этом случае f определена на $\Omega 1$, а F — на Ω. Если в уже рассматривавшемся примере f = «число максимальных элементов последовательности» не использовать идеальных элементов $\pm\infty$, то функция $F = (f, \max)$ будет определена лишь на $\Omega 1$, хотя f определена на Ω.

Естественно, такие расхождения надо учитывать при программной реализации. Например, нахождение числа максимальных элементов последовательности без использования идеальных элементов $\pm\infty$ пришлось бы записать так:

если P.пуста **то ответ** := 0 **иначе**
. P.встать в начало
. P.прочесть очередной элемент в ⟨**вых** : x⟩
. y := 1; m := x
. **цикл** ∀x ∈ **непрочитанной части** P **выполнять**
. . **выбор**
. . . **при** x < m ⇒ **ничего не делать**
. . . **при** x = m ⇒ y := y + 1
. . . **при** x > m ⇒ y := 1; m := x
. . **конец выбора**
. **конец цикла**
. **ответ** := y
конец если

Как видно из этого примера, иметь дело с индуктивными функциями на Ω проще, чем с функциями на Ω1. Поэтому, если индуктивная функция F определена на Ω1, разумно попытаться доопределить ее на Ω с сохранением G, т. е. попытаться найти такое y0 = = F(Δ), что

$$\forall x \in X \quad F(x) = F(\Delta * x) = G(F(\Delta), x) = G(y0, x), \quad \text{т. е.}$$
$$G(y0, x) = F(x).$$

Если при этом F рассматривается как индуктивное расширение f, то надо еще сохранить и π:

$$f(\Delta) = \pi(F(\Delta)) = \pi(y0).$$

Вернемся к примеру «число максимальных элементов последовательности». Функцию max: Ω1(R)→ $\overline{R}$ можно было доопределить на Ω, так как существует y0 = −∞ ∈ $\overline{R}$ такое, что

$$\forall x \in X \quad G(-\infty, x) = \max(-\infty, x) = x = F(x).$$

Аналогично индуктивную функцию F_* = «произведение элементов числовой последовательности» (F: Ω1(R)→ R, G(y, x) = y∗x) можно доопределить на Ω, так как существует y0 = 1 ∈ R такое, что

$$\forall x \in X \quad G(1, x) = 1 * x = x = F(x).$$

Наоборот, индуктивную функцию F = «первый элемент последовательности» (F: Ω1(X)→ X, G(y, x) = y) доопределить на Ω нельзя, так как не существует такого y0 ∈ X, что

$$\forall x \in X \quad G(y0, x) = F(x), \quad \text{т. е.} \quad y0 = x.$$

Другими словами, не существует y0 ∈ X, который был бы равен всем x ∈ X одновременно.

Наконец, если такого y0 ∈ Y для F: Ω1(X)→ Y и G: Y∗X→ Y найти не удается, то можно попытаться расширить пространство Y. Собственно говоря, именно такое расширение пространства R до $\overline{R}$

и было использовано в примере «число максимальных элементов последовательности».

Практика. На практике индуктивное вычисление функции на пространстве последовательностей сводится обычно к применению одного из двух приемов. Первый прием состоит в том, чтобы прямо представить неиндуктивную функцию в виде композиции индуктивных (например, среднеарифметическое можно представить в виде Σ/n, где Σ — сумма, а n — число элементов последовательности — две индуктивные функции). Если это удается, то в качестве индуктивного расширения можно взять объединение полученных индуктивных функций (для среднеарифметического — пару (Σ, n)). Разложение функции в композицию индуктивных, однако, творческая задача и потому применяется только в простейших случаях, где такое разложение более или менее очевидно.

Второй прием состоит в выражении $f(\omega*x)$ через $f(\omega)$ и x. После этого надо посмотреть, какой информации, кроме $f(\omega)$ и x, не хватает для вычисления $f(\omega*x)$. Пусть это информация $f1(\omega)$. Если $f1(\omega*x)$ выражается через $f(\omega)$, $f1(\omega)$ и x, то в качестве индуктивного расширения f можно взять пару $(f, f1)$. В противном случае недостающая информация обозначается $f2(\omega)$ и рассматривается $f2(\omega*x)$. И опять: если $f2(\omega*x)$ выражается через x и функции $f(\omega)$, $f1(\omega)$ и $f2(\omega)$, то тройка $(f, f1, f2)$ является индуктивным расширением f. В противном случае рассматривается недостающая информация $f3$ и т. д. до тех пор, пока не будет получено индуктивное расширение f. Второй прием, в отличие от первого, является регулярным и почти не требует творческих усилий. Ниже мы в основном будем пользоваться именно вторым приемом — добавлением к функции недостающей для индуктивности информации.

Рассмотрим несколько примеров индуктивного вычисления функций на пространстве последовательностей.

Номер первого элемента, равного x0. Пусть стоит задача определить номер первого элемента, равного x0, в последовательности P элементов типа X:

дано : P : **последовательность элементов типа X,**
x0: X
получить: | номер первого вхождения x0 в P или 0,
| если элемента, равного x0, в P нет

Попытаемся вычислить этот номер индуктивно:

$$f(\Delta) = 0, \quad f(\omega * x) = \begin{cases} f(\omega), & \text{если } x0 \in \omega \text{ или } x \neq x0, \\ n1 & \text{в противном случае,} \end{cases}$$

где $n1 = n + 1$ — номер элемента x (n — длина последовательности ω). Условие $x0 \in \omega$ легко выразимо через $f(\omega)$: $x0 \in \omega \Leftrightarrow$

$\Leftrightarrow f(\omega) \neq 0$. Поэтому $f(\omega * x)$ можно переписать в виде

$$f(\omega * x) = \begin{cases} f(\omega), & \text{если } f(\omega) \neq 0 \text{ или } x \neq x0, \\ n + 1, & \text{если } f(\omega) = 0 \text{ и } x = x0. \end{cases}$$

Таким образом, знания $f(\omega)$ и x недостаточно для вычисления $f(\omega * x)$ — нужно знать еще и n — длину ω. Рассмотрим более сложную функцию F, включив в нее и эту информацию:

$$F = (f, n), \quad F : \Omega \to \mathbf{Z} + * \mathbf{Z} +, \quad n(\omega) = \text{«число элементов } \omega\text{»}.$$

Очевидно, что F является индуктивным расширением f. Хотя это расширение и не минимально (докажите это), оно определено на всем Ω и легко вычислимо. Поэтому для практических целей его достаточно.

Наконец, заметим, что все значения $f \neq 0$ являются стационарными. Используя обозначения «ответ» и «число прочитанных» вместо f и n соответственно, можно записать программу в виде:

```
программа номер первого ⟨вх: x0⟩ в последовательности ⟨вх:P⟩ : Z+
. дано      : P : последовательность элементов типа X,
.               x0: X
. получить: | номер первого вхождения x0 в P или 0,
.           | если элемента, равного x0, в P нет
. --------
. P. встать в начало
. ответ := 0; число прочитанных := 0
. цикл ∀x ∈ непрочитанной части P
. . пока ответ = 0
. . выполнять
. . число прочитанных.увеличить на 1
. . если x = x0 то ответ := число прочитанных конец если
. конец цикла
конец программы
```

Среднеарифметическое.

```
дано      : P : последовательность элементов типа R,
            P.не пуста
получить: | среднеарифметическое элементов последовательности
```

Мы уже говорили о том, что в качестве индуктивного расширения для этой функции можно взять $F = (S, n)$, где $S: \Omega \to \mathbf{R}$ — сумма элементов последовательности, а $n: \Omega \to \mathbf{Z}+$ — число элементов. Соответствующая программа записывается в виде:

```
программа среднеарифметическое ⟨вх : P⟩ : R
. дано      : P : последовательность элементов типа R,
.               P.не пуста
```

```
. получить: | среднеарифметическое элементов последовательности
. ---------
. P.встать в начало
. сумма := 0; n := 0
. цикл ∀x ∈ непрочитанной части P выполнять
. .   сумма := сумма + x
. .   n     := n     + 1
. конец цикла
. ответ := сумма/n
конец программы
```

Представление среднеарифметического в виде композиции индуктивных функций, как уже отмечалось, является творческой задачей. Можно было идти и стандартным (вторым) путем, т. е. выражать $f(\omega * x)$ через $f(\omega)$ и x. Посмотрим, к чему приведет этот путь:

$$f(\omega * x) = (S(\omega) + x)/(n(\omega) + 1) = (f(\omega) * n(\omega) + x)/(n(\omega) + 1)$$

Так как в правой части кроме $f(\omega)$ и x используется $n(\omega)$, то

$$F = (f, n), \quad \text{где} \quad n(\omega) = \text{«число элементов } \omega\text{»}, \; n : \Omega \to \mathbf{Z}+.$$

Функция F в целом определена лишь на $\Omega 1$. Попробуем продолжить ее на Ω. Для этого надо найти такое y0, что

$$\forall x \in \mathbf{R} \quad x = f(x) = f(\Delta * x) = (f(\Delta) * n(\Delta) + x)/(n(\Delta) + 1) = (y0 * 0 + x)/1$$

или $\forall x \in \mathbf{R}\ x = y0 * 0 + x$. Но полученное равенство выполнено $\forall y0 \in \mathbf{R}$. Следовательно, можно доопределить $f(\Delta)$ чем угодно, например нулем. В результате получим программу:

```
программа среднеарифметическое ⟨вх : P⟩ : R
. дано     : P : последовательность элементов типа R,
.            P.не пуста
. получить: | среднеарифметическое элементов последовательности
. ---------
. P.встать в начало
. f := 0; n := 0
. цикл ∀x ∈ непрочитанной части P выполнять
. .   f := (f * n + x)/(n + 1)
. .   n := n + 1
. конец цикла
. ответ := f
конец программы
```

Дисперсия. Дисперсией элементов последовательности $\omega = x_1 x_2 \ldots x_n$ называется величина $\left(\sum (x_i - m)^2\right)/n$, где m — среднеарифметическое. Попробуем применить первый подход — предста-

вить дисперсию в виде композиции индуктивных функций:

$$f(\omega) = \frac{\sum^{n} (x_i - m)^2}{n} = \frac{\sum^{n} (x_i^2 - 2mx_i + m^2)}{n} =$$

$$= \frac{\sum^{n} x_i^2 - m \sum^{n} x_i - m \sum^{n} (x_i - m)}{n} =$$

$$= \frac{\sum^{n} x_i^2 - m \sum^{n} x_i}{n} = \frac{\sum^{n} x_i^2}{n} - \frac{\left(\sum^{n} x_i\right)^2}{n^2}.$$

Обозначим $s2 = \sum^{n} x_i^2$, $s1 = \sum^{n} x_i$, $s0 = \sum^{n} (1) = n$. Тогда $f(\omega) = s2/s0 - (s1/s0)^2$.

Очевидно, что каждая из трех функций s2, s1, s0 индуктивна, и, значит, F = (s2, s1, s0) является индуктивным расширением f:

```
программа дисперсия ⟨вх : P⟩ : R
. дано      : P : последовательность элементов типа R,
.             P.не пуста
. получить: | дисперсию элементов последовательности
. ---------
. P.встать в начало
. s2 := 0   | сумма квадратов элементов прочитанной части P
. s1 := 0   | сумма           элементов прочитанной части P
. s0 := 0   | число           элементов прочитанной части P
. цикл ∀x ∈ непрочитанной части P выполнять
. .  s2 := s2 + x * * 2
. .  s1 := s1 + x
. .  s0 := s0 + 1
. конец цикла
. ответ := s2/s0 — (s1/s0) ** 2
конец программы
```

Задачу можно было решать и стандартным (вторым) путем — выражать $f(\omega * x)$ через $f(\omega)$. В этом случае мы бы получили

$$f(\omega * x) = \frac{n}{n+1}\left(f(\omega) + \frac{(x-m)^2}{n+1}\right), \quad F = (f, m, n),$$

где m — среднеарифметическое, а n — число элементов последовательности.

Функция F индуктивна (докажите это), однако в целом определена на $\Omega 1$. Функция n определена на Ω; m, как мы знаем, можно доопределить на Ω произвольным образом (например, нулем).

Посмотрим, нельзя ли доопределить f:

$$f(x) = \frac{\sum (x - x)}{1} = 0 \quad \forall x \in R.$$

С другой стороны,

$$f(\Delta * x) = \frac{n(\Delta)}{n(\Delta)+1}\left(f(\Delta) + \frac{(x - m(\Delta))^2}{n(\Delta)+1}\right) = \frac{0}{1}(f(\Delta) + x^2) = 0.$$

Таким образом, $\forall y0 \in R\ f(x) = G(y0, x)$, и можно доопределить f на Δ произвольно, например положив $f(\Delta) = 0$.

```
программа дисперсия ⟨вх: P⟩: R
. дано      : P : последовательность элементов типа R,
.               P.не пуста
. получить: | дисперсию элементов последовательности
. --------
. P.встать в начало
. f := 0    | дисперсия прочитанной части P
. m := 0    | среднеарифметическое прочитанной части P
. n := 0    | число элементов прочитанной части P
. цикл ∀x ∈ непрочитанной части P выполнять
. . f := n/(n + 1) * (f + (x - m) ** 2/(n + 1))
. . m := (m * n + x)/(n + 1)
. . n := n + 1
. конец цикла
. ответ := f
конец программы
```

Поиск по образцу.

дано : P : последовательность элементов типа символ
получить: | число вхождений группы из 4 символов "abcd" в P

Действуем стандартным (вторым) путем: $f(\Delta) = 0$,

$$f(\omega * x) = \begin{cases} f(\omega) + 1, & \text{если } x = \text{"d"} \text{ и } \omega \text{ кончается на "abc",} \\ f(\omega) & \text{в противном случае.} \end{cases}$$

В правой части кроме $f(\omega)$ и x фигурирует $f1(\omega) =$ «ω кончается на "abc"». Таким образом, для вычисления $f(\omega*x)$ нужно знать $f1(\omega)$. Попробуем рассмотреть функцию $F1 = (f, f1)$. Легко видеть, что эта функция не является индуктивной: информации, заключенной в $f(\omega)$, $f1(\omega)$, и x хватает для вычисления $f(\omega*x)$, но недостаточно для вычисления $f1(\omega*x)$:

$$f1(\omega * x) = \begin{cases} \text{да}, & \text{если } x = \text{"c"} \text{ и } \omega \text{ кончается на "ab",} \\ \text{нет} & \text{в противном случае.} \end{cases}$$

Таким образом, для вычисления $f1(\omega*x)$ нужно знать $f2(\omega) =$ «ω кончается на "ab"». Рассмотрим $F2 = (f, f1, f2)$. Однако и эта функ-

ция не является индуктивной: заключенной в ней информации недостаточно для нахождения f2(ω*x):

$$f2(\omega * x) = \begin{cases} \text{да}, & \text{если } x = \text{"b"} \text{ и } \omega \text{ кончается на "a"}, \\ \text{нет} & \text{в противном случае.} \end{cases}$$

Рассмотрим f3(ω) = «ω кончается на "a"» и соответственно F3 = = (f, f1, f2, f3). Легко видеть, что F3 уже индуктивна:

$$f3(\omega * x) = \begin{cases} \text{да}, & \text{если } x = \text{"a"}, \\ \text{нет} & \text{в противном случае.} \end{cases}$$

Функция F3 является индуктивным расширением f, но она достаточно сложна. Поэтому попробуем с помощью критерия минимальности найти расширение попроще.

Сразу видно, что расширение F3 не минимально, так как f1, f2 и f3 не независимы: например, если f1(ω) = да, то заведомо f2(ω) = = нет и f3(ω) = нет. У тройки (f1, f2, f3) всего четыре допустимых состояния. Эти состояния можно представить одним числом в диапазоне от 0 до 3:

$$n(\omega) = \begin{cases} 3, & \text{если } \omega \text{ кончается на "abc"}, \\ 2, & \text{если } \omega \text{ кончается на "ab"}, \\ 1, & \text{если } \omega \text{ кончается на "a"}, \\ 0 & \text{в противном случае.} \end{cases}$$

Функция F = (f, n): Ω → Z+*{0 .. 3} индуктивна и, более того, является минимальным индуктивным расширением f (докажите это).

```
программа число вхождений abcd в (вх: P): Z +
. дано      : P : последовательность элементов типа символ
. получить: | число вхождений группы из 4 символов "abcd" в P
. --------
. P.встать в начало
. f := 0 | число найденных вхождений
. n := 0 | число найденных символов в незаконченном вхождении
. цикл ∀x ∈ непрочитанной части P выполнять
. .  выбор
. .  . при n = 3 и x = "d" ⇒ n := 0; f := f + 1
. .  . при n = 2 и x = "c" ⇒ n := 3
. .  . при n = 1 и x = "b" ⇒ n := 2
. .  . при         x = "a" ⇒ n := 1
. .  . иначе               ⇒ n := 0
. .  конец выбора
. конец цикла
. ответ := f
конец программы
```

Значение заданного по убывающим степеням многочлена в точке. Пусть заданы точка t0, последовательность P, в которой в порядке убывания степеней расположены коэффициенты многочлена, и требуется определить значение многочлена в точке t0. Обозначим состояние последовательности P через $\omega = a_n \ldots a_2 a_1 a_0$, где a_n — первый элемент последовательности, а a_0 — последний. Наша задача, таким образом, состоит в нахождении

$$f(\omega) = a_n t0^n + \ldots + a_2 t0^2 + a_1 t0 + a_0.$$

Будем действовать стандартно:

$$f(\omega * x) = a_n t0^{n+1} + \ldots + a_2 t0^3 + a_1 t0^2 + a_0 t0 + x = f(\omega) * t0 + x.$$

В правой части не используется ничего, кроме $f(\omega)$, x и не зависящей от последовательности величины t0, т. е. функция f индуктивна. Из соотношения $x = f(x) = f(\Delta * x) = f(\Delta) * t0 + x$ следует, что $f(\Delta) * t0 = 0$, т. е. $f(\Delta) = 0$.

```
программа значение многочлена ⟨вх : P⟩ в точке ⟨вх : t0⟩ : R
. дано      :  P : последовательность элементов типа R, t0 : R
. получить: | значение в точке t0 многочлена, коэффициенты кото-
.           | рого расположены в P в порядке убывания степеней
. --------
. P.встать в начало
. f := 0
. цикл ∀x ∈ непрочитанной части P выполнять
. .   f := f * t0 + x
. конец цикла
. ответ := f
конец программы
```

Обратите внимание, что мы регулярным образом, ничего не придумывая, получили схему Горнера вычисления значения многочлена в точке.

Производная многочлена, заданного по убывающим степеням. Пусть в условиях предыдущей задачи надо определить не значение многочлена, а значение его производной $f'(\omega)$. Мы могли бы стандартно выражать $f'(\omega * x)$, но того же результата можно достичь, просто продифференцировав $f(\omega * x)$ по t и взяв значение в t0:

$$f'(\omega * x) = (f(\omega) * t0 + x)' = f'(\omega) * t0 + f(\omega).$$

В правой части кроме $f'(\omega)$ и константы t0 фигурирует еще и $f(\omega)$. Рассмотрим $F = (f', f)$. Эта функция индуктивна, и ее можно продолжить на Ω, положив $f'(\Delta) = 0$:

```
программа производная многочлена ⟨вх : P⟩ в точке ⟨вх : t0⟩ : R
. дано      :  P : последовательность элементов типа R, t0 : R
```

```
. получить: | значение в точке t0 производной многочлена,
.           | коэффициенты которого расположены в P
.           | по убыванию степеней
. --------
. P.встать в начало
. df := 0
. f := 0
. цикл ∀x ∈ непрочитанной части P выполнять
. .  df := df * t0 + f
. .  f := f * t0 + x
. конец цикла
. ответ := df
конец программы
```

Полученную схему, по-видимому, разумно назвать схемой Горнера вычисления производной многочлена в точке.

Анализ правильности формулы. Пусть $X = \{(,), t, +\}$ — алфавит из четырех символов. Среди последовательностей в алфавите X некоторые являются правильными алгебраическими формулами, например t, (t), t + t, ((t + t) + t). Нужно определить, является ли последовательность символов правильной формулой:

$f(\omega) =$ «ω является правильной формулой», $f : \Omega 1(X) \to$ {да, нет}.

Мы не будем подробно расписывать ход решения этой задачи. Приведем сразу индуктивное расширение $F = (f1, f2, f3)$ функции f, где

f1 = «ω может быть продолжена до правильной формулы», f1: $\Omega \to$ {да, нет},

f2 = «число левых скобок — число правых скобок», f2: $\Omega \to \mathbf{Z}$,

f3 = «последний элемент ω», f3: $\Omega 1 \to X$.

Функцию F можно доопределить на Ω, положив $F(\Lambda) =$ = (да, 0, "+"). Отображение π определим следующим образом:

$f(\omega) =$ да $\Leftrightarrow F(\omega) =$ (да, 0, "t") или $F(\omega) =$ (да, 0, ")").

Таким образом, пустая последовательность «кончается» знаком + и «является» неправильной формулой. Этот же вывод можно сделать и из очевидного тождества

$\forall \omega \in \Omega 1 \quad f(\omega) = f("t + " * \omega)$.

Подставляя формально в это тождество $\omega = \Lambda$, получим $f(\Lambda) =$ $= f("t+") =$ нет.

```
программа формула ⟨вх : P⟩ правильна : да/нет
. дано      : P : последовательность элементов типа символ
.           | P состоит из символов "(", ")", "t", "+"
. получить:
```

```
. --------
. Р.встать в начало
. начало правильно := да
. разность         := 0
. последний элемент := "+"
. цикл ∀x ∈ непрочитанной части Р
. .  пока начало правильно
. .  выполнять
. .  выбор
. .  . при x = "(" ⇒ разность.увеличить на 1
. .  . при x = ")" ⇒ разность.уменьшить на 1
. .  конец выбора
. .  начало правильно := разность ⩾ 0 и
. .          сочетание ⟨последний элемент,x⟩ допустимо
. .  последний элемент := x
. конец цикла
. ответ := начало правильно и разность = 0 и
.        (последний элемент = ")" или последний элемент = "t")
конец программы

программа сочетание ⟨вх: x1,x2: символ⟩ допустимо: да/нет
. дано     :
. получить:
. --------
. ответ := (x1="(" и (x2="(" или x2="t")) или
.          (x1="+" и (x2="(" или x2="t")) или
.          (x1="t" и (x2=")" или x2="+")) или
.          (x1=")" и (x2=")" или x2="+"))
конец программы
```

Замечание 1. *Бинарные операции и индуктивные функции.* В частном случае, когда область значений индуктивной функции F совпадает с алфавитом X, действие $\circ$ является бинарной операцией на X и значение F на $\omega = x_1x_2 \ldots x_n$ выражается формулой

$$F(\omega) = x_1 \circ x_2 \circ \ldots \circ x_n,$$

где действия выполняются слева направо. На $\Omega 1$ функция F доопределяется формулой $F(x) = x$ $\forall x \in X$. F можно продолжить на Ω, положив $F(\Delta) = e$, тогда и только тогда, когда $\forall x \in X$ $e \circ x = x$, т. е. только тогда, когда e является левым нейтральным элементом операции $\circ$. Элемент $s \in X$ является стационарным значением тогда и только тогда, когда $\forall x \in X$ $s \circ x = s$.

Так, например, числовая функция «сумма элементов» задается бинарной операцией $+$, «произведение элементов» — операцией $*$, «максимум элементов» — бинарной операцией max.

Замечание 2. *Однопроходные алгоритмы и минимальность.* При решении конкретных задач, связанных с последовательностями, возможны разные стратегии. Например, задачу «число максимальных элементов последовательности» можно решать так: сначала перебрать все элементы последовательности и найти максимум, потом перебрать элементы последовательности еще раз и подсчитать число элементов, равных найденному максимуму. Каждый перебор всех элементов последовательности в программировании принято называть *проходом*. Общие стратегии (алгоритмы) работы с последовательностями принято классифицировать по числу проходов и называть *однопроходными*, *двухпроходными* и т. д. Одну и ту же задачу можно решать за разное число проходов. Схема индуктивного вычисления функций на пространстве последовательностей всегда приводит к однопроходному алгоритму, т. е. алгоритму, который читает элементы последовательности от начала к концу по одному разу каждый. Обычно однопроходные алгоритмы выполняются быстрее, чем многопроходные. Таким образом, индуктивное вычисление является оптимальным по скорости.

Минимальность индуктивного расширения означает минимальность по памяти, т. е. по объему информации, которая хранится в процессе вычисления.

Таким образом, основной теоретический результат этого раздела можно сформулировать так: *для любой функции f на пространстве последовательностей существует минимальный по памяти вычисляющий f однопроходный алгоритм; этот алгоритм единствен с точностью до изоморфизма.*

Замечание 3. *Другие программные структуры данных.* Изложенный выше метод индуктивного вычисления значений функций применим и к функциям, заданным на других программных структурах — векторах, списках, очередях и т. п. Например, значение в точке t0 производной многочлена, коэффициенты которого в порядке убывания степеней расположены в динамическом векторе, можно найти так:

```
программа производная многочлена ⟨вх : v⟩ в точке ⟨вх : t0⟩ : R
. дано     : v : динамический вектор элементов типа R,
.            t0 : R
. получить: | значение в точке t0 производной многочлена,
.           | коэффициенты которого расположены в v
.           | по убыванию степеней
. --------
. df := 0
. f := 0
. цикл ∀i ∈ 1..v.число элементов выполнять
```

```
. .  df := df * t0 + f
. .  f  := f * t0 + v (i)
. конец цикла
. ответ := df
конец программы
```

Элементы других структур не обязательно всегда перебирать в том порядке, в котором они находятся в структуре. Если, например, в динамическом векторе коэффициенты многочлена расположены в порядке возрастания степеней, то можно перебирать их от конца к началу. Тогда в получающейся последовательности ω они будут расположены в порядке убывания степеней и можно будет применить схему Горнера:

```
программа производная многочлена ⟨вх : v⟩ в точке ⟨вх : t0⟩ : R
. дано     : v : динамический вектор элементов типа R,
.            t0 : R
. получить: | значение в точке t0 производной многочлена,
.           | коэффициенты которого расположены в v
.           | по возрастанию степеней
. --------
. df := 0
  f := 0
. цикл ∀i ∈ 1..v.число элементов реверс выполнять
. .  df := df * t0 + f
. .  f := f * t0 + v (i)
. конец цикла
. ответ := df
конец программы
```

Замечание 4. *Разные задачи и схема индуктивного вычисления функции.* Пространство последовательностей полезно уметь узнавать в самых разных ситуациях. Вернемся, например, к задаче «число препятствий в полосе» из разд. 4. Продвигаясь вдоль полосы с запада на восток и анализируя клетку слева от Путника, мы фактически имеем дело с последовательностью элементов типа «занято/свободно» и с функцией «число препятствий в полосе», определенной на таких последовательностях. Попробуем решить эту задачу регулярным образом, находя индуктивное расширение искомой функции:

$$f(\omega * x) = \begin{cases} f(\omega) + 1, & \text{если } x \text{ — первая клетка нового} \\ & \text{препятствия,} \\ f(\omega) & \text{в противном случае.} \end{cases}$$

В правой части используется информация о соответствии x первой клетке нового препятствия. Это так тогда и только тогда, когда

x = занято и последний элемент ω ≡ свободно. Функция

F = (f, последний элемент ω)

является индуктивным расширением f, и программу можно записать в виде:

```
программа число препятствий в полосе
. дано      : | Путник у западного края под полосой препятствий
.             | лицом на восток
. получить: | Путник у восточного края, в Счетчике — число
.             | препятствий в полосе
. объекты:
.  было занято: да/нет
. --------
```

Объект «было занято» будет содержать информацию о последнем элементе ω. Поскольку F определена на непустых последовательностях, а положение Путника в начальный момент соответствует состоянию «прочитан один элемент последовательности», то прежде всего установим начальное значение F, т. е. состояние Счетчика и объекта «было занято»:

```
. Счетчик установить в нуль
. если Путник.слева занято то
. .  Счетчик.увеличить на единицу
. конец если
. было занято := Путник.слева занято
```

Само индуктивное вычисление F записывается как обычно, только вместо чтения элемента последовательности надо перемещать Путника на восток:

```
. цикл пока Путник.впереди свободно выполнять
. .  Путник.сделать шаг  | т. е. получить очередной элемент ω
. .  если Путник.слева занято и не было занято то
. .  .  Счетчик.увеличить на единицу
. .  конец если
. .  было занято := Путник.слева занято
. конец цикла
конец программы
```

Заметьте, что в аналогичной программе в разд. 4 никакие объекты не используются. Возникает вопрос: как же так — ведь функция «число препятствий в полосе» не является индуктивной и без дополнительной информации ее за один проход вычислить нельзя! Дело в том, однако, что в разд. 4 мы тоже использовали эту безусловно необходимую информацию о последней пройденной клетке, но с чисто программистской точки зрения делали это по-другому:

вместо того чтобы запоминать текущее состояние в специальном объекте «было занято», потом делать шаг и смотреть, что было раньше, мы сначала анализировали текущее состояние и попадали в один из двух случаев в выборе, а уж потом — внутри выбора — делали очередной шаг. Образно говоря, информация о последней пройденной клетке запоминалась не в специальном объекте, а в тексте программы.

В общем случае, если $F(\omega) = (f(\omega), f1$ (последний элемент $\omega))$ и число различных значений f1 невелико, можно хранить состояние f1 не в объекте программы, а в ее тексте, т. е. использовать выбор с соответствующим количеством случаев: сначала анализировать состояние f1, а потом — внутри выбора — читать очередной элемент последовательности и вычислять новое значение f.

ЗАДАЧИ И УПРАЖНЕНИЯ

В приведенных ниже задачах указана функция, значение которой надо вычислить. Требуется определить, является ли эта функция индуктивной. Если да — выписать G, если нет — придумать индуктивное расширение и выписать G для него. Написать программу ⟨вх : P, вых : искомое значение⟩, считая, что ω — это состояние последовательности P элементов типа X. Как и раньше, если противное явно не оговорено, под X понимается произвольный алфавит из двух или более символов.

1. Последний элемент ω, f: $\Omega 1 \to X$.
2. В ω встречается x0, f: $\Omega \to$ {да, нет}.
3. Число элементов, удовлетворяющих условию Q(x), f: $\Omega \to \mathbf{Z}+$.
4. Максимальное значение, f: $\Omega 1(R) \to R$.
5. Максимальное значение, f: $\Omega(R+) \to R+$.
6. Минимальное значение, f: $\Omega 1(R+) \to R+$.
7. Минимальное и максимальное значения, f: $\Omega 1(R) \to R*R$.
8. Три наибольших значения, f: $\Omega 3(R) \to M \in R^3$, $M = \{(x1, x2, x3)\colon x1 \leqslant x2 \leqslant x3\}$.
9. Сумма элементов, f: $\Omega(R) \to R$.
10. Произведение элементов, f: $\Omega(R) \to R$.

В задачах 11—14 SA обозначает множество всех конечных подмножеств множества A.

11. Объединение элементов, f: $\Omega(SA) \to SA$, A — любое.
12. Пересечение элементов, f: $\Omega(SA) \to SA$, A — любое.
13. Объединение элементов, f: $\Omega(SA) \to SA$, A = {0, 1}.
14. Пересечение элементов, f: $\Omega(SA) \to SA$, A = {0, 1}.
15. Наибольший общий делитель элементов, f: $\Omega 1(N) \to N$.
16. Величина числа, записанного в двоичной системе счисления, f: $\Omega 1(\{0, 1\}) \to Z+$.

17. Все элементы равны между собой, f: $\Omega(X) \to$ {да, нет}.

18. Число различных элементов, f: $\Omega(X) \to Z+$.

19. Число пар соседних элементов ω, удовлетворяющих условию $Q(x1, x2)$, f: $\Omega(X) \to Z+$.

20. Номер последнего элемента, равного x0, f: $\Omega(X) \to Z+$, $f(\Delta) = 0$.

21. Число вхождений в ω подпоследовательности $p_1p_2 \ldots p_k$, все p_i различны, f: $\Omega(X) \to Z+$.

22. Число локальных максимумов, f: $\Omega(R) \to Z+$. (Элемент последовательности называется локальным максимумом, если у него нет соседа большего, чем сам элемент. В любой одноэлементной последовательности, например, ровно один локальный максимум).

23. Последовательность возрастает, f: $\Omega(R) \to$ {да, нет}, $f(\Delta) =$ = да.

24. Номер первого элемента, имеющего максимальное значение, f: $\Omega(R) \to Z+$, $f(\Delta) = 0$.

25. Значение записанного по возрастающим степеням многочлена в точке t0, f: $\Omega(R) \to R$, $f(\Delta) = 0$.

26. Значение производной записанного по возрастающим степеням многочлена в точке t0, f: $\Omega(R) \to R$, $f(\Delta) = 0$.

27. Значение k-й производной записанного по убывающим степеням многочлена в точке t0, f: $\Omega(R) \to R$, $f(\Delta) = 0$.

28. Средняя длина связной возрастающей подпоследовательности, f: $\Omega 1(R) \to R+$.

29. Средняя длина связной постоянной подпоследовательности; f: $\Omega 1(R) \to R+$.

30. Средняя длина связной группы пробелов, f: $\Omega(S) \to R+$.

31. Средняя длина идентификатора, f: $\Omega(S) \to R+$ (идентификатором называется последовательность букв и цифр, начинающаяся с буквы).

32. Количество вхождений в последовательность цифры, которая встречается в последовательности максимальное число раз, f: $\Omega(X) \to Z+$, $X = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

33. Размерность пространства, натянутого на последовательность векторов R^2, f: $\Omega(R^2) \to Z+$.

34. Сумма квадратов отрицательных элементов числовой последовательности, f: $\Omega(R) \to R$.

35. Число элементов числовой последовательности, больших всех предыдущих элементов, f: $\Omega(R) \to Z+$.

ГЛАВА 2
НЕСКОЛЬКО ПРИМЕРОВ ПРОГРАММ

9. Проект «Выпуклая оболочка последовательно поступающих точек плоскости»

Формулировка задачи. Пусть требуется создать исполнителя для вычисления некоторых геометрических характеристик (например, периметра и площади) выпуклой оболочки последовательно поступающих точек плоскости.

Напомним, что выпуклой оболочкой conv(M) множества M точек плоскости называется минимальное выпуклое множество, содержащее все заданные точки. Если представить себе точки конечного множества в виде вбитых в доску гвоздей, то выпуклая оболочка — это многоугольник, форму которого принимает натянутое на гвозди резиновое кольцо (рис. 9.1).

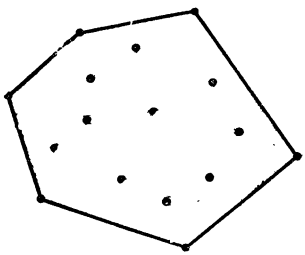


Рис. 9.1. Выпуклая оболочка множества точек плоскости

Слова «последовательно поступающих» означают, что точки множества не заданы все сразу, а поступают друг за другом, причем в любой момент времени исполнитель должен уметь ответить на вопрос о периметре или площади выпуклой оболочки уже поступивших точек. Таким образом, требуется создать исполнителя «Выпуклая оболочка» со следующей системой предписаний:

1. начать работу
2. добавить точку ⟨вх : точка R2⟩
3. периметр : **число**
4. площадь : **число**
5. кончить работу

где тип:

точка R2 = **запись**(X, Y: **число**)

Никакого базового исполнителя в условии задачи не задано, т. е. при реализации Выпуклой оболочки можно пользоваться лишь возможностями Универсального Выполнителя.

Естественно, возможны разные стратегии решения этой задачи, например:

1) хранить все поступающие точки — это позволяет мгновенно обрабатывать добавление новой точки, но требует существенных усилий при вычислении периметра и площади;

2) хранить только вершины выпуклой оболочки уже поступивших точек (очевидно, что этого достаточно для вычисления выпуклой оболочки при добавлении новой точки) — это сокращает объем хранимой информации и ускоряет вычисление периметра и площади, но усложняет добавление новой точки;

3) хранить не только выпуклую оболочку уже поступивших точек, но и ее периметр и площадь, т. е. вообще все действия производить при добавлении новой точки, а по предписаниям «периметр» и «площадь» лишь выдавать готовые значения.

Выбор той или иной стратегии решения должен определяться спецификой задачи: если точки поступают часто и их надо обрабатывать быстро, а запросы на периметр и площадь редки, то можно выбрать первую стратегию; если, наоборот, точки поступают редко и время их обработки не очень важно, а запросы про периметр и площадь требуют максимально быстрого ответа, то лучше избрать третий вариант. Мы будем считать, что условия задачи требуют мгновенной выдачи периметра и площади выпуклой оболочки уже поступивших точек, и будем придерживаться третьей стратегии.

Основные идеи. Фактически в этой задаче мы имеем дело с функцией на пространстве последовательностей точек плоскости. Значением функции является выпуклая оболочка точек последовательности, а также периметр и площадь этой оболочки. Схема индуктивного вычисления функции в данном случае означает, что при добавлении новой точки x мы не вычисляем заново новую выпуклую оболочку $f(\omega*x)$, а модифицируем старую оболочку $f(\omega)$, т. е. выражаем $f(\omega*x)$ через $f(\omega)$ и x.

Отложим пока вопросы про периметр и площадь и посмотрим, как меняется выпуклая оболочка при добавлении новой точки. Ясно, что если новая точка оказалась внутри или на границе старой выпуклой оболочки, то оболочка не меняется. Если же новая точка оказалась вне старой оболочки, то оболочка меняется так, как изображено на рис. 9.2. Если в новой точке расположить лампочку, то она осветит удаляемые ребра старой оболочки; поэтому будем называть эти ребра освещенными из новой точки или просто освещенными. Алгоритм модификации выпуклой оболочки можно описать так:

1) удалить освещенные ребра, 2) концы оставшейся ломаной соединить двумя новыми ребрами с новой точкой. (*)

Мы, впрочем, описали только общий случай. Если новая точка лежит на продолжении одного из ребер, то оболочка должна

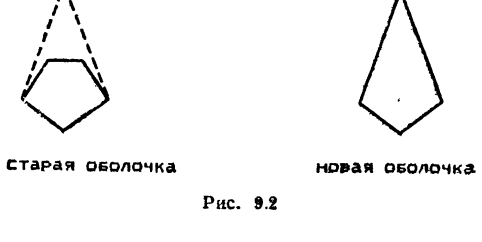


Рис. 9.2

измениться так, как показано на рис. 9.3. Поэтому ребро, на продолжении которого лежит точка, мы также будем считать освещенным.

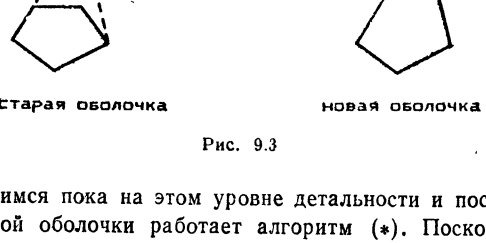


Рис. 9.3

Остановимся пока на этом уровне детальности и посмотрим, начиная с какой оболочки работает алгоритм (*). Поскольку в нем фигурируют «оставшаяся ломаная» и ее концы, то алгоритм работает, лишь начиная с невырожденного многоугольника. Следовательно, пустая (нет точек), нульмерная (одна точка) и одномерная (отрезок) выпуклые оболочки — это особые случаи, которые должны обрабатываться отдельно.

По сути дела, это первое (после выбора стратегии) волевое решение — мы разбили нашу задачу на две: перечисленные выше особые случаи и общий случай, когда выпуклая оболочка двумерна. Поэтому, действуя в соответствии с технологией «сверху вниз», примем решение, что для обработки общего случая у нас будет специальный исполнитель «Многоугольник», и попробуем реализовать исполнителя «Выпуклая оболочка» на его базе. (Заметим, что эти решения действительно являются волевыми — мы могли бы вместо них пытаться заменить (*) на алгоритм, который работает, начиная с отрезка или даже с одной точки, но не стали этого делать.)

Первый шаг декомпозиции. Итак, реализуем исполнителя «Выпуклая оболочка» на базе исполнителя «Многоугольник», который должен производить все действия с невырожденной (двумерной)

выпуклой оболочкой. Начнем с предписания «добавить точку». Поскольку нам надо как-то различать, с каким случаем мы имеем дело, то введем глобальную переменную

число углов: **перечисление** (нуль, один, два, много)

которая будет содержать число «углов» (вершин) в выпуклой оболочке. Тогда программу «добавить точку» можно записать в виде:

программа добавить точку ⟨**вх**: T : точка R2⟩
. **дано** : | поступила новая точка T
. **получить**: | новая оболочка = conv (старая оболочка, T)
. ---
. **выбор**
. . **при** число углов = нуль ⇒ обработка нульугольника ⟨**вх**: T⟩
. . **при** число углов = один ⇒ обработка одноугольника ⟨**вх**: T⟩
. . **при** число углов = два ⇒ обработка двуугольника ⟨**вх**: T⟩
. . **при** число углов = много ⇒ обработка многоугольника ⟨**вх**: T⟩
. **конец выбора**
конец программы

где «обработка нульугольника», «обработка одноугольника» и др. — это подпрограммы (локальные предписания исполнителя «Выпуклая оболочка»). Нам, впрочем, понадобится где-то хранить выпуклую оболочку уже поступивших точек. Если она не вырождена, мы будем хранить ее в исполнителе «Многоугольник», а для особых случаев заведем глобальные объекты A и B типа «точка R2», в которых будем хранить точку (в A) нульмерной оболочки или концы отрезка (A, B) одномерной.

Таким образом, обработка нульугольника состоит просто в запоминании поступившей точки и получении одноугольника:

программа обработка нульугольника ⟨**вх**: T : точка R2⟩
. **дано** : число углов = нуль | старая оболочка пуста
. **получить**: | новая оболочка = T
. ---
. A := T
. число углов := один
конец программы

При обработке одноугольника новая точка может попасть в старую выпуклую оболочку (совпасть со старой точкой) — в этом случае оболочка не меняется:

программа обработка одноугольника ⟨**вх**: T : точка R2⟩
. **дано** : число углов = один | старая оболочка — точка A
. **получить**: | новая оболочка = conv (A, T)
. ---

```
. если Т = А то ничего не делать иначе
. . В := Т
. . число углов := два
. конец если
конец программы
```

При обработке двуугольника (отрезка) возможны уже три разных случая: новая точка принадлежит старому отрезку (оболочка не меняется); новая точка вне отрезка, но на одной прямой с ним (отрезок меняется, но остается отрезком); новая точка находится в общем положении с концами отрезка (оболочка становится треугольником). При анализе этих случаев нам придется узнавать, находятся ли точки А, В и Т на одной прямой, находится ли точка Т между А и В и т. п. Для выполнения таких элементарных действий на плоскости мы введем исполнителя «Планиметр» и будем реализовывать Выпуклую оболочку на базе исполнителей «Многоугольник» и «Планиметр».

Не выписывая дальнейших рассуждений, приведем полную реализацию исполнителя «Выпуклая оболочка»:

```
исполнитель Выпуклая оболочка
. СП:
.       1. начать работу
.       2. добавить точку ⟨вх: точка R2⟩
.       — обработка нульугольника ⟨вх: точка R2⟩
.       — обработка одноугольника ⟨вх: точка R2⟩
.       — обработка двуугольника ⟨вх: точка R2⟩
.       — обработка многоугольника ⟨вх: точка R2⟩
.       3. периметр                                  : число
.       4. площадь                                   : число
.       5. кончить работу
.
. типы:
.       точка R2 = запись (X, Y: число)
.
. объекты:
.       число углов: перечисление (нуль, один, два, много)
.       A, B       : точка R2
.
. идеи реализации:
.       будем хранить выпуклую оболочку поступивших точек и при
.       поступлении новой точки модифицировать ее и ее характе-
.       ристики (периметр, площадь)
.
```

```
. используемые исполнители:
.       Многоугольник (М) и Планиметр (П)
конец описаний |----------------------------------

программа начать работу == число углов.установить в нуль
программа добавить точку ⟨вх: Т: точка R2⟩
. дано     : |поступила новая точка Т
. получить: |новая оболочка = conv (старая оболочка, Т)
. --------
. выбор
. . при число углов = нуль  ⇒ обработка нульугольника  ⟨вх: Т⟩
. . при число углов = один  ⇒ обработка одноугольника  ⟨вх: Т⟩
. . при число углов = два   ⇒ обработка двуугольника   ⟨вх: Т⟩
. . при число углов = много ⇒ обработка многоугольника ⟨вх: Т⟩
. конец выбора
конец программы

программа обработка нульугольника ⟨вх: Т: точка R2⟩
. дано     : число углов = нуль |старая оболочка пуста
. получить: |новая оболочка = Т
. --------
. А := Т
. число углов := один
конец программы

программа обработка одноугольника ⟨вх: Т: точка R2⟩
. дано     : число углов = один |старая оболочка — точка А
. получить: |новая оболочка = conv (А, Т)
. --------
. если Т = А то ничего не делать иначе
. .   В := Т
. .   число углов := два
. конец если
конец программы

программа обработка двуугольника ⟨вх: Т: точка R2⟩
. дано     : число углов = два |старая оболочка = отрезок [А, В]
. получить: |новая оболочка = conv ([А, В], Т)
. --------
. выбор
. . при П.точки ⟨А, В, Т⟩ не на одной прямой ⇒
. .                  М.начать работу ⟨вх: А, В, Т⟩
. .                  число углов := много
. .
. . при П.точка ⟨Т⟩ внутри отрезка ⟨А, В⟩ ⇒ ничего не делать
```

```
. . при П.точка ⟨B⟩ внутри отрезка ⟨A,T⟩ ⇒ B := T
. . при П.точка ⟨A⟩ внутри отрезка ⟨T,B⟩ ⇒ A := T
. конец выбора
конец программы

программа обработка многоугольника ⟨вх: T⟩ == М.обработать
          ⟨вх: T⟩
программа периметр: число
. дано     :
. получить:
. --------
. выбор
. . при число углов = много ⇒ ответ := М.периметр многоугольника
. . при число углов = два  ⇒ ответ := П.R⟨A,B⟩ + П.R⟨B,A⟩
. . иначе                  ⇒ ответ := 0.0
. конец выбора
конец программы

программа площадь: число
. дано     :
. получить:
. --------
.  если число углов ≠ много то ответ := 0.0 иначе
.    ответ := М.площадь многоугольника
.  конец если
конец программы

программа кончить работу
. дано     :
. получить:
. --------
.  если число углов = много то М.кончить работу конец если
конец программы
конец исполнителя | ==========================
```

Заметьте, что мы начинаем работу с исполнителем «Многоугольник» только в тот момент, когда выпуклая оболочка становится двумерной. Возможен случай, когда этого не происходит. Поэтому в программе «кончить работу» мы кончаем работу с исполнителем «Многоугольник» только в том случае, если оболочка является двумерной.

Второй шаг декомпозиции. Наша задача сейчас — реализовать исполнителя «Многоугольник», система предписаний которого выглядит так:

1. начать работу ⟨вх: A, B, C: точкаR2⟩
2. обработать ⟨вх: T: точка R2⟩

3. периметр многоугольника : **число**
4. площадь многоугольника : **число**
5. кончить работу

Поскольку мы приняли решение все действия, включая вычисление периметра и площади выпуклой оболочки, производить при добавлении новой точки, то у нас будут глобальные объекты

периметр, площадь: **число**

значения которых будут модифицироваться при поступлении новой точки. Программы «периметр многоугольника» и «площадь многоугольника» при этом становятся тривиальными:

программа периметр многоугольника: **число** = = периметр
программа площадь многоугольника: **число** = = площадь

Напомним, что такая запись эквивалентна следующей:

программа площадь многоугольника: **число**
. **дано** :
. **получить:**
. ---------
. **ответ** := площадь
конец программы

Основное, что нам осталось, — это реализовать предписание «обработать», т. е. модифицировать выпуклую оболочку, а также ее периметр и площадь при поступлении новой точки. Прежде всего заметим, что понятие освещенности в (*) было определено неформально и при этом использовалось расположение точки относительно всего многоугольника. Желательно было бы, чтобы освещенность или неосвещенность ребра определялась *только по точке и ребру*. Это можно сделать для ориентированных ребер (ориентированного многоугольника) следующим образом: рассмотрим ориентированное ребро AB, новую точку T и ориентированную площадь треугольника ABT (рис. 9.4): S(ΔABT) = = ((A.X — T.X)*(B.Y — T.Y) — (A.Y — T.Y)*(B.X — T.X))*0.5 (ориентированная площадь треугольника считается как половина площади параллелограмма, натянутого на векторы TA и TB).

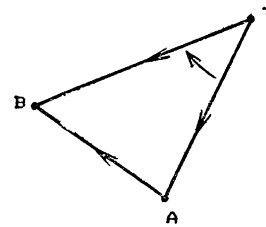


Рис. 9.4

Легко видеть (рис. 9.5), что если многоугольник ориентирован «против часовой стрелки», то ребро AB освещено из T тогда и только тогда, когда

s(ΔABT) < 0 **или** (s(ΔABT) = 0 **и** точка T вне отрезка AB).

Заметим также, что при таком формальном определении понятия освещенности никакое ребро многоугольника не освещено ни из одной точки внутри или на границе многоугольника.

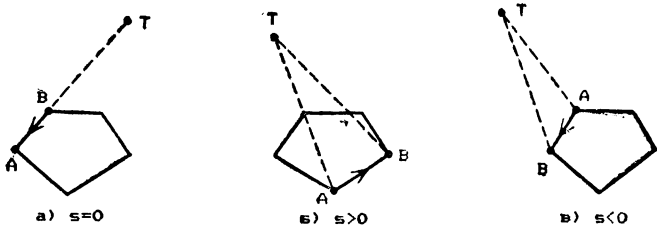


Рис. 9.5

Далее надо понять, как хранить многоугольник (выпуклую оболочку). Мы примем решения

1) хранить линейно упорядоченную последовательность вершин многоугольника (ориентация против часовой стрелки);

2) сами эти вершины хранить в деке Д элементов типа «точка R2» (рис. 9.6) *). Ребро a, соединяющее конец дека с началом, назовем *текущим*. Поскольку в деке в каждый момент доступны только два элемента — начало и конец, то и работать можно только

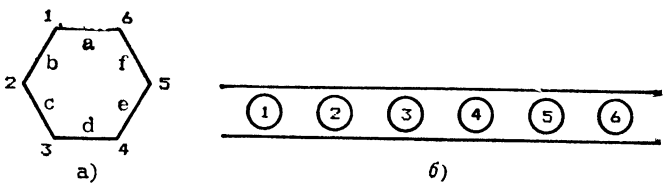


Рис. 9.6. а) многоугольник, б) состояние дека (указаны номера точек плоскости)

с текущим ребром многоугольника. Текущее ребро многоугольника, впрочем, легко сменить: если переместить один элемент из начала дека в конец, то многоугольник не изменится, а текущим ребром станет ребро b (следующее за a в порядке против часовой стрелки).

*) Хотя выбор дека и кажется произвольным, это не вполне верно. Выбор конкретной структуры для хранения вершин многоугольника определяется теми операциями, которые будут производиться над вершинами при обработке новой точки. Таким образом, нам следовало попытаться как-нибудь (вчерне) написать программу «обработать», прикинуть, какие операции над вершинами нужны, и лишь после этого выбирать подходящую структуру и реализовывать программу «обработать» начисто. Так оно и было на самом деле. Здесь, однако, мы описываем лишь «чистовой» вариант, что и приводит к кажущейся произвольности решения 2.

Если переместить из начала дека в конец еще один элемент, то текущим станет ребро с и т. д.

Таким образом, модификация выпуклой оболочки при поступлении новой точки достаточно формализована, и мы можем начать писать соответствующую программу. Осталось еще понять, как должны модифицироваться периметр и площадь. Желательно было бы описать эти изменения в терминах элементарных операций над ребрами — что надо делать с периметром и площадью при удалении одного отдельно взятого освещенного ребра и что при добавлении новых ребер. В данном случае это достаточно просто: периметр надо при удалении ребра уменьшать на длину ребра, а при добавлении увеличивать; площадь же надо увеличивать на величину площади треугольника АТВ при удалении каждого освещенного ребра АВ, а при добавлении новых ребер ничего делать не надо (рис. 9.7). Заштрихованная на рис. 9.7 площадь состоит из площадей треугольников АВТ для каждого освещенного ребра АВ (на рис. 9.7 таких ребер три). Далее нам понадобится еще и число ребер выпуклой оболочки, поэтому кроме периметра и площади будем модифицировать также и число ребер. Подпрограмму, учитывающую удаление освещенного ребра, можно записать так:

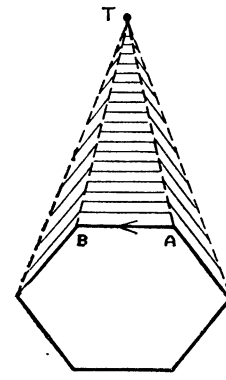


Рис. 9.7

```
программа учесть удаление ребра (вх: A,B) (вх: T: точка R2)
. дано      : A,B: точка R2 | удаляемое ребро
.             и ребро (A,B) освещено из (T)
. получить: | число ребер, периметр и площадь соответствующим
.           | образом изменены
. --------
.  число ребер.уменьшить на 1
.  периметр.уменьшить на П.R(A,B)
.  площадь.увеличить на abs(П.S(A,B,T))
конец программы
```

Саму программу обработки новой точки мы напишем по ходу реализации исполнителя «Многоугольник»:

```
исполнитель Многоугольник
. СП:
.  1. начать работу (вх: A,B,C: точка R2)
.  2. обработать (вх: T: точка R2)
```

```
. — ребро ⟨вх: A, B: точка R2⟩ освещено из ⟨вх: T: точка R2⟩
.                                                    : да/нет
. — учесть удаление ребра ⟨вх: A, B: точка R2⟩⟨вх: T: точка R2
. 3. периметр многоугольника                         : число
. 4. площадь многоугольника                          : число
. 5. кончить работу
.
. объекты:
. Д          : дек элементов типа точка R2
. число ребер: целое
. периметр   : число
. площадь    : число
.
. идеи реализации:
.  Вершины многоугольника хранятся в деке (последовательно).
. Порядок вершин в деке определяет ориентацию ребер много-
. угольника. Этот порядок выбирается так, что никакое ребро
. многоугольника не освещено ни из какой точки многоугольника
.
. используемые исполнители:
. Планиметр (П)
конец описаний | ----------------------------------

программа начать работу ⟨вх: A, B, C: точка R2⟩
. дано     : П.точки ⟨A, B, C⟩ не на одной прямой
. получить: | создан треугольник правильной ориентации, т. е.
.           | точки внутри треугольника не освещают его ребер
. --------
. Д.начать работу
. Д.добавить элемент ⟨B⟩ в начало дека
. если ребро ⟨A, C⟩ освещено из ⟨B⟩
. . то
. . Д.добавить элемент ⟨A⟩ в начало дека
. . Д.добавить элемент ⟨C⟩ в конец дека
. . иначе
. . Д.добавить элемент ⟨C⟩ в начало дека
. . Д.добавить элемент ⟨A⟩ в конец дека
. конец если
.  утв: | точки ΔABC не освещают его ребер; в частности,
.        ребро ⟨Д.конец, Д.начало⟩ не освещено из ⟨B⟩
. число ребер := 3
. периметр  := П.R⟨A, B⟩ + П.R⟨B, C⟩ + П.R⟨C, A⟩
. площадь   := abs (П.S⟨A, B, C⟩)
конец программы
```

```
программа обработать (вх: Т: точка R2)
. дано      : | поступила новая точка Т
. получить: | новая оболочка = conv (старая оболочка, Т)
. ---------
```

Прежде всего надо понять, освещено ли из новой точки хоть одно ребро многоугольника:

```
. цикл число ребер раз
. . пока ребро (Д.конец, Д.начало) не освещено из (Т)
. . выполнять
. .  Д.взять элемент из начала дека в (вых: Х)
. .  Д.добавить элемент (вх: Х) в конец дека
. конец цикла
.  утв: | ребро (Д.конец, Д.начало) освещено из (Т) или
.      | в многоугольнике нет ни одного освещенного ребра
```

Дальше возникают два случая. Если в многоугольнике освещенных ребер нет, то это означает, что новая точка попала внутрь или на границу старой выпуклой оболочки, и, значит, больше ничего делать не надо. Если же ребро (Д.конец, Д.начало) освещено, то надо удалить все освещенные ребра и соединить концы оставшейся ломаной с новой точкой. При выполнении этих действий надо одновременно модифицировать периметр, площадь и число ребер оболочки:

```
. если ребро (Д.конец, Д.начало) освещено из (Т) то
. .
. . учесть удаление ребра (Д.конец, Д.начало) (вх: Т)
. .
. . | -- удалить освещенные ребра из начала дека:
. .       Д.взять элемент из начала дека в (вых: Х)
. .       цикл пока ребро (Х, Д.начало) освещено из (Т)
. .       . выполнять
. .       . учесть удаление ребра (Х, Д.начало) (вх: Т)
. .       . Д.взять элемент из начала дека в (вых: Х)
. .       конец цикла
. .       Д.добавить элемент (вх: Х) в начало дека
```

Фрагмент программы для удаления освещенных ребер из конца дека реализуйте самостоятельно.

```
. . | -- добавить новые ребра в точку Т:
. . число ребер.увеличить на 2
. . периметр.увеличить на П.R (Д.конец, Т) + П.R (Т, Д.начало)
. . Д.добавить элемент (вх: Т) в начало дека
```

```
. конец если
конец программы

программа ребро ⟨вх: A, B⟩ освещено из ⟨вх: T⟩: да/нет ==
  (П.S ⟨A, B, T⟩ < 0.0) или
  (П.S ⟨A, B, T⟩ = 0.0 и П.точка ⟨T⟩ не внутри отрезка ⟨A, B⟩)

программа учесть удаление ребра ⟨вх: A, B⟩⟨вх: T: точка R2⟩
. дано      : A, B: точка R2 | удаляемое ребро
.             и ребро ⟨A, B⟩ освещено из ⟨T⟩
. получить: | число ребер, периметр и площадь соответствующим
.           | образом изменены
. --------
.  число ребер.уменьшить на 1
.  периметр.уменьшить на П.R ⟨A, B⟩
.  площадь.увеличить на abs (П.S ⟨A, T, B⟩)
конец программы

программа периметр многоугольника: число == периметр
программа площадь многоугольника: число == площадь
программа кончить работу                 == Д.кончить работу
конец исполнителя | ==========================
```

Третий шаг декомпозиции. Осталось реализовать исполнителя «Планиметр»:

```
исполнитель Планиметр
. СП:
.   1. точки ⟨вх: A, B, C: точка R2⟩ на одной прямой   :да/нет
.   2. точка ⟨вх: T: точка R2⟩ внутри отрезка
.                          ⟨вх: A, B: точка R2⟩        :да/нет
.   3. R ⟨вх: A, B: точка R2⟩                          :число
.   4. S ⟨вх: A, B, C: точка R2⟩                       :число
.
. типы:
.   точка R2 = запись (X, Y: число)
конец описаний | ------------------------------

программа точки ⟨вх: A, B, C⟩ на одной прямой: да/нет ==
                                          (S ⟨A, B, C⟩ = 0)
программа точка ⟨вх: T⟩ внутри отрезка ⟨вх: A, B⟩: да/нет ==
                                  (R ⟨A, T⟩ + R ⟨T, B⟩ ≤ R ⟨A, B⟩)
программа R ⟨A, B⟩: число == sqrt (A.X—B.X) ** 2 + (A.Y —
                                                  · B.Y) ** 2)
программа S ⟨A, B, C⟩: число == ((A.X—C.X) * (B.Y—C.Y) —
                                  (A.Y—C.Y) * (B.X—C.X)) * 0.5
конец исполнителя | ==============================
```

Ориентированная площадь треугольника считается как половина площади параллелограмма, натянутого на векторы CA и CB.

ЗАДАЧИ И УПРАЖНЕНИЯ

1. Пользуясь исполнителем «Выпуклая оболочка» как базовым, напишите программу

программа периметр выпуклой оболочки множества ⟨**вх**: M⟩: **число**
• **дано** : M: **множество элементов типа** точка R2
• **получить**: | **ответ**=периметр выпуклой оболочки точек множества
• --------

2. Упростите проект «Выпуклая оболочка», считая, что работа начинается с трех точек в общем положении.

3. Нарисуйте состояние дека после поступления в указанном порядке следующих точек (рис. 9.8).

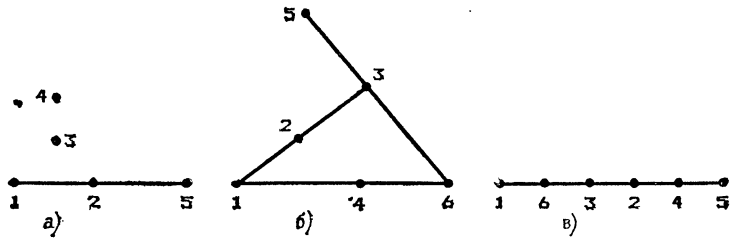


Рис. 9.8

4. Опишите, что произойдет, если изменить знак в программе, анализирующей освещенность ребра из точки, т. е. написать

программа ребро ⟨**вх**: A, B⟩ освещено из ⟨**вх**: T⟩: **да/нет** ==
(П. S ⟨A, T, B⟩ > 0.0) **или**
(П. S ⟨A, T, B⟩ = 0.0 **и** П. точка ⟨T⟩ **не** внутри отрезка ⟨A. B⟩)

Решите упражнение 3 в этой ситуации.

5. Добавьте к исполнителю «Выпуклая оболочка» предписание

точка ⟨**вх**: T: точка R2⟩ внутри выпуклой оболочки : **да/нет**

и реализуйте его.

6. Добавьте к исполнителю «Выпуклая оболочка» предписания

а) диаметр выпуклой оболочки,
б) центр тяжести вершин выпуклой оболочки

и реализуйте их аналогично предписаниям «периметр» и «площадь».

7. Переделайте исполнителя «Выпуклая оболочка» так, чтобы он мог обрабатывать точки трехмерного пространства, расположенные в одной плоскости (если новая точка лежит вне плоскости старых, исполнитель должен давать отказ).

8. Придумайте понятие освещенности, при котором добавление точки к двуугольнику будет происходить по тому же алгоритму, что и добавление точки к многоугольнику. (Разрешается наличие нулевых и развернутых углов, т. е. ребра выпуклой оболочки могут быть расположены на одной прямой.)

9. Переделайте проект «Выпуклая оболочка последовательно поступающих точек плоскости» в проект «Выпуклая оболочка последовательно поступающих точек плоскости Лобачевского (модель Пуанкаре)».

10. Придумайте алгоритм построения выпуклой оболочки, основанный на хранении неупорядоченного множества ребер, где (с математической точки зрения) ребро — это неупорядоченная пара точек плоскости.

11. Реализуйте выпуклую оболочку последовательно поступающих точек в трехмерном пространстве, считая, что работа начинается с четырех точек в общем положении (используйте обобщение алгоритма задачи 10).

12. Сделайте задачу 11 в N-мерном пространстве при N = 10.

10. Компиляция и интерпретация. Реализация простейшего компилятора с языка арифметических формул

Мы уже составляли по формуле соответствующую программу для Стекового калькулятора (разд. 1). Теперь напишем программу, которая делает это для любой формулы *автоматически*. Такая программа называется *компилятором* (переводчиком) с языка арифметических формул на язык программ для Стекового калькулятора. Однако прежде всего мы уточним, что такое «язык» и что такое «компилятор».

Язык и грамматика.

Определения. Пусть A — некоторый *алфавит*, т. е. произвольное непустое множество. Элементы этого множества мы будем называть *символами*. Произвольная конечная последовательность символов алфавита (в том числе и пустая) называется *цепочкой*. Произвольное подмножество LA множества всех возможных цепочек называется *языком над A*.

Над одним и тем же алфавитом можно определить много разных языков. Пусть, например, A = {0, 1} — алфавит из двух символов 0 и 1. Мы можем рассмотреть язык всех возможных цепочек над A; язык непустых цепочек; язык непустых цепочек, начинающихся с 1; язык цепочек, состоящих только из 1, и т. д. За-

метьте, что здесь мы описываем множество цепочек языка неформально, считая, что никаких двусмысленностей или неясностей не возникает. Такое *словесное* задание языка, однако, удобно лишь для простейших случаев. Рассмотрим, например, язык LA правильных арифметических формул над алфавитом A, состоящим из 26 строчных латинских букв, двух скобок и четырех знаков арифметических действий:

$$A = \{a,b,c,d, \ldots z,(,),+,-,*,/\}.$$

Для простоты ограничимся случаем, когда в формуле фигурируют только односимвольные переменные a, b, c, d, ..., z и запрещено использование унарных операций. Таким образом, цепочки

a
b — a * c
((a + b) * (c + d) — a)/(e + f)

являются правильными формулами, т. е. принадлежат языку LA, а цепочки

))) + a * *
aa + b
* a
— *a*

правильными формулами не являются, т. е. языку LA не принадлежат. Заметьте, что в данном случае мы не описали все множество цепочек языка, а проапеллировали к некоторому интуитивному представлению о том, что такое «правильная формула без унарных операций». Являются ли в нашем понимании правильными формулами цепочки

Δ (т. е. пустая цепочка, не содержащая символов)
a + (—b)
(((((a)))))
( )

уже не очень ясно. Поэтому обычно такие достаточно сложные языки задаются не словесно, а формально, с тем чтобы принадлежность цепочки языку можно было доказать или опровергнуть, исходя из определения. Одним из наиболее распространенных способов задания языка является его задание с помощью грамматики.

Определение. Пусть A — некоторый алфавит, M — *метаалфавит*, т. е. какой-то другой алфавит, не пересекающийся с A ($A \cap M = \varnothing$). Элементы метаалфавита M называются *метасимволами*. *Грамматикой* называется система правил (т. е. конечная последовательность строк) вида

$\alpha \rightarrow$ цепочка над $(M \cup A)$,

где $\alpha \in M$ — какой-то метасимвол, в которой для каждого $\alpha \in M$ встречается хотя бы одна строка с α в левой части (до стрелочки) и в которой выделен один метасимвол, называемый *главным*.

Пример. Пусть A — алфавит, над которым мы рассматривали язык правильных арифметических формул, $M = \{\alpha, \beta, \gamma\}$, α — главный метасимвол, а грамматика такова:

$\alpha \to (\alpha)$
$\alpha \to \beta$
$\alpha \to \alpha\gamma\alpha$

$\beta \to a$
$\beta \to b$
$\beta \to c$
$\beta \to d$
. . .
$\beta \to z$
$\gamma \to +$
$\gamma \to -$
$\gamma \to *$
$\gamma \to /$

Содержательно каждое правило грамматики имеет смысл подстановки. Например, строка «$\alpha \to \alpha\gamma\alpha$» означает возможность замены метасимвола α на цепочку $\alpha\gamma\alpha$. Начав с главного метасимвола и пользуясь различными правилами грамматики (т. е. многократно выполняя подстановки), мы можем получать различные цепочки, состоящие из символов и метасимволов. Например, цепочку $(x + y)$ можно получить так (над стрелками указаны те правила грамматики, т. е. те подстановки, с помощью которых получены соответствующие цепочки):

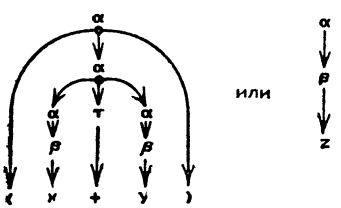


Рис. 10.1. Деревья вывода для формул (x + y) и z

$$\alpha \xrightarrow{\alpha\to(\alpha)} (\alpha) \xrightarrow{\alpha\to\alpha\gamma\alpha} (\alpha\gamma\alpha) \xrightarrow{\alpha\to\beta} (\alpha\gamma\beta) \xrightarrow{\alpha\to\beta} \longrightarrow (\beta\gamma\beta) \xrightarrow{\gamma\to+} (\beta + \beta) \xrightarrow{\beta\to x} (x + \beta) \xrightarrow{\beta\to y} (x + y)$$

А цепочку, состоящую из единственного символа z, так:

$$\alpha \xrightarrow{\alpha\to\beta} \beta \xrightarrow{\beta\to z} z$$

Обычно такие последовательности подстановок и их результаты записывают более компактно в виде так называемого *дерева вывода* или *дерева разбора* (рис. 10.1).

Заметим, что если в цепочке встречается метасимвол, то ее можно преобразовать дальше, применив одно из правил грамматики с этим метасимволом в левой части. Если же метасимволов в цепочке не осталось, то процесс ее преобразования закончен и больше с цепочкой ничего сделать нельзя. По этой причине обычные символы часто также называют *терминалами* («конечными» символами), а метасимволы — *нетерминалами.*

Определение. Все множество цепочек над A (т. е. цепочек без метасимволов), которые можно получить таким образом, называется *языком, порожденным данной грамматикой*, или *языком, заданным с помощью данной грамматики.*

До сих пор мы рассуждали на чисто символическом уровне. Как правило, однако, метасимволы (нетерминалы) имеют некоторый смысл. Например, в приведенной выше грамматике α означает правильную формулу, β — имя переменной, а γ — знак операции. В этом смысле метасимволы можно понимать как имена понятий, которые вводятся для формального описания того или иного языка. Поэтому обычно для их обозначения используют сами понятия, взятые в угловые скобки, например ⟨формула⟩, ⟨имя переменной⟩, ⟨знак операции⟩. Грамматика при этом примет вид

⟨формула⟩→(⟨формула⟩)
⟨формула⟩→⟨имя переменной⟩
⟨формула⟩→⟨формула⟩⟨знак операции⟩⟨формула⟩

и т. д.

Кроме того, правила грамматики с одним и тем же нетерминалом в левой части объединяют, используя вместо стрелочки знак ::= (читается «это есть») и разделяя варианты вертикальной чертой (читается «либо»):

```
⟨формула⟩::=(⟨формула⟩)|⟨имя переменной⟩|
            ⟨формула⟩⟨знак операции⟩⟨формула⟩
⟨имя переменной⟩::=a|b|c|d|...|z                    (*)
⟨знак операции⟩::=+|-|*|/
```

Такую запись правил грамматики принято называть *нормальной формулой Бэкуса — Наура* или сокращенно НФБН. В настоящее время НФБН и ее незначительные модификации широко используются в программистской литературе для описания тех или иных языков программирования.

Один и тот же язык можно описать при помощи разных наборов понятий и, значит, задать с помощью разных грамматик.

Например, грамматика

```
⟨формула⟩: : = ⟨терм⟩ | ⟨терм⟩ + ⟨формула⟩
                      | ⟨терм⟩ — ⟨формула⟩
⟨терм⟩: : = ⟨множитель⟩ | ⟨множитель⟩ * ⟨терм⟩                    (**)
                       | ⟨множитель⟩/⟨терм⟩
⟨множитель⟩: : = (⟨формула⟩) | ⟨имя переменной⟩
⟨имя переменной⟩: : = a | b | c | d | ... | z
```

задает тот же язык, что и грамматика (*) (докажите это!).

Выбор той или иной грамматики (того или иного набора понятий) для описания языка зависит от приложений. Например, если мы только проверяем правильность формулы, то удобно использовать простую грамматику (*). Если же, кроме того, нас интересует старшинство операций при вычислении по формуле, то удобнее грамматика (**), понятия которой это старшинство отражают.

Компиляция. *Компиляцией* (или *трансляцией*) называется преобразование текста (конкретной цепочки символов) с одного языка (А) в семантически эквивалентный текст на другом языке (В). Например, с языка правильных арифметических формул на язык правильных программ для Стекового калькулятора (рис. 10.2). (Мы



Рис. 10.2

будем рассматривать только такие программы для Стекового калькулятора, в которых в качестве входных параметров фигурируют лишь односимвольные имена переменных от a до z.)

Компилятором называется исполнитель (или программа), который *автоматически* осуществляет такое преобразование текста с одного языка на другой.

Рекурсивная реализация компилятора правильных формул. Будем для определенности считать, что язык правильных формул задается грамматикой (**), и попробуем реализовать компилятор в соответствии с этой грамматикой:

исполнитель Компилятор правильных формул
. СП:
. 1. компилировать формулу
. — обработать формулу
. — обработать терм

```
.      — обработать множитель
.      — обработать имя переменной
.
. идеи реализации:
. Правильная формула задается грамматикой с 4 метасимволами:
.
. ⟨формула⟩ ::= ⟨терм⟩ ! ⟨терм⟩ + ⟨формула⟩ ! ⟨терм⟩ — ⟨формула⟩
. ⟨терм⟩ ::= ⟨множитель⟩ ! ⟨множитель⟩ * ⟨терм⟩ ! ⟨множитель⟩/⟨терм⟩
. ⟨множитель⟩ ::= (⟨формула⟩) ! ⟨имя переменной⟩
. ⟨имя переменной⟩ ::= a!b!c! ...!y!z
.
.  Для каждого метасимвола пишется отдельная программа.
. Например, программа «обработать терм» находит в начале
. непрочитанной части последовательности Р терм максимальной
. длины, читает его и печатает соответствующий фрагмент про-
. граммы для Стекового калькулятора.
.
. используемые исполнители:
.   Р: последовательность элементов типа символ
конец описаний |--------------------------------

программа компилировать формулу
. дано     : Р : последовательность элементов типа символ
.          | Р содержит правильную арифметическую формулу
. получить: | содержимое Р откомпилировано, т. е. напечатана
.          | соответствующая программа для Стекового кальку-
.          | лятора
. --------
. Р.встать в начало последовательности
. печатать ⟨"СК.начать работу"⟩
. обработать формулу
.   утв: Р.нет непрочитанных элементов
. печатать ⟨"СК.показать результат"⟩
. печатать ⟨"СК.кончить работу"⟩
конец программы
```

Здесь нам впервые встречается операция «печатать» Универсального Выполнителя, которая печатает на бумаге строку, указанную в угловых скобках в качестве параметров. Например, при выполнении строки

```
печатать ⟨"СК.начать работу"⟩
```

будет напечатано

```
СК.начать работу
```

У операции «печатать» может быть несколько параметров, тогда они печатаются друг за другом. Если в качестве параметра указан некоторый текст в кавычках, то печатается сам этот текст, а если указано имя объекта, то печатается значение объекта. Например, если объект x типа символ имеет значение "a", то при выполнении строки

печатать ⟨"СК.добавить ⟨",x,"⟩ в стек"⟩

будет напечатано

СК.добавить ⟨a⟩ в стек

В используемой нами грамматике понятие ⟨формула⟩ определяется через понятия ⟨терм⟩ и ⟨формула⟩:

⟨формула⟩ ::= ⟨терм⟩ | ⟨терм⟩ + ⟨формула⟩ | ⟨терм⟩ — ⟨формула⟩

Естественно поэтому попытаться реализовать обработку формулы рекурсивно. (Напомним, что рекурсия — это ситуация или программистский прием, состоящий в том, что программа непосредственно или через другие программы обращается к себе как к подпрограмме.)

В соответствии с определением формулы при ее компиляции можно сначала найти и компилировать терм: либо этот терм должен совпадать со всей формулой, либо за ним должен следовать знак + или —. Например, для формулы a * b * c + d нужно сначала компилировать терм a * b * c (но не a * b!), а для формулы (a + b * c) — терм (a + b * c), совпадающий со всей формулой. Таким образом, после компиляции терма возможны три ситуации: 1) в формуле больше символов нет; 2) далее идет знак +, а за ним формула; 3) далее идет знак —, а за ним формула. Соответственно в целом обработку (компиляцию) формулы можно записать так:

```
программа обработать формулу
. дано     : P: последовательность элементов типа символ
.          | непрочная часть P начинается с правильной
.          | арифметической формулы
. получить: | из непрочитанной части P прочитана правильная
.          | формула максимальной длины. Напечатан соответ-
.          | ствующий фрагмент программы для Стекового каль-
.          | кулятора
.---------
. обработать терм
. выбор
. . при P.нет непрочитанных элементов ⇒ ничего не делать
. . при P.очередной элемент ="+"⇒
```

```
. .                    Р.пропустить очередной элемент
. .                    обработать формулу
. .                    печатать ⟨"СК.сложить"⟩
. . при Р.очередной элемент ="—" ⇒
. .                    Р.пропустить очередной элемент
. .                    обработать формулу
. .                    печатать ⟨"СК.вычесть"⟩
. . иначе ⇒ ничего не делать
. конец выбора
конец программы
```

Обратите внимание, что в программе «обработать формулу» мы обрабатываем непрочитанную часть Р не до конца последовательности, а лишь до конца правильной формулы. Это связано с тем, что далее при обработке множителя в соответствии с правилом

⟨множитель⟩ ::= (⟨формула⟩) | ⟨имя переменной⟩

мы будем вызывать программу «обработать формулу» для компиляции выражения внутри скобок, а такая обработка должна заканчиваться при достижении закрывающей скобки.

Наконец, коль скоро мы воспользовались рекурсией, мы обязаны гарантировать, что программа «обработать формулу» не будет вызывать себя бесконечно. В данном случае убедиться в этом довольно легко — поскольку перед каждым новым вызовом программы «обработать формулу» непрочитанная часть Р уменьшается, а последовательность Р конечна, то и вызовов может быть лишь конечное число. Следовательно, рано или поздно обработка формулы закончится. Аналогичным образом в соответствии с грамматикой реализуются программы «обработать терм», «обработать множитель» и «обработать имя переменной»:

```
программа обработать терм
. дано      : | в начале непрочитанной части исполнителя Р — терм
. получить: | из непрочитанной части Р прочитан терм максималь-
.           | ной длины. Напечатан соответствующий фрагмент
.           | программы для Стекового калькулятора
.---------
. обработать множитель
. выбор
. . при Р.нет непрочитанных элементов ⇒ ничего не делать
. . при Р.очередной элемент = "*" ⇒
. .                    Р.пропустить очередной элемент
. .                    обработать терм
. .                    печатать ⟨"СК.умножить"⟩
. . при Р.очередной элемент ="/" ⇒
. .                    Р.пропустить очередной элемент
```

```
. .                обработать терм
. .                печатать ⟨"СК.разделить ⟩
. . иначе ⇒ ничего не делать
. конец выбора
конец программы

программа обработать множитель
. дано     : | в начале непрочитанной части Р — множитель
. получить: | из непрочитанной части Р прочитан множитель макси-
.           | мальной длины. Напечатан соответствующий фрагмент
.           | программы для Стекового калькулятора
. --------
. если Р.очередной элемент ≠"(" то обработать имя переменной
. . иначе
. .   Р.пропустить очередной элемент | т. е. пропустить "("
. .   отработать формулу
. .     утв: Р.очередной элемент = ")"
. .   Р.пропустить очередной элемент | т. е. пропустить ")"
. конец если
конец программы

программа обработать имя переменной
. дано     : "a" ≤ Р.очередной элемент ≤ "z"
. получить: | очередной элемент Р (имя переменной) прочитан.
.           | Напечатана соответствующая строка программы для
.           | Стекового калькулятора
. --------
. печатать ⟨"СК.добавить ⟨", Р.очередной элемент,"⟩ в стек"⟩
. Р.пропустить очередной элемент
конец программы

конец исполнителя |==========================
```

Откомпилируйте в соответствии с написанной выше программой правильную формулу a — b — c. Обратите внимание, что полученная в результате компиляции программа для Стекового калькулятора соответствует формуле a — (b — c), т. е. в этой ситуации наш компилятор работает неверно. Почему так получилось? Чтобы разобраться в этом вопросе, давайте построим дерево вывода формулы a — b — c в грамматике (**) (рис. 10.3). Если мы теперь вместо a, b и c подставим числа и начнем подниматься вверх по дереву вывода, вычисляя значения нетерминалов, то величина b — c будет сначала вычислена, а потом вычтена из a. Наш компилятор работает точно так же. Таким образом, операции одинакового старшинства выполняются справа налево не из-за ошибок в компиляторе, а из-за того, что грамматика (**) языка арифметических

формул не соответствует принятому в арифметике порядку выполнения действий одинакового старшинства.

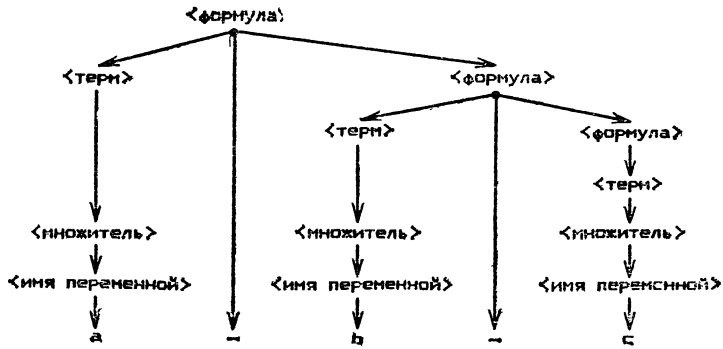


Рис. 10.3. Дерево вывода формулы a — b — c в грамматике (* *)

Грамматику (**) легко преобразовать к виду (***), соответствующему правильному порядку выполнения действий:

$$
\begin{array}{l}
\langle\text{формула}\rangle ::= \langle\text{терм}\rangle \mid \langle\text{формула}\rangle + \langle\text{терм}\rangle \\
\qquad\qquad\qquad\qquad \mid \langle\text{формула}\rangle - \langle\text{терм}\rangle \\
\langle\text{терм}\rangle ::= \langle\text{множитель}\rangle \mid \langle\text{терм}\rangle * \langle\text{множитель}\rangle \\
\qquad\qquad\qquad\qquad \mid \langle\text{терм}\rangle / \langle\text{множитель}\rangle \\
\langle\text{множитель}\rangle ::= (\langle\text{формула}\rangle) \mid \langle\text{имя переменной}\rangle \\
\langle\text{имя переменной}\rangle ::= \mathrm{a} \mid \mathrm{b} \mid \mathrm{c} \mid \mathrm{d} \mid \ldots \mid \mathrm{z}
\end{array}
\qquad (***)
$$

Однако рекурсивно реализовать соответствующий этой грамматике компилятор так, как это было сделано выше, нельзя. Невозможно, например, при обработке формулы по очередному элементу последовательности определить, надо ли обрабатывать терм или формулу. Но даже если бы это оказалось возможным, мы бы не имели права в программе обработки формулы рекурсивно обратиться к себе, так как в этот момент непрочитанная часть последовательности еще не изменилась и, следовательно, в этом месте программа бы обращалась к себе до бесконечности.

Приведенную выше реализацию компилятора правильных арифметических формул можно подправить так, чтобы операции выполнялись в нужном порядке. Заметим, однако, что при этом основное достоинство рекурсивной реализации — простая связь грамматики и программы — будет потеряно. Поэтому мы сначала изменим форму описания языка и лишь потом переделаем компилятор в соответствии с новым описанием языка правильных арифметических формул.

Прежде всего заметим, что в соответствии с грамматикой (**) формула состоит из одного или нескольких термов, соединенных знаками + или —, а терм состоит из одного или нескольких множителей, соединенных знаками * или /. Введем новые понятия (метасимволы)

⟨аддитивная операция⟩ ::= + | —
⟨мультипликативная операция⟩ ::= * | /

и зададим понятия ⟨формула⟩ и ⟨терм⟩ в новой форме:

⟨формула⟩ ::= ⟨терм⟩ {⟨аддитивная операция⟩ ⟨терм⟩}
⟨терм⟩ ::= ⟨множитель⟩ {⟨мультипликативная операция⟩ ⟨множитель⟩}

где фигурные скобки означают повторение содержимого нуль или более раз. Остальные понятия грамматики (**) оставим без изменений. При реализации компилятора в соответствии с новым описанием языка паре фигурных скобок будет соответствовать цикл. Программа «обработать формулу», например, примет вид:

```
программа обработать формулу
. дано     : P : последовательность элементов типа символ
.          | непрочитанная часть исполнителя P начинается с
.          | правильной арифметической формулы
. получить: | из непрочитанной части P прочитана правильная
.          | формула максимальной длины. Напечатан соответст-
.          | вующий фрагмент программы для Стекового
.          | калькулятора
. --------
. обработать терм
. цикл выполнять
. . выбор
. . . при P.нет непрочитанных элементов ⇒ выход из цикла
. . . при P.очередной элемент = "+" ⇒
. . .                 P.пропустить очередной элемент
. . .                 обработать терм
. . .                 печатать ⟨"СК.сложить"⟩
. . . при P.очередной элемент = "—" ⇒
. . .                 P.пропустить очередной элемент
. . .                 обработать терм
. . .                 печатать ⟨"СК.вычесть"⟩
. . . иначе ⇒ выход из цикла
. . конец выбора
. конец цикла
конец программы
```

Аналогичным образом модифицируется программа «обработать терм». Заметим, что, хотя теперь программа «обработать формулу» явно к себе не обращается, неявная рекурсия осталась: программа «обработать формулу» вызывает программу «обработать терм», которая вызывает программу «обработать множитель», которая вызывает исходную программу «обработать формулу».

При компиляции по новой программе формула a — b — c будет обработана правильно. Действительно, эта формула состоит из трех термов: a, b и c. Вначале будет обработан первый из них, а потом в цикле два других. Соответственно до цикла будет напечатано

```
СК.добавить ⟨a⟩ в стек
```

А в цикле сначала будет напечатано

```
СК.добавить ⟨b⟩ в стек
СК.вычесть
```

а потом

```
СК.добавить ⟨c⟩ в стек
СК.вычесть
```

Компиляция и интерпретация. Компилятор — это переводчик, который переводит текст с одного языка на другой. Результатом работы компилятора формул является текст программы, а не результат ее выполнения. Мы могли бы, однако, поступить по-другому: вместо того чтобы печатать программу для Стекового калькулятора, можно было выполнять соответствующие предписания по мере их появления, т. е. вместо строк вида

```
печатать ("СК.начать работу")
печатать ("СК.сложить")
```

написать соответственно

```
СК.начать работу
СК.сложить
```

В этом случае результатом работы программы был бы не текст, а горящая на табло Стекового калькулятора величина, вычисленная в соответствии с заданной формулой. Такие программы называются *интерпретаторами*, а процесс их выполнения — *интерпретацией*. В процессе интерпретации правильной арифметической формулы интерпретатор читает формулу, разбирает ее и по мере разбора выполняет нужные действия.

И компиляция, и интерпретация формулы имеют и преимущества, и недостатки. Например, если нам нужно много раз считать по одной и той же формуле с разными числовыми данными, то выгодно эту формулу заранее откомпилировать, а затем считать по готовой программе. Если же нам нужно посчитать по формуле всего один раз, то выгоднее сразу начать ее интерпретировать — при этом не понадобится получать и хранить программу для Стекового калькулятора. Это, конечно, весьма грубое описание. Более подробно на достоинствах и недостатках компиляции и интерпретации мы остановимся в разд. 20.

Реализация компилятора с языка арифметических формул с помощью стека. Будем рассматривать результат компиляции — про-

грамму для Стекового калькулятора — как функцию на пространстве последовательностей символов. Поскольку мы имеем дело с функцией на пространстве последовательностей, попробуем вычислить ее индуктивно за один проход.

Прежде всего заметим, что любую правильную формулу можно скомпилировать так, что

а) строки с именами переменных (предписания «добавить в стек») в программе будут идти в том же порядке, что и имена переменных в формуле;

б) все операции (сложить/вычесть/умножить/разделить) в программе будут расположены позже соответствующих знаков операций в формуле.

Этот факт легко доказывается индукцией по числу знаков операций в формуле. Для формулы без знаков операций это, очевидно, так. Предположим, что это верно для любой формулы, в которой не больше n знаков операций. Докажем, что тогда это верно и для формулы с n + 1 знаками операций. Действительно, выделим в такой формуле тот знак операции, который при вычислении по этой формуле выполняется последним (если таких знаков несколько, то возьмем любой из них, например самый правый). Возможны две ситуации:

⟨правильная формула x⟩ ⟨знак⟩ ⟨правильная формула y⟩

либо

(...(⟨правильная формула x⟩ ⟨знак⟩ ⟨правильная формула y⟩)...).

В любом случае формулы x и y содержат не более n знаков операций и по предположению индукции могут быть скомпилированы во фрагменты программы X и Y, удовлетворяющие условиям а) и б). Записав фрагмент программы X, затем фрагмент Y, а за ними операцию Стекового калькулятора, соответствующую знаку, мы получим фрагмент программы, удовлетворяющий условиям а) и б), для формулы в целом.

Таким образом, формулу можно компилировать так: встретив имя переменной, немедленно печатать соответствующую строку программы, а встретив знак операции или скобку, печатать те из предыдущих, уже прочитанных, но еще невыполненных операций (будем их называть *отложенными*), которые выполнимы в этот момент, после чего «откладывать» и новый знак z. Поскольку среди оставшихся отложенных операций нет таких, которые выполнимы до z, то для хранения отложенных операций можно воспользоваться стеком (назовем его *стеком операций*). Этот стек и есть та информация, которая необходима для индуктивной компиляции формулы.

исполнитель Компилятор формул
. **СП:**
. 1. компилировать правильную формулу ⟨**вх** : P⟩
. — обработать символ ⟨**вх** : c : **символ**⟩
. — обработать отложенные операции ⟨**вх** : c : **символ**⟩

```
. типы:
.   вид = перечисление (имя, левая скобка, правая скобка, знак)
. объекты:
.   стек операций : стек элементов типа символ
. используемые исполнители:
.   Справки
конец описаний |--------------------------------

программа компилировать правильную формулу ⟨вх : Р⟩
. дано      :  Р : последовательность элементов типа символ
.              | Р содержит правильную формулу
. получить: | Р откомпилирована, т. е. напечатана соответствующая
.           | программа для Стекового калькулятора
.---------
. печатать ⟨"СК.начать работу"⟩
. стек операций.начать работу
```

В стеке операций в каждый момент времени мы будем хранить отложенные и не выполненные к этому моменту операции. В момент окончания формулы все эти отложенные операции надо выполнить. Аналогично в момент прочтения правой скобки надо выполнить все операции, которые были отложены с момента появления соответствующей левой скобки. Чтобы не разбирать окончание формулы особо, мы возьмем всю формулу в скобки (точнее, проимитируем такое «взятие в скобки»):

```
. обработать символ ⟨вх: "("⟩
. Р.встать в начало последовательности
. цикл для каждого с из непрочитанной части Р выполнять
. .  обработать символ ⟨вх : с⟩
. конец цикла
. обработать символ ⟨вх : ")"⟩
.   утв: стек операций.стек пуст
. стек операций.кончить работу
. печатать ⟨"СК.показать результат"⟩
. печатать ⟨"СК.кончить работу"⟩
конец программы

программа обработать символ ⟨вх : с : символ⟩
. дано      :
. получить:
.---------
. вид: = Справки.вид символа ⟨с⟩
. выбор
. . при вид = имя ⇒ печатать ⟨"СК.добавить ⟨", с, "⟩ в стек"⟩
. . при вид = левая скобка ⇒ стек операций.добавить ⟨с⟩
```

```
. . при вид = знак ⇒ обработать отложенные операции ⟨с⟩
. .                  стек операций.добавить ⟨с⟩
. . при вид = правая скобка ⇒
. .                  обработать отложенные операции ⟨с⟩
. .                    утв: стек операций.вершина = "("
. .                  стек операций.удалить вершину стека
. конец выбора
конец программы

программа обработать отложенные операции ⟨вх : с : символ⟩
. дано     : (Справки.вид символа ⟨с⟩ = знак или
.              Справки.вид символа ⟨с⟩ = правая скобка) и
.            стек операций.стек не пуст | там, как минимум, "("
. получить: | операции из стека операций, которые можно выпол-
.           | нить до "с", выполнены.
.--------
. цикл пока Справки.⟨стек операций.вершина⟩ выполнима до ⟨с⟩
. . выполнять
. .  стек операций.взять элемент из стека в ⟨вых : х⟩
. .  выбор
. .  . при х = "+" ⇒ печатать ⟨"СК.сложить"⟩
. .  . при х = "-" ⇒ печатать ⟨"СК.вычесть"⟩
. .  . при х = "*" ⇒ печатать ⟨"СК.умножить"⟩
. .  . при х = "/" ⇒ печатать ⟨"СК.разделить"⟩
. .  . иначе ⇒ отказ
. .  конец выбора
. конец цикла
конец программы

конец исполнителя |==========================

исполнитель Справки
  СП:
.  1. вид символа ⟨вх : с : символ⟩                    : вид
.  2. ⟨вх : с1 : символ⟩ выполнима до ⟨вх : с2 : символ⟩  : да/нет
.  — вес ⟨вх : с : символ⟩                             : 1..2
.
. типы:
.   вид = перечисление (имя, левая скобка, правая скобка, знак)
конец описаний |-------------------------------

программа вид символа ⟨вх : с : символ⟩ : вид
. дано     : | с — символ из правильной формулы
. получить:
.--------
```

```
. выбор
. .   при с = "+" или с = "—" или с = "*" или с = "/"
. .                          ⇒ ответ := знак
. .   при с = "("            ⇒ ответ := левая скобка
. .   при с = ")"            ⇒ ответ := правая скобка
. .   при "a" ≤ с ≤ "z"      ⇒ ответ := имя
. .   иначе ⇒ отказ
. конец выбора
конец программы

программа ⟨вх : с1 : символ⟩ выполнима до ⟨вх : с2⟩ : да/нет
. дано     : (вид символа ⟨с1⟩ = знак или
.              вид символа ⟨с1⟩ = левая скобка) и
.            (вид символа ⟨с2⟩ = знак или
.             вид символа ⟨с2⟩ = правая скобка)
. получить:
.--------
. выбор
. .   при вид символа ⟨с1⟩ = левая  скобка ⇒ ответ := нет
. .   при вид символа ⟨с2⟩ = правая скобка ⇒ ответ := да
. .   иначе ⇒ ответ := (вес ⟨с1⟩ ≥ вес ⟨с2⟩)
. конец выбора
конец программы

программа вес ⟨вх : с : символ⟩ : 1..2
. дано     : вид символа ⟨с⟩ = знак
. получить:
.--------
.  выбор
. .   при с = "+" ⇒ ответ := 1
. .   при с = "—" ⇒ ответ := 1
. .   при с = "*" ⇒ ответ := 2
. .   при с = "/" ⇒ ответ := 2
. конец выбора
конец программы

конец исполнителя |==========================
```

Заметьте, насколько наша новая реализация сложнее рекурсивной. Однако в отличие от рекурсивной, где простейшие изменения требуют существенных творческих усилий или даже переписывания всей программы целиком, здесь у нас изменение порядка операций с обычного на «справа налево» требует замены всего одного знака — в программе «выполнима до» надо вместо ≥ написать >. Даже более сложная задача — переделать компилятор так,

чтобы сложение выполнялось справа налево, а вычитание — слева направо, — требует добавления лишь одной строки без какого-либо изменения структуры всей реализации.

В заключение отметим, что мы компилировали только правильные формулы. В реальной жизни от компилятора требуется, чтобы он обрабатывал не только цепочки языка, но и любые другие (не принадлежащие языку) последовательности символов, сообщая, почему эти последовательности не принадлежат языку



Рис. 10.4. а) последовательность символов, б) результат компиляции

(рис. 10.4) (в данном случае под символом понимается предопределенный тип нашего языка программирования, а не элемент алфавита языка арифметических формул). Анализ правильности формулы может быть проведен так, как это было сделано в разд. 8.

ЗАДАЧИ И УПРАЖНЕНИЯ

1. Задайте с помощью грамматики язык

а) десятичных целых чисел со знаком,

б) правильных программ для Стекового калькулятора (т. е. язык всех программ, которые получаются при компиляции всех правильных формул; символами алфавита в данном случае следует считать отдельные строчки программы).

2. Докажите, что цепочка $(a*(b+c)+d)/(a+d)$ является правильной арифметической формулой

а) в грамматике (*),

б) в грамматике (**)

(доказательство состоит в придумывании дерева или последовательности вывода).

Задачи 3—5 относятся к реализации компилятора с помощью стека.

3. Нарисуйте последовательные состояния стека операций при компилировании формулы $(a*(b+c)-d)/(a+d)$.

4. Измените Компилятор формул так, чтобы сложение выполнялось справа налево, а остальные операции — слева направо.

5. Измените Компилятор формул так, чтобы в формуле в произвольных местах могло стоять произвольное количество пробелов и чтобы эти пробелы полностью игнорировались при компиляции.

11. Проект «Построение изображения полиэдра»

В настоящее время во многих областях науки и техники используются разные методы моделирования геометрических объектов в трехмерном пространстве. Один из способов моделирования состоит в том, чтобы аппроксимировать реальный объект набором выпуклых плоских многоугольников (рис. 11.1). Такой набор мы

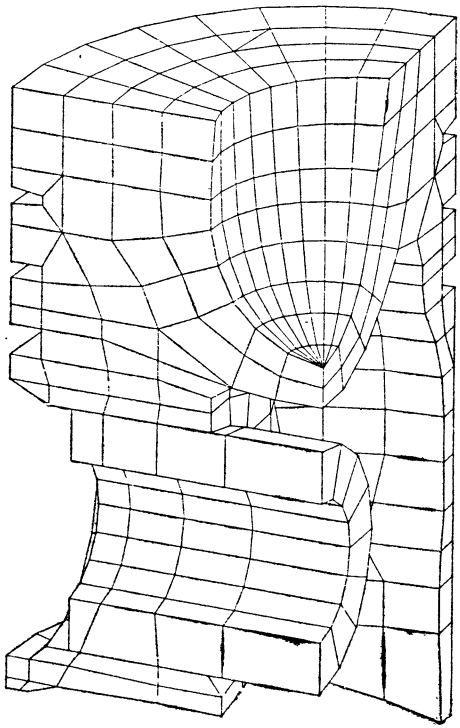


Рис. 11.1

будем называть полиэдром, многоугольники — гранями, а их стороны — ребрами полиэдра. Как мы видим, при изображении полиэдра реально изображаются не сами многоугольники, а линии, их ограничивающие, т. е. ребра. Однако если изобразить все ребра вообще, то получится запутанная, непривычная для человека картинка (рис. 11.2).

Для получения удобного для человека рис. 11.1 нужно при построении изображения учитывать, что некоторые ребра могут оказаться полностью или частично невидимыми, так как их могут

загородить грани полиэдра. Таким образом, при построении изображения надо «удалить» (т. е. не рисовать) части ребер, которые

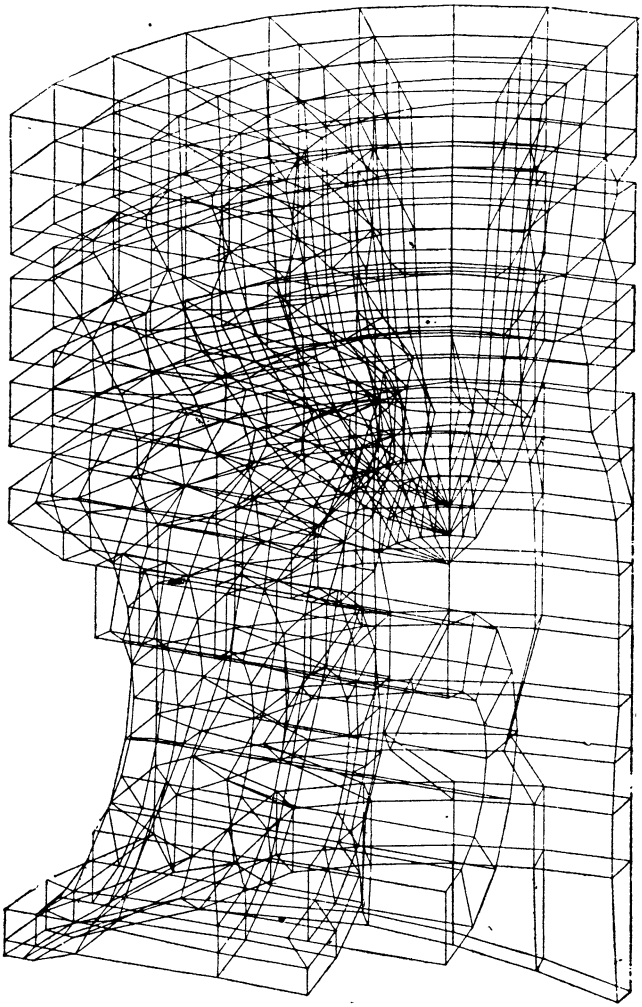


Рис. 11.2

не видны. Поэтому в программистской литературе эта задача часто называется задачей «удаления невидимых линий». Ею мы и займемся в настоящем разделе.

Постановка задачи. Пусть заданы полиэдр, расположенный в трехмерном пространстве, и направление проектирования (некото-

рый вектор). Мы будем представлять себе этот вектор направленным вверх, параллельные ему прямые и плоскости называть *вертикальными*, а ортогональные — *горизонтальными*. Задача состоит в том, чтобы построить вид полиэдра из «бесконечно высокой» точки с учетом того, что одни грани могут загораживать другие, или,

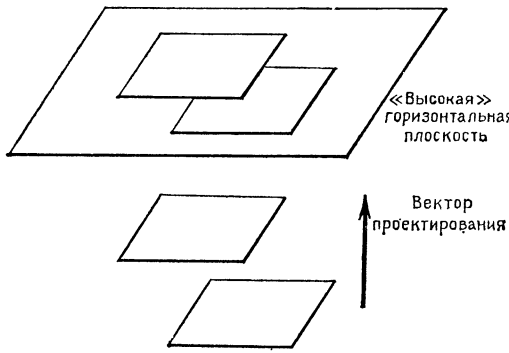


Рис. 11.3

другими словами, построить плоское изображение видимой части полиэдра (параллельная проекция вдоль вектора проектирования) на «достаточно высокой» горизонтальной плоскости (рис. 11.3).

Примерами полиэдров могут служить поверхности куба или пакета молока с отрезанным углом, полураскрытая веером книга и даже набор пересекающихся многоугольников (рис. 11.4).

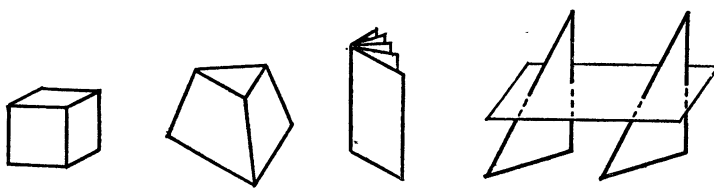


Рис. 11.4

Простейший способ задать полиэдр — перечислить координаты вершин каждой грани. На практике, однако, этот способ не применяется, так как он обладает рядом недостатков. Поэтому мы поступим следующим образом: занумеруем все вершины полиэдра и будем считать, что каждое ребро полиэдра задается парой номеров — номером начала и номером конца. Далее для задания грани перечислим все ребра, ограничивающие данную грань (т. е. все пары номеров вершин). Описанная информация задает так называемый *абстрактный* полиэдр и позволяет отвечать на комбина-

торные вопросы вида «сколько вершин в данной грани?», «имеют ли две данные грани общую вершину?» и т. д. Эта комбинаторная информация, однако, ничего не говорит о расположении полиэдра в трехмерном пространстве. Чтобы задать такое расположение, достаточно задать отображение множества «абстрактных» вершин (т. е. множества их номеров) в R3. Эта часть информации о полиэдре далее называется *метрической* (числовой).

Будем считать, что и комбинаторная, и метрическая информация о конкретном полиэдре, изображение которого надо построить, содержится в базовом исполнителе «Полиэдр» и может быть получена с помощью следующих предписаний:

1. множество граней :: **множество элементов типа** грань
2. множество ребер :: **множество элементов типа** ребро
3. R3 ⟨**вх** : вершина⟩ : точка R3 | отображение вершин в R3

типы:

грань = **множество элементов типа** ребро
ребро = **запись** (начало, конец : вершина)
вершина = Z+
точка R3 = **запись** (x, y, **z : число**)

Информация о векторе проектирования может быть получена с помощью предписания

1. вектор проектирования :: вектор R3

базового исполнителя «Проектор». Этот же исполнитель по предписанию

2. изобразить отрезок ⟨**вх** : отрезок R3⟩

проектирует отрезок вдоль вектора проектирования и изображает его. Здесь

типы:

отрезок R3 = **запись** (начало, конец : точка R3)
точка R3, вектор R3 = **запись** (x, y, **z : число**)

Таким образом, наша задача — реализовать программу построения изображения полиэдра на базе исполнителей «Полиэдр» и «Проектор».

Реализация. Поскольку плоское изображение полиэдра состоит из проекций видимых частей ребер, то естественно строить его следующим образом: перебрать все ребра полиэдра и для каждого ребра изобразить его видимую часть. Мы здесь опять воспользуемся аналогией со светом — расположим «бесконечно высоко вверху» солнце и будем называть видимые части ребра *просветами*, а часть, загороженную некоторой гранью, — *тенью от грани на ребро* (рис. 11.5).

Для получения изображения видимой части ребра можно поступить так: перебрать все грани полиэдра, для каждой грани построить ее тень на ребро, посмотреть, что останется не затененным, и эти просветы изобразить.

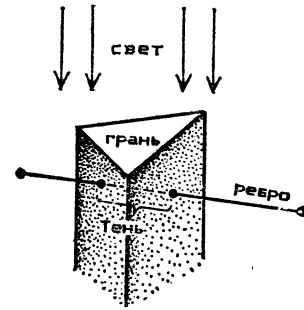


Рис. 11.5

Поскольку мы сначала перебираем все ребра, а потом для каждого ребра перебираем все грани, то самым крупным образованием в этой ситуации является ребро. Поэтому в соответствии с технологией «сверху вниз» примем решение, что все действия с конкретным ребром в целом будет производить специальный промежуточный исполнитель «Изобразитель ребра». Тогда первый шаг декомпозиции можно записать так:

программа построение изображения полиэдра
. **дано** : | полиэдр в R3 (исполнитель "Полиэдр")
. | направление проектирования (исполнитель "Проектор")
. **получить:** | плоское изображение ребер полиэдра (параллельная
. | проекция) с "удалением невидимых линий"
.---------
. **цикл** ∀ ребро ∈ Полиэдр.множество ребер **выполнять**
. . Изобразитель ребра.начать работу ⟨**вх**: ребро⟩
. . **цикл** ∀ грань ∈ Полиэдр.множество граней **выполнять**
. . . Изобразитель ребра.учесть тень от грани ⟨**вх**: грань⟩
. . **конец цикла**
. . Изобразитель ребра.кончить работу
. **конец цикла**
конец программы

Обратите внимание, что ребро, с которым будет работать Изобразитель ребра, указывается только в предписании «начать работу». Мы предполагаем, что Изобразитель ребра сам запомнит это ребро и потом использует при учете тени от грани. Мы предполагаем также, что по предписанию «кончить работу» Изобразитель ребра изображает оставшиеся на ребре просветы и лишь затем заканчивает работу.

Теперь надо формализовать понятие «тень от грани на ребро» и понять как Изобразитель ребра будет по этим теням строить просветы. Первая идея состоит в том, что поскольку каждая тень и каждый просвет — это некоторые отрезки на ребре, то можно ввести одномерные координаты на ребре и работать с отрезками

в этих одномерных координатах, а не с отрезками в R3. Такой переход от трехмерных объектов к одномерным заметно упрощает работу. В соответствии с технологией «сверху вниз» задачу получения «одномерных» теней мы возложим на специального исполнителя «Геометр», а все операции над отрезками в одномерных координатах — на исполнителя «Отрезки». Вторая идея состоит в том, что если ребро считать фиксированным, то просветы можно рассматривать как индуктивную функцию от последовательности учитываемых граней, а точнее их теней. При получении очередной

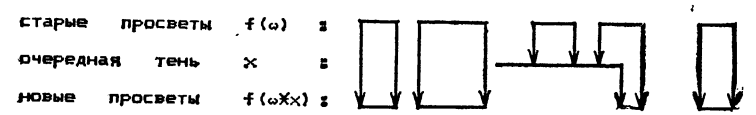


Рис. 11.6

тени x надо просмотреть все имеющиеся просветы f(ω) и исключить из них части, попавшие в тень. Оставшиеся после этого просветы и есть f(ω * x) (рис. 11.6).

В начальный момент (до получения первой тени от грани) просветы f(Δ) должны состоять из одного отрезка, соответствующего всему ребру.

Нам будет удобно хранить просветы в линейном однонаправленном списке (Л1-списке) и модифицировать этот список при получении очередной тени:

```
исполнитель Изобразитель ребра
. СП:
.   1. начать работу ⟨вх: ребро⟩
.   2. учесть тень от грани ⟨вх: грань⟩
.   3. кончить работу
.
. типы:
.  отрезок = ... | отрезок на изображаемом ребре
. объекты:
.  Просветы: Л1-список элементов типа отрезок
.
. идеи реализации:
.  Список "Просветы" хранит просветы между тенями от уже уч-
.  тенных (с момента начала работы) граней
.
. используемые исполнители:
.  Геометр, Проектор, Отрезки
конец описаний |----------------------------------
```

```
программа начать работу ⟨вх: ребро⟩
. дано      :  ребро: ребро | "абстрактное" ребро полиэдра
. получить: | Просветы состоят из всего ребра
.---------
.  Геометр.начать работу ⟨вх: ребро,
.                                  Проектор.вектор проектирования⟩
.  Просветы.начать работу
.  Просветы.добавить элемент ⟨вх: Геометр.ребро1⟩ за указателем
конец программы
```

Здесь «ребро1» — предписание Геометра, которое вырабатывает значение типа «отрезок», задающее все ребро в одномерных координатах. Использование этого предписания позволяет в Изобразителе ребра не знать конкретных одномерных координат начала и конца ребра.

```
программа учесть тень от грани ⟨вх: грань⟩
. дано      :  грань: грань | "абстрактная" грань полиэдра
. получить: | в списке "Просветы" учтена тень от новой грани
.---------
.  тень := Геометр.тень от грани ⟨вх: грань⟩ | в одномерн.коорд.
.  если Отрезки.отрезок ⟨тень⟩ вырожден то ничего не делать
.  . иначе
.  . Просветы.установить указатель в начало списка
.  . цикл пока Просветы.указатель не в конце списка выполнять
.  . .  Просветы.взять элемент за указателем в ⟨вых: просвет⟩
.  . .  Отрезки.получить разность ⟨вх: просвет, тень⟩
.  . .                  в ⟨вых: левая компонента, правая компонента⟩
.  . .  если Отрезки.отрезок ⟨левая компонента⟩ не вырожден то
.  . .  . Просветы.добавить ⟨левая компонента⟩ за указателем
.  . .  . Просветы.передвинуть указатель вперед
.  . .  конец если
   . .  если Отрезки.отрезок ⟨правая компонента⟩ не вырожден
   . .  . то
.  . .  . Просветы.добавить ⟨правая компонента⟩ за указателем
.  . .  . Просветы.передвинуть указатель вперед
.  . .  конец если
.  . конец цикла
.  конец если
конец программы
```

Чтобы понять работу этой программы, надо разобраться в том, какой отрезок называется вырожденным и что такое разность двух отрезков.

Отрезком с началом a и концом b мы называем множество точек $t \in R$ таких, что $a \leqslant t \leqslant b$. Тем самым при $a = b$ отрезок

сводится к точке, а при $a > b$ является пустым. В обоих случаях (т. е. при $a \geqslant b$) отрезок называется *вырожденным*.

Предписание «получить разность ⟨вх: A, B⟩...» исполнителя «Отрезки» получает разность множества точек отрезка A и множества внутренних точек отрезка B, например, как изображено на рис. 11.7.



Рис. 11.7

В зависимости от расположения отрезков «просвет» и «тень» разность может быть пустой (если просвет целиком попал в тень) или состоять из одной точки (рис. 11.8), из одного отрезка, двух точек, отрезка и точки, двух отрезков.

Нам, однако, будет удобнее считать, что разность *во всех случаях состоит из двух отрезков*, но один из них или они оба могут

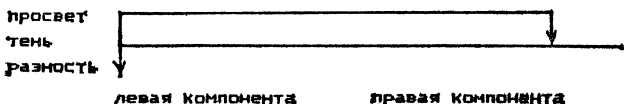


Рис. 11.8

оказаться вырожденными (одноточечными или пустыми). Например, если просвет и тень не пересекаются, то в результате одна из компонент разности будет совпадать с просветом, а другая будет вырожденной. Если просвет целиком попадает в тень, то обе компоненты разности будут вырожденными.

```
программа кончить работу
. дано      :
. получить: | изображение просветов (т. е. видимой части ребра)
. --------
. Просветы.установить указатель в начало списка
. цикл ∀ x ∈ части списка Просветы за указателем выполнять
. . Проектор.изобразить отрезок ⟨Геометр.R3⟨вх: x⟩⟩
. конец цикла
. Просветы.кончить работу
. Геометр.кончить работу
конец программы
```

Здесь «Геометр.R3⟨вх : x⟩» — это предписание, которое по отрезку x в одномерных координатах на ребре вырабатывает значение, являющееся отрезком в трехмерном пространстве.

конец исполнителя |==========

Итак, мы совершили очередной шаг декомпозиции и нам осталось реализовать исполнителей «Геометр» и «Отрезки». Поскольку мы подозреваем, что Геометр в иерархии исполнителей будет занимать более высокое положение (так как при его реализации могут потребоваться какие-то операции с отрезками), то мы сначала реализуем его. Система предписаний этого исполнителя выглядит так:

1. начать работу ⟨вх : ребро, вектор проектирования⟩
2. ребро1 : отрезок
3. тень от грани ⟨вх : грань⟩ : отрезок
4. R3 ⟨вх : отрезок⟩ : отрезок R_3
5. кончить работу

Поскольку Геометр информацию об обрабатываемом ребре и векторе проектирования получает лишь в предписании «начать работу», а эта информация необходима и в других предписаниях (например, при нахождении тени от грани), то мы введем глобальные объекты

объекты:

x0, x1: точка R3 | начало и конец обрабатываемого ребра
p : вектор R3 | вектор проектирования

в которых при начале работы запомним соответствующую информацию.

Обратите внимание, что мы различаем точки и векторы трехмерного пространства. Две точки можно вычесть и получить вектор, к точке можно прибавить вектор и снова получить точку и т. д. Прибавить точку к точке или умножить точку на число нельзя. Подобное разграничение позволяет контролировать правильность формул в аналитической геометрии подобно тому, как учет размерностей величин позволяет проводить контроль физических формул. Все операции над точками и векторами R3 мы будем выполнять с помощью специального исполнителя «Стереометр». Именно он будет сводить эти операции к операциям, которые умеет выполнять Универсальный Выполнитель. Например, для векторов x и y мы можем написать x := y, но вместо x := —y должны писать

x := Стереометр.произведение ⟨y⟩ на ⟨—1⟩.

Предписания «ребро1» и «тень от грани» должны вырабатывать отрезок в одномерных координатах на ребре. Мы будем обо-

значать точки ребра в одномерных координатах буквой t, а сами координаты называть *t-координатами.* По одномерной координате t точка x на ребре с началом x0 и концом x1 вычисляется следующим образом:

$$x = (x1 - x0) * (t - t0)/(t1 - t0) + x0,$$

где t0 и t1 — t-координаты начала и конца ребра.

Основным и наиболее содержательным предписанием Геометра является предписание «тень от грани». Именно в процессе реализации этого предписания были введены локальные предписания, перечисленные в системе предписаний через дефис (-):

```
исполнитель Геометр
. СП:
.   1. начать работу ⟨вх: ребро, вектор проектирования⟩
.   2. ребро 1                                          : отрезок
.   3. тень от грани ⟨вх: грань⟩                        : отрезок
.   — грань ⟨вх: грань⟩ вертикальна                     : да/нет
.   — центр ⟨вх: грань⟩                                 : точка R3
.   — горизонтальное пересечение ⟨вх: a, b, c: точка R3⟩ : отрезок
.   — вертикальное пересечение ⟨вх: a, b, c: точка R3⟩  : отрезок
.   — пересечение с ⟨вх: a: точка R3, n: вектор R3⟩     : отрезок
.
.   4. R3 ⟨вх: отрезок⟩                                 : отрезок R3
.   — R3_образ ⟨вх: t: число⟩                           : точка R3
.
.   5. кончить работу
.
. константы:
.   t0      = 0.0 | одномерная координата начала ребра
.   t1      = 1.0 | одномерная координата конца ребра
.
. типы:
.  отрезок R3 = запись (начало, конец: точка R3)
.  точка R3, вектор R3 = ...
.  отрезок  = запись (начало, конец: число)
.  грань    = множество элементов типа ребро
.  ребро    = запись (начало, конец: вершина)
.  вершина  = ...
.
. объекты:
.  x0, x1: точка R3 | начало и конец обрабатываемого ребра
.  p     : вектор R3 | вектор проектирования
```

```
•
. используемые исполнители:
. Полиэдр, Отрезки, Стереометр (С)
конец описаний | ----------------------------------

программа начать работу ⟨вх: ребро, вектор проектирования⟩
. дано      : вектор проектирования: вектор R3,
•               ребро: ребро |"абстрактное" ребро полиэдра
. получить: | установлены соотв. значения глобальных объектов
. --------
.  x0 := Полиэдр.R3 ⟨ребро.начало⟩
.  x1 := Полиэдр.R3 ⟨ребро.конец⟩
.  p := вектор проектирования
конец программы

программа ребро1: отрезок ==(t0, t1)
```

Итак, мы подошли к центральному месту реализации Геометра — нахождению тени от грани на ребро. Прежде всего разберемся с тем, что такое тень. Будем в рамках нашей аналогии считать, что на грань сверху падает параллельный вектору p поток света. Множество затеняемых гранью точек пространства образует при этом полубесконечную призму, у которой основанием является грань полиэдра, а боковая поверхность параллельна вектору p и направлена в противоположную сторону (рис. 11.5). Тень от грани на ребро есть пересечение ребра с этой призмой:

тень = ребро ∩ призма

(Здесь и далее мы будем считать, что грань не вертикальна, т. е. не параллельна вектору p.) Заметим, что призму следует считать открытой, т. е. считать, что граница призмы тени не принадлежит, так как в противном случае каждая грань будет затенять себя и изображение окажется пустым.

Естественно попытаться свести нахождение пересечения ребра с призмой к каким-нибудь более элементарным операциям. Мы сделаем это, представив призму в виде пересечения открытых полупространств. Например, треугольную призму можно представить в виде пересечения четырех полупространств (рис. 11.9).

Первое полупространство ограничено плоскостью, проходящей через грань полиэдра, и расположено со стороны, противоположной направлению вектора p; это полупространство мы будем называть *горизонтальным*. Остальные полупространства ограничены вертикальными плоскостями, проходящими через то или иное ребро грани, и расположены так, что сама грань (или, что то же самое, ее центр)

содержится в соответствующем полупространстве; эти полупространства мы будем называть *вертикальными*. Итак,

призма = Г ∩ В1 ∩ В2 ∩ В3,

где Г — горизонтальное полупространство, а В1, В2, В3 — вертикальные. Следовательно,

тень = ребро ∩ призма = ребро ∩ Г ∩ В1 ∩ В2 ∩ В3 =
= РГ ∩ РВ1 ∩ РВ2 ∩ РВ3,

где РГ = ребро ∩ Г, а РВi = ребро ∩ Bi, i = 1, 2, 3. И РГ, и РВi являются отрезками (точнее интервалами или полуинтервалами) на

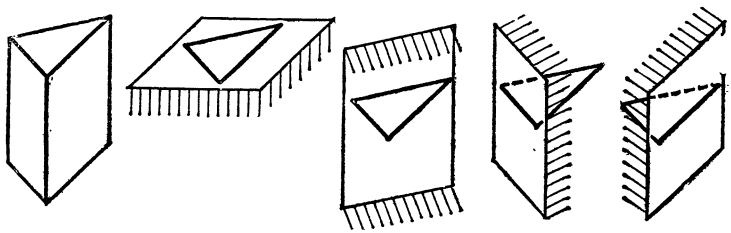


Рис. 11.9. а) призма, б) четыре полупространства, пересечение которых дает призму

ребре; мы их будем называть соответственно *горизонтальным* и *вертикальными пересечениями*.

Таким образом, мы можем отдельно находить пересечения ребра с каждым из полупространств, а потом пересечь получившиеся отрезки на ребре между собой. Поскольку грань может быть многоугольной, естественно сначала найти горизонтальное пересечение, а оставшиеся вертикальные обрабатывать в цикле:

```
программа тень от грани (вх: грань): отрезок
. дано      : грань: грань | "абстрактная" грань полиэдра
. получить: | затеняемый гранью отрезок ребра в t-координатах
.           | (или пустой отрезок, если грань вертикальна)
. --------
.  если грань (вх: грань) вертикальна то ответ := (t1, t0)
.  . иначе
.  .  с := центр (вх: грань)
.  .  грань.скопировать какой нибудь элемент в (вых: ребро)
.  .  а := Полиэдр.R3 (ребро.начало)
.  .  b := Полиэдр.R3 (ребро.конец)
.  .  ответ := горизонтальное пересечение (вх: a, b, c)
.  .  цикл ∀ ребро ∈ множества грань выполнять
.  .  .  а := Полиэдр.R3 (ребро.начало)
```

```
. . .  b := Полиэдр.R3 ⟨ребро.конец⟩
. . .  z := вертикальное пересечение ⟨вх: a,b,c⟩
. . .  Отрезки.⟨вх/вых: ответ⟩ пересечь с ⟨вх: z⟩
. .  конец цикла
.  конец если
конец программы
```

При реализации этой программы мы в соответствии с технологией «сверху вниз» отложили «на потом» всю содержательную работу (аналитическую геометрию), для чего воспользовались четырьмя локальными программами.

Программа «грань вертикальна» проверяет, параллельна ли грань вектору проектирования. Программа «центр» предназначена для получения какой-нибудь внутренней точки грани. В качестве такой точки можно взять, например, центр тяжести вершин грани. Эти две программы реализуйте самостоятельно.

Оставшиеся две программы находят пересечение ребра с горизонтальным или вертикальным полупространством соответственно. Начало и конец ребра, а также вектор проектирования являются глобальными объектами Геометра, а полупространства задаются в виде трех точек трехмерного пространства.

```
программа горизонтальное пересечение ⟨вх: a,b,c⟩: отрезок
.  дано     :  a,b,c: точка R3  | три точки плоскости грани,
.                                | не лежащие на одной прямой
.  получить: | пересечение (в t-координатах ) ребра (x0,x1) с
.           | полупространством, ограниченным плоскостью abc и
.           | направленным в сторону вектора — p
. --------
.  ab := C.разность ⟨вх: b,a⟩   | вектор ab := b — a   (точки в R3)
.  ac := C.разность ⟨вх: c,a⟩   | вектор ac := c — a   (точки в R3)
.  n := C.векторное произведение ⟨ab,ac⟩        | выбор нормали к
.  если C.скалярное произведение ⟨n,p⟩ < 0.0    | плоскости abc,
.  . то n := C.произведение ⟨n⟩ на ⟨—1⟩         | сонаправленной
.  конец если                                  | с вектором p
.  ответ := пересечение с ⟨вх: a,n⟩
конец программы

программа вертикальное пересечение ⟨вх: a,b,c⟩: отрезок
.  дано     :  a,b,c: точка R3  | три точки плоскости грани,
.                                | не лежащие на одной прямой
.  получить: | пересечение (в t-координатах) ребра (x0,x1) с
.           | полупространством, которое содержит точку c и
.           | ограничено вертикальной (параллельной p)
.           | плоскостью, проходящей через точки a,b
. --------
```

```
. ab := С.разность ⟨вх : b,a⟩ | вектор ab := b — a (точки в R3)
. ac := С.разность ⟨вх : c,a⟩ | вектор ac := c — a (точки в R3)
. n := С.векторное произведение ⟨ab,p⟩          | выбор нормали
. если С.скалярное произведение ⟨n,ac⟩ > 0.0 | к плоскости,
. . то n := С.произведение ⟨n⟩ на ⟨—1⟩         | направленной
. конец если                                    | наружу от грани
. ответ := пересечение с ⟨вх : a,n⟩
конец программы
```

Итак, мы свели получение тени от грани на ребро к нахождению пересечения (в t-координатах) ребра и полупространства, заданного точкой a граничной плоскости и нормалью n к этой плоскости, направленной наружу полупространства (рис. 11.10). Для нахождения этого пересечения рассмотрим линейную неоднородную функцию

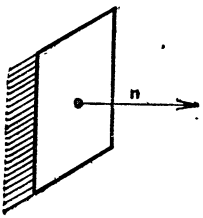


Рис. 11.10

L (x) = скалярное произведение ⟨n, x — a⟩,

которая обращается в нуль на граничной плоскости и возрастает вдоль вектора n. Тогда ограничивающее полупространство можно описать как множество тех точек $x \in R3$, в которых $L(x) < 0$, т. е. как область отрицательности функции L. В концах обрабатываемого ребра функция L принимает значения

L0 = L (x0) = скалярное произведение ⟨n, x0 — a⟩,
L1 = L (x1) = скалярное произведение ⟨n, x1 — a⟩.

Рассмотрим сужение этой функции на обрабатываемое ребро и подставим вместо x его выражение через t-координату на ребре. Вновь полученная функция (будем обозначать ее той же буквой L) определена на отрезке [t0, t1] и принимает на его концах значения L0 и L1. Поскольку эта функция является линейной неоднородной функцией, то ее значения в промежуточных точках отрезка можно вычислить по формуле

$$L(t) = (L1 - L0) * (t - t0)/(t1 - t0) + L0.$$

В частности, корень функции L(t) можно найти из соотношения

$$(L1 - L0) * (t - t0)/(t1 - t0) + L0 = 0.0,$$

т. е. $t := t0 + (0.0 - L0)/(L1 - L0) * (t1 - t0)$.

Таким образом, искомое пересечение обрабатываемого ребра с ограничивающим полупространством — это интервал отрицательности

функции L(t) на отрезке [t0, t1]. Пустой интервал мы будем кодировать парой (t1, t0):

```
программа пересечение с ⟨вх: a, n⟩: отрезок
. дано     : a: точка R3
.            n: вектор R3 | n≠0
. получить: | пересечение (в t-координатах) ребра (x0, x1) с
.           | полупространством, которое направлено в сторону
.           | вектора —n и ограничено плоскостью, с нормалью n
.           | проходящей через точку a
. --------
.  ax0 := С.разность ⟨вх: x0, a⟩
.  ax1 := С.разность ⟨вх: x1, a⟩
.  L0 := С.скалярное произведение ⟨n, ax0⟩ | т. е. L0 := (n, x0 — a)
.  L1 := С.скалярное произведение ⟨n, ax1⟩ | т. е. L1 := (n, x1 — a)
.  выбор
.  . при L0 ⩾ 0.0 и L1 ⩾ 0.0 ⇒ ответ := (t1, t0) | пусто
.  . при L0 < 0.0 и L1 < 0.0 ⇒ ответ := (t0, t1) | все ребро
.  . иначе ⇒
.  .  t := t0 + (0.0 — L0)/(L1 — L0) * (t1—t0) | корень L
.  .  выбор
.  .  . при L0 < 0.0 ⇒ ответ := (t0, t)
.  .  . при L1 < 0.0 ⇒ ответ := (t, t1)
.  .  . иначе ⇒ отказ
.  .  конец выбора
.  конец выбора
конец программы

программа R3 ⟨вх: отрезок⟩: отрезок R3 ==
     (R3_образ ⟨вх: отрезок.начало⟩, R3_образ ⟨вх: отрезок.конец⟩)
программа R3_образ ⟨вх: t: число⟩: точка R3
. дано     : t: число, t0 ⩽ t ⩽ t1 | t — координата точки на ребре
. получить: | координаты точки t в R3:
.           | x0 + (x1 — x0) * (t — t0)/(t1 — t0)
. --------
.  dx    := С.разность ⟨вх: x1, x0⟩
.  dxt   := С.произведение ⟨вх: dx⟩ на ⟨вх: (t — t0)/(t1 — t0)⟩
.  ответ := С.сумма ⟨вх: x0, dxt⟩
конец программы

программа кончить работу == ничего не делать

конец исполнителя | ==========================
```

Еще один шаг декомпозиции совершен, и теперь осталось реализовать исполнителей «Отрезки» и «Стереометр». Эти исполнители

не зависят друг от друга, и поэтому мы сначала реализуем то, что попроще — Отрезки:

```
исполнитель Отрезки
. СП:
.   1. отрезок ⟨вх: A: отрезок⟩ вырожден      : да/нет
.   2. получить разность ⟨вх: A, B: отрезок⟩ в ⟨вых: L, R: отрезок⟩
.   3. ⟨вх/вых: A: отрезок⟩ пересечь с ⟨вх: B: отрезок⟩
.
.   типы:
.    отрезок = запись (начало, конец: число)
.                | если A.начало > A.конец, то отрезок A пуст
конец описаний |----------------------------------->

программа отрезок ⟨вх: A: отрезок⟩ вырожден: да/нет ==
                                          (A.начало ⩾ A.конец)
программа получить разность ⟨вх: A, B⟩ в ⟨вых: L, R⟩
. дано      : A, B: отрезок
. получить: L, R: отрезок | левая (Left) и правая (Right) отно-
.           | сительно B компоненты разности A\B
. --------
. L.начало := A.начало
. L.конец  := минимум (A.конец, B.начало)
. R.начало := максимум (A.начало, B.конец)
. R.конец  := A.конец
конец программы

программа ⟨вх/вых: A: отрезок⟩ пересечь с ⟨вх: B: отрезок⟩
. дано      : A, B: отрезок
. получить: | A := A ∩ B
. --------
. A.начало := максимум (A.начало, B.начало)
. A.конец  := минимум (A.конец, B.конец)
конец программы
конец исполнителя |==============================
```

Убедиться в правильности программы «пересечь с» можно, либо разобрав все случаи на картинках, либо формально, следующим образом:

$$A = \{t \geqslant A.\text{начало}\} \cap \{t \leqslant A.\text{конец}\} \text{ (пересечение двух лучей)}$$
$$B = \{t \geqslant B.\text{начало}\} \cap \{t \leqslant B.\text{конец}\}$$

Значит,

$$A \cap B = \{t \geqslant A.\text{начало}\} \cap \{t \geqslant B.\text{начало}\} \cap \{t \leqslant A.\text{конец}\} \cap \{t \leqslant B.\text{конец}\} =$$
$$= \{t \geqslant \text{максимум } (A.\text{начало}, B.\text{начало})\} \cap \{t \leqslant \text{минимум } (A.\text{конец}, B.\text{конец})\}$$

И наконец, последний шаг декомпозиции — реализация Стереометра, т. е. сведение операций над точками и векторами в R3 к операциям над координатами этих точек и векторов:

```
исполнитель Стереометр
. СП:
.   1. разность ⟨вх: a, b: точка R3⟩                          : вектор R3
.   2. сумма ⟨вх: a: точка R3, b: вектор R3⟩                  : точка R3
.   3. произведение ⟨вх: a: вектор R3⟩ на ⟨вх: k: число⟩      : вектор R3
.   4. скалярное произведение ⟨вх: a, b: вектор R3⟩           : число
.   5. векторное произведение ⟨вх: a, b: вектор R3⟩           : вектор R3
.
. типы:
.   точка R3, вектор R3 = запись (x, y, z: число)
конец описаний |----------------------------------------------

программа разность ⟨вх: a, b: точка R3⟩: вектор R3 ==
                                  ((a.x — b.x), (a.y — b.y), (a.z — b.z))
программа сумма ⟨вх: a: точка R3, b: вектор R3⟩: точка R3 ==
                                  ((a.x + b.x), (a.y + b.y), (a.z + b.z))
программа произведение ⟨вх: a⟩ на ⟨вх: k: число⟩: вектор R3 ==
                                  ((k * a.x), (k * a.y), (k * a.z))
программа скалярное произведение ⟨вх: a, b: вектор R3⟩: число ==
                                  (a.x * b.x + a.y * b.y + a.z * b.z)
программа векторное произведение ⟨вх: a, b⟩: вектор R3 ==
((a.y * b.z — a.z * b.y), (a.z * b.x — a.x * b.z), (a.x*b.y — a.y*b.x))
конец исполнителя |==========================================
```

Оптимизация алгоритма построения изображения полиэдра. Итак, мы завершили реализацию проекта «Построение изображения полиэдра» и теперь можем обсудить, что получилось. Получилась разработанная по технологии «сверху вниз» хорошо структурированная программа, которая, однако, достаточно наивна, делает много лишней работы и на сложных полиэдрах (несколько тысяч ребер и граней) будет работать очень долго. Хорошая структурированность программы позволяет ее легко менять и оптимизировать. Мы сейчас опишем две такие модификации: фильтрование и предкомпиляцию граней. Еще одна модификация будет описана в разд. 15.

Фильтрование граней. Посмотрим, где наша программа делает лишнюю работу, которой можно было бы избежать. Наиболее содержательным является выполнение программы «Изобразитель ребра.учесть тень от грани» — при этом происходит обращение к исполнителю «Геометр», находятся нормаль и центр грани, горизонтальное и вертикальные пересечения, пересекаются отрезки в t-координатах на ребре и т. д. Все эти действия выполняются для каждой

пары (ребро, грань). Но взгляните на рис. 11.11 — для каждого конкретного ребра лишь очень немногие грани (а именно грани, расположенные на плоскости проекции рядом с ребром) действительно

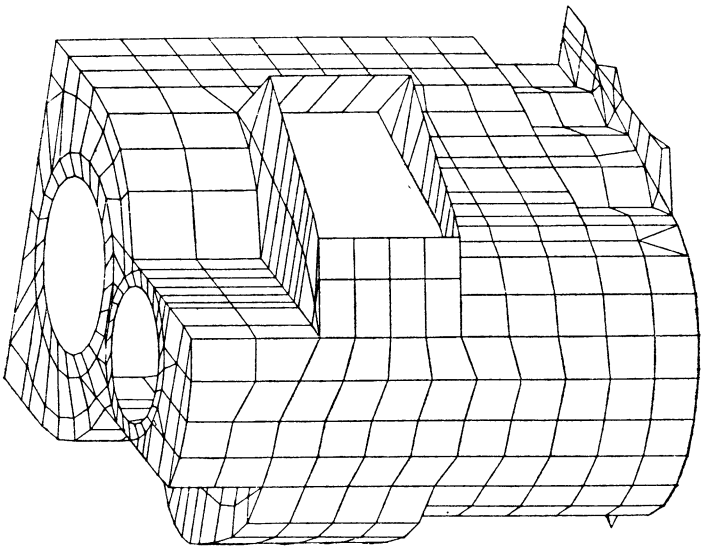


Рис. 11.11

оказывают какое-то влияние на его видимость. Подавляющая же часть граней расположена на плоскости проекции далеко от ребра и заведомо его не затеняет. Естествен вопрос: нельзя ли для далеких от ребра граней никаких горизонтальных и вертикальных пересечений не вычислять, а прямо говорить, что грань тени на ребро не дает?

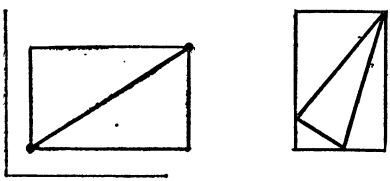


Рис. 11.12

Одно из решений выглядит так: введем на плоскости проекции координаты u, v. Назовем *прямоугольником грани* наименьший прямоугольник вида umin $\leqslant$ u $\leqslant$ umax, vmin $\leqslant$ v $\leqslant$ vmax, содержащий проекцию грани. Аналогично определим *прямоугольник ребра* (рис. 11.12).

Очевидно, что если прямоугольники ребра и грани не пересекаются, то грань заведомо не затеняет ребро, и, значит, эту грань можно не учитывать при построении изображения видимой части ребра:

```
программа построение изображения полиэдра
. дано      : | полиэдр в R3 (исполнитель "Полиэдр")
.             | направление проектирования (исполнитель "Проектор")
. получить: | плоское изображение ребер полиэдра (параллельная
.             | проекция) с "удалением невидимых линий"
. --------
.  цикл ∀ ребро ∈ Полиэдр.множество ребер выполнять
.  .  Фильтр.начать работу ⟨вх: ребро⟩
.  .  Изобразитель ребра.начать работу ⟨вх: ребро⟩
.  .  цикл ∀ грань ∈ Полиэдр.множество граней выполнять
.  .  .  если Фильтр.грань ⟨вх: грань⟩ имеет отношение к ребру
.  .  .  .  то
.  .  .  .  Изобразитель ребра.учесть тень от грани ⟨вх: грань⟩
.  .  .  конец если
.  .  конец цикла
.  .  Изобразитель ребра.кончить работу
.  .  Фильтр.кончить работу
.  конец цикла
конец программы
```

Здесь предписание «грань имеет отношение к ребру» исполнителя «Фильтр» вырабатывает значение да, если прямоугольник грани пересекает прямоугольник ребра, и нет в противном случае.

```
исполнитель Фильтр
. СП:
.  1. начать работу ⟨вх: ребро⟩
.  2. грань ⟨вх: грань⟩ имеет отношение к ребру    : да/нет
.  3. кончить работу
.
.  типы:
.  прямоугольник = запись (umin, vmin, umax, vmax: R)
.
.  объекты:
.  P: прямоугольник | прямоугольник ребра на плоскости UV
.
.  используемые исполнители:
.  UV  | (нахождение прямоугольников ребер и граней)
.
конец описаний |----------
```

```
программа начать работу (вх: ребро)
. дано      : ребро : ребро
. получить:
. --------
.  P := UV.прямоугольник ребра (зх: ребро)
конец программы

программа грань (вх: грань) имеет отношение к ребру: да/нет
. дано      :  грань : грань
. получить: | ответ = (прямоугольники ребра и грани пересекаются)
. --------
.  Г := UV.прямоугольник грани (вх: грань)
.  ответ :=
.     максимум (P.umin,Г.umin) < минимум(P.umax,Г.umax) и
.     максимум (P.vmin,Г.vmin) < минимум (P.vmax,Г.vmax)
конец программы

программа кончить работу = ничего не делать
конец исполнителя | ==========================
```

Реализация исполнителя UV оставляется читателю. Заметим лишь, что координаты u и v на плоскости проекции могут быть и косоугольными. Например, если вектор проектирования не параллелен плоскости $z = 0$ (т. е. $p.z \neq 0$), то в качестве u и v можно использовать просто x и y компоненты трехмерных координат точек плоскости проекции.

В модифицированной программе для далеких от ребра граней проводятся только тривиальные вычисления прямоугольника грани и сравнение этого прямоугольника с прямоугольником ребра. Поскольку для каждого конкретного ребра подавляющая масса граней являются далекими, то скорость построения изображения полиэдра возрастет примерно во столько раз, во сколько фильтрование быстрее учета тени от грани на ребро. Таким образом, фильтр тем лучше, чем меньше граней он пропускает и чем быстрее он работает. Эти требования являются противоречивыми. Идеальным по критерию минимума граней является фильтр, который пропускает только действительно затеняющие ребро грани. Однако определение таких граней почти не отличается от нахождения тени от грани на ребро и никакого выигрыша в скорости не дает. Идеальным по критерию скорости работы является фильтр, который пропускает все грани вообще, но применение такого фильтра бессмысленно. Поэтому требуется разумный компромисс — фильтр, который быстро отсеивает основную массу не имеющих отношения к ребру граней, пропуская, быть может, какое-то количество «подозрительных» гра-

ней, на самом деле ребро не затеняющих. Одним из таких разумных компромиссов и является реализованный выше фильтр на основе прямоугольников.

Предкомпиляция граней. Можно ожидать, что после введения фильтра основное время работы будет уходить на выполнение программы «грань имеет отношение к ребру». Эта программа вызывается для каждой пары (ребро, грань) и состоит в вычислении прямоугольника грани и анализе его пересечения с прямоугольником ребра. Таким образом, прямоугольник каждой грани вычисляется многократно (заново для каждого ребра). Очевидная оптимизация состоит в том, чтобы заранее до начала построения изображения один раз вычислить прямоугольники всех граней, запомнить их и использовать при фильтровании.

Описанный прием носит общий характер и называется *заменой интерпретации компиляцией*. Если задан объект и необходимо многократно использовать сведения, которые извлекаются из описания объекта ценой каких-то вычислений, то вместо многократного повторения этих вычислений (*интерпретации описания объекта*) можно провести их заранее (*компилировать описание объекта*) и запомнить результат. Заметим, что для запоминания результата компиляции требуется дополнительная память. Вопрос о соотношении интерпретации и компиляции в каждом конкретном случае решается по-своему, исходя из специфики задачи. Если, например, основное время построения изображения полиэдра и после введения фильтра тратится на нахождение теней от граней, то при предкомпиляции разумно вычислять не только прямоугольник грани, но и коэффициенты линейных неоднородных функций, задающих тень от грани в R3.

Обратите внимание, что, несмотря на все оптимизации, мы по-прежнему для каждого ребра *анализируем все грани*, только теперь для большинства граней этот анализ производится достаточно быстро. Построение изображения полиэдра можно еще ускорить, если для каждого ребра анализировать только близкие, «подозрительные» грани, а далекие *вообще никак не рассматривать*. Идеи такой реализации (идеи двумерного хеширования по равномерной сетке), а также сама реализация будут изложены в разд. 15.

ЗАДАЧИ И УПРАЖНЕНИЯ

В задачах 1—7, при необходимости изменив проект «Построение изображения полиэдра», напишите программу, которая находит:

1. Сумму длин всех ребер полиэдра.
2. Число вершин грани ⟨**вх** : грань⟩.
3. Число геометрически различных вершин полиэдра.
4. Видима ли точка ⟨**вх** : точка R3⟩.
5. Число граней полиэдра, пересекающихся с ⟨**вх** : отрезок R3⟩.
6. Число всех полностью видимых ребер полиэдра.
7. Число всех полностью невидимых ребер полиэдра.

8. Реализуйте исполнителя UV.

9. Реализуйте предкомпиляцию прямоугольников граней, считая, что предписание «Полиэдр.множество граней» принимает и вырабатывает значение типа **«множество элементов типа грань с нагрузкой типа** прямоугольник».

10. Считая, что исполнитель «Проектор» вместо вектора проектирования работает с точкой f из R3 и плоскостью П, отделяющей f от полиэдра, напишите программу, которая строит изображение полиэдра на плоскости П при перспективной проекции с центром в точке f.

ГЛАВА 3
СТРУКТУРЫ ДАННЫХ И ИХ РЕАЛИЗАЦИИ

Эта глава посвящена реализациям одних структур данных на базе других. Задачи первых трех разделов формулируются так: «Пусть в языке программирования отсутствует способ конструирования объектов X, но имеется способ конструирования Y. Реализовать исполнителя X на базе Y». Например, реализовать исполнителя «Стек элементов типа E» на базе исполнителя «Вектор элементов типа E».

С практической точки зрения решения таких задач интересуют нас по двум причинам.

Во-первых, разбор реализаций одних структур данных на базе других позволяет глубже усвоить, «прочувствовать» сами эти структуры, а также научиться при необходимости реализовать другие структуры, не являющиеся предопределенными. Для математика, впрочем, вопрос о сводимости одних структур данных к другим может представлять и самостоятельный интерес.

Во-вторых, такие задачи сплошь и рядом возникают при использовании широко распространенных производственных языков программирования (Фортран, Си, разнообразные ассемблеры, Бейсик, Паскаль, Ада и др.). Из перечисленных в разд. 7 структур эти языки содержат только способы конструирования **вектор** и **матрица** (в несколько более общем виде называемом *массивом*). Дело в том, что память современных ЭВМ физически устроена как вектор элементов некоторого типа с индексом от нуля до некоторого максимального числа (более подробно об устройстве ЭВМ см. следующую главу). Именно поэтому способ конструирования **вектор** присутствует во всех языках программирования, и именно поэтому почти все реализации этой главы будут реализациями на базе вектора. Не следует, однако, думать, что на практике любая структура реализуется на базе вектора за один шаг декомпозиции. Часто приходится сделать несколько шагов, используя промежуточные структуры данных, прежде чем искомая структура окажется реализованной на базе вектора. Практический пример такой многошаговой реализации приведен в разд. 18.

При решении задач иногда возникает необходимость в использовании и нестандартных структур, формы работы с элементами которых специфичны для каждой конкретной задачи. Примерам таких

структур и методам их реализации посвящены разд. 16—18. Начиная с этих разделов, на наших страницах начнет появляться аббревиатура ЭВМ — электронно-вычислительная машина. Включение информации об ЭВМ в курс программирования вызвано тем, что на практике именно ЭВМ лежат в основе тех реальных Универсальных Выполнителей, которые выполняют наши программы. Тем самым программисту полезно знать общее устройство и принцип работы, алгоритмы функционирования отдельных компонент и другие детали конструкции ЭВМ. Этим вопросам будет посвящена следующая глава. Здесь же, занимаясь вопросами реализации различных структур данных, мы будем объяснять некоторые детали устройства ЭВМ, чтобы показать, откуда возникает та или иная структура, та или иная задача.

С теоретической точки зрения глава описывает некоторые новые структуры данных (файловая система, виртуальная память), а ее основное содержание составляют общие методы и конкретные приемы реализации различных структур данных на базе вектора: непрерывные реализации, ссылочные реализации, бинарный поиск, хеширование и т. д.

12. Примеры реализации одних структур данных на базе других. Непрерывные реализации на базе вектора

Последовательность на базе двух очередей. Рассмотрим задачу реализации исполнителя «Последовательность элементов типа Е» на базе двух исполнителей типа «Очередь элементов типа Е». Другими словами, надо написать программы, соответствующие предписаниям исполнителя «Последовательность», которые будут «создавать видимость» работы с последовательностью, в то время как на самом деле будут манипулировать двумя очередями. Этот факт надо постоянно иметь в виду — во всех реализациях и здесь и далее всегда будут *имитируемая* структура и структуры, с которыми мы на самом деле работаем. На идейном уровне решение таких задач состоит в том, что надо придумать размещение элементов имитируемой (или воображаемой) структуры в реальной и понять, к каким реальным действиям будут приводить операции с воображаемой структурой. Первую часть задачи — размещение элементов — удобно придумывать и изображать на картинках (рис. 12.1).

На рис. 12.1 элементы воображаемой структуры «Последовательность» изображены кружками и расположены в двух очередях (левая и правая «трубы» соответственно). При этом элементы воображаемой прочитанной части последовательности расположены в левой очереди, а элементы воображаемой непрочитанной части — в правой очереди.

Когда такая картинка придумана, надо просмотреть систему предписаний имитируемого исполнителя (исполнителя «Последовательность») и убедиться в том, что выбранный порядок размещения элементов позволяет проимитировать все операции с этим исполнителем. В данном случае это действительно так. Например, имитация предписания «прочесть очередной элемент последовательности в ⟨вых : E⟩» состоит в том, что надо взять элемент из начала правой очереди, добавить его в конец левой очереди (т. е. в прочитанную часть последовательности), а также скопировать состояние этого элемента в выходной параметр E.

Отметим, что число реальных действий с очередями, выполняемых при имитации этого предписания, невелико и не зависит от

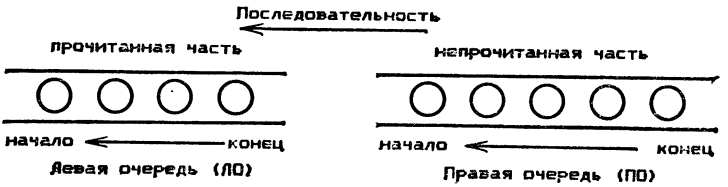


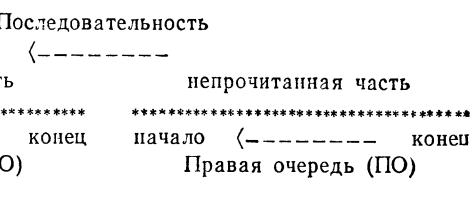
Рис. 12.1

числа элементов в последовательности или в очередях. В реализации последовательности это свойство (назовем его *отсутствием массовых операций*) будет выполнено для всех предписаний, кроме предписания «встать в начало последовательности», когда число действий зависит от количества элементов имитируемой последовательности. Допустима или нет такая медленная имитация этого предписания, зависит от ситуации, в которой мы собираемся использовать последовательность.

Идеи имитации остальных предписаний исполнителя «Последовательность» ясны из текста реализации.

исполнитель Последовательность | (на базе двух очередей)
. **СП:**
. 1. начать работу .
. 2. сделать последовательность пустой
. 3. последовательность пуста : да/нет
. 4. добавить элемент ⟨вх : E⟩ в конец последовательности
. 5. встать в начало последовательности
. 6. есть непрочитанные элементы : да/нет
. 7. нет непрочитанных элементов : да/нет
. 8. прочесть очередной элемент в ⟨вых : E⟩
. 9. очередной элемент последовательности ; :E
. 10. пропустить очередной элемент последовательности

. 11. кончить работу
.
. **идеи реализации:**
. Последовательность будем хранить в левой (ЛО) и правой (ПО) очередях. Левая очередь будет соответствовать прочитанной части последовательности, а правая — непрочитанной:
.



.
. **используемые исполнители:**
. ЛО, ПО: **очередь элементов типа** E
.
конец описаний |----------

программа начать работу == ЛО.начать работу; ПО.начать работу
программа сделать последовательность пустой ==
ЛО.сделать очередь пустой; ПО.сделать очередь пустой
программа последовательность пуста: **да/нет** ==
(ЛО.очередь пуста **и** ПО.очередь пуста)
программа добавить элемент ⟨**вх**: E⟩ в конец последовательности ==
ПО.добавить элемент ⟨**вх**: E⟩ **в** конец очереди

программа встать **в** начало последовательности
. **дано** : | последовательность как-то расположена в 2 очередях
. **получить:** | последовательность целиком в правой очереди, т. е.
. ЛО.очередь пуста
.---------
. **цикл** ∀ E ∈ **очереди** ПО **выполнять**
. . ЛО.добавить элемент ⟨**вх**: E⟩ в конец очереди
. **конец цикла**
. **утв:** ПО.очередь пуста
. **цикл** ∀ E ∈ **очереди** ЛО **выполнять**
. . ПО.добавить элемент ⟨**вх**: E⟩ в конец очереди
. **конец цикла**
. **утв:** ЛО.очередь пуста
конец программы

программа есть непрочитанные элементы: **да/нет** ==
ПО.очередь не пуста
программа нет непрочитанных элементов: **да/нет** ==
ПО.очередь пуста

```
программа прочесть очередной элемент в ⟨вых: E⟩
. дано      : есть непрочитанные элементы
. получить:
.--------
.  ПО.взять элемент из начала очереди в ⟨вых: E⟩
.  ЛО.добавить элемент ⟨вх: E⟩ в конец очереди
конец программы

программа очередной элемент последовательности :: E
. дано      : есть непрочитанные элементы
. получить:
.--------
.  ответ == ПО.начало очереди
конец программы

программа пропустить очередной элемент последовательности
. дано      : есть непрочитанные элементы
. получить:
.--------
.  ПО.взять элемент из начала очереди в ⟨вых: E⟩
.  ЛО.добавить элемент ⟨вх: E⟩ в конец очереди
конец программы

программа кончить работу ==
                          ЛО.кончить работу; ПО.кончить работу
конец исполнителя |==========================
```

Здесь мы впервые встречаемся с программной реализацией предписания, принимающего и вырабатывающего значение. Такое предписание является *обозначением* для некоторого глобального объекта, в данном случае — объекта «ПО.начало очереди». Обратите внимание на форму записи ответа в таких программах — вместо **ответ:=** пишется **ответ==**, причем справа от знака == (читается «это есть») может стоять только некоторый глобальный объект или его обозначение. Таким образом, запись

Последовательность.очередной элемент := 7

означает то же самое, что и

ПО.начало очереди := 7,

и означает, что элемент, находящийся в начале правой очереди, должен быть установлен в состояние 7. Аналогично запись

x := x + Последовательность.очередной элемент

означает то же самое, что и

x := x + ПО.начало очереди.

При использовании реализованного выше исполнителя «Последовательность» можно отвлечься от реализации и считать (воображать), что этот исполнитель действительно содержит последовательность элементов типа Е. Повторим, однако, что при реализации этого исполнителя мы на самом деле манипулируем двумя очередями, лишь *имитируя соответствующие действия с последовательностью.*

Непрерывные реализации на базе вектора. Теперь рассмотрим задачу реализации различных структур (стек, дек, очередь и т. п.) элементов типа Е на базе вектора элементов типа Е. Поскольку число элементов вектора заранее фиксировано и конечно, то на базе вектора можно реализовать лишь *ограниченные* стек, дек, очередь и пр. Ограниченная структура отличается от потенциально неограниченной тем, что может вместить не более некоторого, заранее фиксированного числа элементов. При попытке добавить элемент сверх этого числа (при «переполнении») возникает ситуация «отказ». Система предписаний ограниченной структуры кроме обычных содержит также предписания

есть свободное место/нет свободного места : да/нет

Эти предписания проверяют, является ли уже количество элементов структуры максимальным (нет свободного места) или еще нет (свободное место есть).

Будем называть *простыми* объекты предопределенных типов (**да/нет, символ, Z+, Z, R**), а также объекты, построенные с помощью способов конструирования **перечисление** и **отрезок**. Во всех реализациях на базе вектора предполагается, что кроме самого вектора можно использовать произвольное число объектов простых типов, а также записей из них. Единственное требование состоит в том, чтобы количество таких объектов не зависело от количества элементов в векторе.

В этом разделе мы изучим только один метод реализации структур на базе вектора, который называется *непрерывной реализацией*. Суть этого метода состоит в том, что *элементы структуры размещаются в векторе* «непрерывным куском» — *рядом друг с другом, а порядок элементов в структуре*, если он есть, *определяется порядком их следования в векторе.*

Ограниченный стек на базе вектора. При реализации ограниченного стека на базе вектора расположим элементы имитируемого стека в начале вектора (рис. 12.2). В изображенной на рис. 12.2 ситуации воображаемый стек содержит шесть элементов. Вершина стека находится в компоненте вектора с индексом 6, следующий за ней элемент — в компоненте вектора с индексом 5 и т. д. Дно стека находится в компоненте вектора с индексом 1. Тем самым шесть

элементов воображаемого стека действительно хранятся в векторе «непрерывным куском», занимая элементы вектора с индексами от 1 до 6. Состояния элементов вектора с индексами, большими 6, в этот момент не важны — они никак не влияют на имитацию стека.

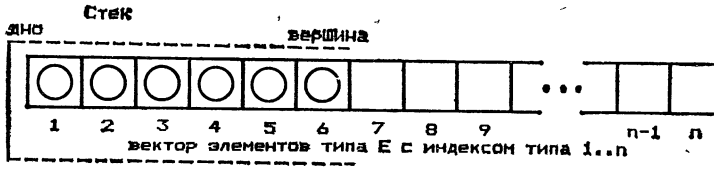


Рис. 12.2

Естественно, что для имитации работы со стеком недостаточно хранить состояния элементов вектора — нужно знать еще и число элементов в стеке. Для хранения этого числа достаточно объекта

мощность : 0..n | число элементов в стеке

исполнитель Ограниченный стек | на базе вектора
. **СП:**
. 1. начать работу
. 2. сделать стек пустым
. 3. стек пуст : да/нет
. 4. есть свободное место : да/нет
. 5. нет свободного места : да/нет
. 6. добавить элемент ⟨**вх** : E⟩ в стек
. 7. взять элемент из стека в ⟨**вых** : E⟩
. 8. вершина стека :: E
. 9. удалить вершину стека
. 10. кончить работу
.
. **константы:**
. максинд = 100 | макс.индекс, т. е. число элементов вектора
.
. **объекты:**
. мощность : 0..максинд | т. е. число элементов в стеке
.
. **идеи реализации:**
. Расположим элементы стека в начале вектора, "повернув" стек дном к началу вектора. В стеке из трех элементов, таким образом, вершина стека будет в 3 компоненте вектора, следующий за вершиной элемент — во 2, а дно стека — в 1.
.
. **используемые исполнители:**

```
. вектор:вектор элементов типа Е с индексом 1..максинд
.
конец описаний |----------------------------------

программа начать работу == вектор.начать работу; мощность := 0
программа сделать стек пустым          == мощность: = 0
программа стек пуст          : да/нет == мощность  = 0
программа есть свободное место: да/нет == мощность  < максинд
программа нет свободного места: да/нет == мощность  = максинд

программа добавить элемент (вх: Е) в стек
. дано     : есть свободное место
. получить:
.--------
. мощность.увеличить на 1
. вектор (мощность) := Е
конец программы

программа взять элемент из стека в (вых: Е)
. дано     : стек не пуст
. получить:
.--------
. Е := вершина стека
. удалить вершину стека
конец программы

программа вершина стека :: Е
. дано     : стек не пуст
. получить:
.--------
. ответ == вектор (мощность)
конец программы

программа удалить вершину стека
. дано     : стек не пуст
. получить:
.--------
  мощность.уменьшить на 1
конец программы

программа кончить работу == вектор.кончить работу
конец исполнителя |==============================
```

Ограниченная очередь на базе «циклического» вектора. Следующей задачей является реализация ограниченной очереди на базе «циклического» вектора. Это название, впрочем, не вполне верно — вектор здесь используется самый обычный, а слово «циклический»

относится к идеям реализации. Дело в том, что если как-то расположить элементы очереди в непрерывном куске вектора, а потом начать добавлять элементы в конец и брать из начала, то в некоторый момент времени очередь «упрется» в правый край вектора

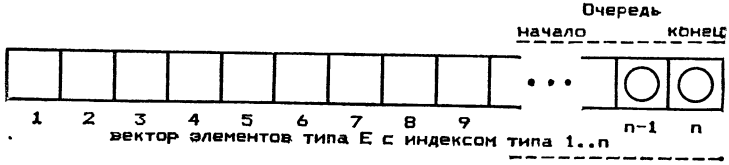


Рис. 12.3

(рис. 12.3). В этот момент для добавления очередного элемента в конец очереди, вообще говоря, надо сдвинуть все элементы очереди влево по вектору, чтобы справа освободилось место. Количество элементарных операций с вектором, которые придется проделать при сдвиге, пропорционально числу элементов в очереди. Таких *массовых* (т. е. зависящих от числа элементов имитируемой структуры) действий можно избежать, если «свернуть вектор в кольцо» (рис. 12.4), т. е. считать, что за последним элементом вектора идет первый. Тогда, «упершись» в правый край вектора, надо по кольцу

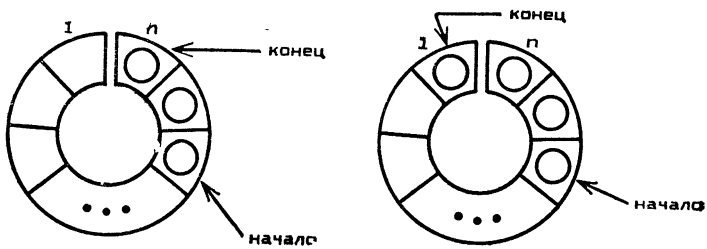


Рис. 12.4

перейти к первому элементу вектора, затем ко второму и т. д. Поскольку при этом элементы очереди на кольце размещаются непрерывным куском, то такая реализация также называется непрерывной. Таким образом, под словом «непрерывность» надо теперь понимать непрерывность на кольце. В программировании непрерывную реализацию очереди на базе «циклического» (т. е. свернутого в кольцо) вектора часто называют *кольцевым буфером.*

Посмотрим, какие еще объекты, кроме вектора, необходимы для имитации очереди. Прежде всего надо знать, где именно в векторе (или, что то же самое, на кольце) расположены элементы воображаемой очереди, т. е. знать, где находится начало очереди и куда

помещать элементы, добавляемые в конец. Будем хранить соответствующую информацию в объектах

начало, конец : индекс | индексы начала и конца очереди

Возникает, правда, вопрос: каковы будут состояния этих объектов в пустой очереди? В одноэлементной очереди «начало» и «конец» совпадают и показывают на единственный элемент очереди (рис. 12.5, *а*). При взятии элемента из начала очереди объект «начало» обычно продвигается вперед (против часовой стрелки) по кольцу. Естественно поэтому считать, что в пустой очереди «начало»

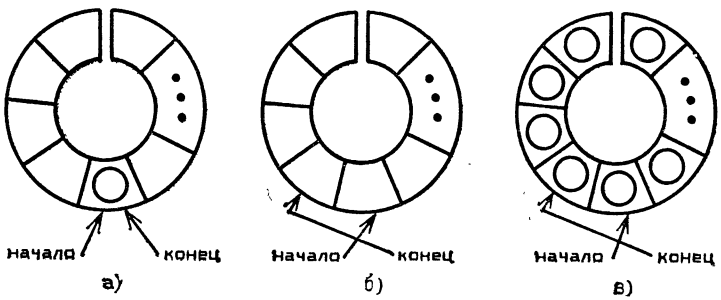


Рис. 12.5

показывает на элемент, следующий за элементом, на который указывает «конец» (рис. 12.5, *б*). Однако в точно таком же положении объекты «начало» и «конец» должны быть, если очередь занимает весь вектор целиком (рис. 12.5, *в*).

Таким образом, чтобы отличить пустую очередь от полной, объектов «начало» и «конец» мало. Не мудрствуя лукаво, введем еще один объект

мощность : 0..максинд | т е. число элементов в очереди

который будем использовать для определения пустоты или непустоты очереди, а также наличия или отсутствия свободного места в точности так, как использовали объект «мощность» при реализации стека.

исполнитель Ограниченная очередь | на базе "циклического" вектора
. **СП:**
. 1. начать работу
. 2. сделать очередь пустой
. 3. очередь пуста : да/нет
. 4. есть свободное место : да/нет
. 5. нет свободного места : да/нет
. 6. добавить элемент (**вх** : Е) в конец очереди

```
. 7. взять элемент из начала очереди в (вых: E)
. 8. начало очереди                                  :: E
. 9. удалить начало очереди
. — следующий индекс (вх: индекс)                    : индекс
. 10. кончить работу
.
. константы:
. максинд = 100 | макс.индекс, т. е. число элементов вектора
. типы:
. индекс = 1..максинд
. объекты:
. мощность: 0..максинд | т. е. число элементов в очереди
. начало, конец: индекс | индексы начала и конца очереди
.
. идеи реализации:
. "Склеим" концы вектора и будем реализовывать очередь на по-
. лучившемся кольце. Элементы очереди будем хранить в компо-
. нентах вектора с индексами начало..конец
.
. используемые исполнители:
. вектор: вектор элементов типа E с индексом типа индекс
конец описаний | ------------------------------

программа начать работу ==
                    вектор.начать работу; сделать очередь пустой
программа сделать очередь пустой
. дано     :
. получить: очередь пуста
.---------
. конец     := максинд
. начало    := следующий индекс (конец)
. мощность := 0
конец программы

программа очередь пуста          : да/нет == (мощность = 0)
программа есть свободное место : да/нет == (мощность < максинд)
программа нет свободного места : да/нет == (мощность = максинд)

программа добавить элемент (вх: E) в конец очереди
. дано     : есть свободное место
. получить:
.---------
. конец := следующий индекс (конец)
. вектор (конец) := E
. мощность.увеличить на 1
конец программы
```

```
программа взять элемент из начала очереди в ⟨вых: E⟩
. дано      : очередь не пуста
. получить:
.--------
.  E := начало очереди
.  удалить начало очереди
конец программы

программа начало очереди:: E
. дано      : очередь не пуста
. получить:
.--------
.  ответ == вектор (начало)
конец программы

программа удалить начало очереди
. дано      : очередь не пуста
. получить:
.--------
.  начало := следующий индекс ⟨начало⟩
.  мощность.уменьшить на 1
конец программы

программа следующий индекс ⟨вх: И: индекс⟩: индекс
. дано      : И: индекс |индекс некоторого элемента вектора
. получить: |индекс следующего элемента вектора. Следующим за
.           |последним считается первый элемент
. --------
. выбор
. . при И < максинд ⇒ ответ := И + 1
. . при И = максинд ⇒ ответ := 1
. конец выбора
конец программы

программа кончить работу == вектор.кончить работу
конец исполнителя |==========================
```

Заметьте, что программу «следующий индекс» можно записать короче, если использовать операцию **mod**, которая получает остаток от деления первого аргумента на второй:

```
программа следующий индекс ⟨вх: И: индекс⟩: индекс ==
                                        (И mod максинд) + 1
```

ЗАДАЧИ И УПРАЖНЕНИЯ

В задачах 1—13 требуется реализовать одну структуру, состоящую из элементов типа E, на базе другой структуры, также состоящей из элементов типа E. Если в задаче фигурирует вектор

(задачи 8 и 9), то это вектор элементов типа E с индексом типа 1 .. максинд, где максинд $\geqslant 1$ — некоторое целое число. Кроме базовой структуры в реализации разрешается использовать любое количество объектов простых типов и записей из них.

1. Последовательность на базе динамического вектора.
2. Динамический вектор на базе последовательности.
3. Стек на базе Л1-списка.
4. Дек на базе Л2-списка.
5. Очередь на базе Л1-списка.
6. Очередь на базе Л2-списка.
7. Множество на базе Л1-списка.
8. Вектор на базе динамического вектора.
9. Вектор на базе последовательности.
10. Динамический вектор на базе нагруженного множества.
11. Л1-список на базе дека.
12. Два стека на базе Л2-списка.
13. Л2-список на базе двух стеков.

Указание. Два стека — это исполнитель со следующей системой предписаний:

1. начать работу
2. сделать стек ⟨**вх** : N⟩ пустым
3. стек ⟨**вх** : N⟩ пуст/стек ⟨**вх** : N⟩ не пуст : **да/нет**
4. добавить элемент ⟨**вх** : E⟩ в стек ⟨**вх** : N⟩
5. взять элемент из стека ⟨**вх** : N⟩ в ⟨**вых** : E⟩
6. вершина стека ⟨**вх** : N⟩ : :E
7. удалить вершину стека ⟨**вх** : N⟩
8. кончить работу

где входной параметр N, принимающий одно из двух значений 1 или 2, имеет смысл номера стека, к которому относится предписание.

В задачах 14—18 надо провести непрерывную реализацию указанных исполнителей на базе одного вектора элементов типа E с индексом типа 1 .. максинд, где максинд $\geqslant 1$ — некоторое целое число. Кроме вектора разрешается использовать любое количество объектов простых типов и записей из них.

14. Ограниченное множество элементов типа E.
15. Ограниченный дек элементов типа E.
16. Ограниченную последовательность элементов типа E.
17. Ограниченный динамический вектор элементов типа E.
18. Два стека элементов типа E, ограниченные в совокупности. (Система предписаний исполнителя «Два стека» приведена в указании к задачам 12 и 13. Слова «ограниченные в совокупности» означают, что ограничено лишь суммарное количество элементов

в обоих стеках. Таким образом, предписание «есть свободное место» является общим для обоих стеков и не имеет параметра, указывающего номер стека.)

19. Реализуйте матрицу элементов типа E с индексами типа (1 .. M, 1. .. N) на базе вектора элементов типа E с индексом типа 1 .. максинд, где максинд = M * N.

13. Ссылочные реализации на базе вектора

В непрерывных реализациях предыдущего раздела мы старались избегать массовых операций, в частности старались не сдвигать элементы структуры в векторе. Поскольку, однако, при непрерывной реализации вектор одновременно выполняет две функции — служит хранилищем элементов имитируемой структуры и определяет порядок элементов этой структуры, то сдвигов можно

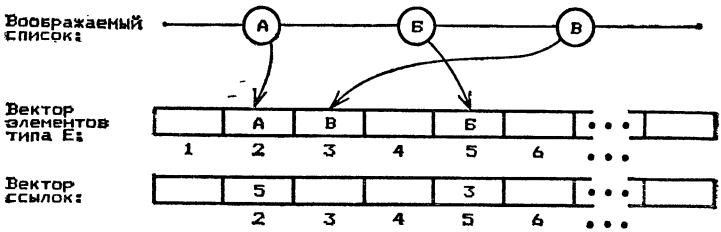


Рис. 13.1

избежать не всегда. Для структур, где порядок элементов не важен (множество) или не меняется (стек, очередь, дек, последовательность, динамический вектор), непрерывная реализация без массовых операций возможна. Но при непрерывной реализации, например, Л1-списка сдвиги неизбежны, так как в Л1-список можно вставить элемент в середину.

Массовых операций можно избежать и при реализации списков, но для этого надо *отдельно хранить элементы* имитируемой структуры и *отдельно задавать их порядок.* Для Л1-списка, например, элементы списка можно произвольным образом расположить в векторе элементов типа E с индексом типа 1 .. максинд, а информацию о порядке элементов задавать во вспомогательном векторе (назовем его вектором *ссылок*) элементов типа 1 .. максинд с индексом также типа 1 .. максинд. Например, если в компоненте вектора элементов типа E с индексом 5 расположен какой-то элемент имитируемого Л1-списка, а следующий за ним элемент списка расположен в компоненте вектора с индексом 3, то эта информация запоминается в векторе ссылок — ссылка с индексом 5 устанавливается в состояние 3 (рис. 13.1).

Таким образом, основная идея ссылочной реализации состоит в отделении информации о содержимом элементов имитируемой структуры (вектор элементов типа Е) от информации об их порядке (вектор ссылок). Ссылки далее будем изображать стрелками, как показано на рис. 13.2.

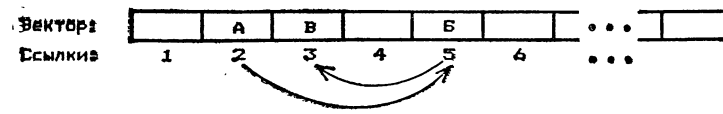


Рис. 13.2

Естественно, что для работы со списком одних только ссылок между элементами недостаточно — надо знать, например, где расположен первый элемент списка, и отличать последний элемент от всех остальных (ведь за ним уже никаких элементов нет). Для этих целей можно было бы ввести два глобальных объекта типа

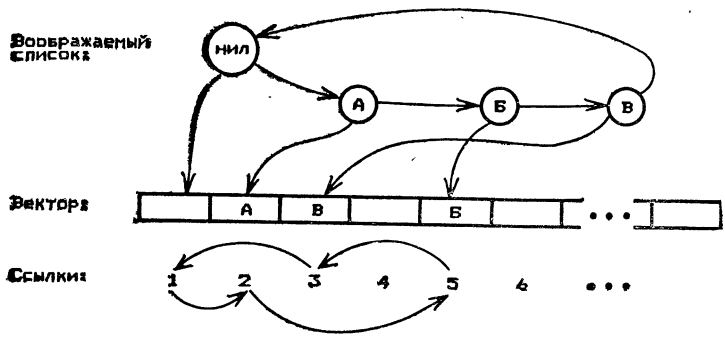


Рис. 13.3

индекс, которые содержали бы индексы начала и конца списка. Мы, однако, этого делать не будем, а снова воспользуемся приемом «сворачивания в кольцо» — «свернем» список, введя специальный воображаемый элемент *нил*, который всегда будем располагать в компоненте вектора с индексом 1 (рис. 13.3).

Заметьте, что теперь ссылки между элементами списка (включая нил) образуют кольцо, начинающееся с 1. Таким образом, ссылка с индексом 1 показывает на первый элемент списка, а если очередная ссылка равна 1, то соответствующий элемент является в списке последним. Пустому списку соответствует ситуация, когда в кольце никаких элементов, кроме нил, нет, т. е. ссылка с индексом 1 равна 1 (показывает «на себя»).

Указатель в списке располагается между элементами, поэтому реализуем его в виде объекта

указатель : запись (@до, @за : индекс),

где поле «@до» содержит индекс элемента до указателя, а поле «@за» — индекс элемента за указателем *). Положению указателя в начале списка, таким образом, соответствует случай, когда поле «@до» равно 1 (указывает на нил), а положению в конце — когда 1 равно поле «@за».

Теперь список проимитирован полностью. Неясным остался только вопрос: как определять наличие или отсутствие свободного места и как находить свободный элемент вектора при имитации добавления элемента к списку? Чтобы не гадать, какие элементы вектора свободны, а какие заняты элементами списка, примем решение, что свободные элементы вектора также будут соединены в кольцо, начинающееся со служебного элемента «нил свободного места» — компоненты вектора с индексом 2 (рис. 13.4).

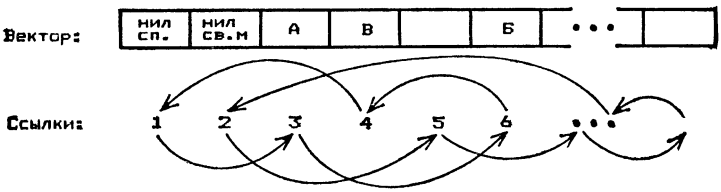


Рис. 13.4

Заметим, что, поскольку элементы с индексами 1 и 2 теперь играют служебную роль, в них никогда не попадет никакой элемент имитируемой структуры. Следовательно, вектор элементов типа Е, предназначенный для хранения элементов списка, может иметь индекс в диапазоне не 1 .. максинд, а 3 .. максинд.

С математической точки зрения каждое состояние вектора ссылок определяет некоторое отображение f множества индексов в себя — индексу i ставится в соответствие значение элемента вектора ссылок с индексом i. Принятые нами решения означают, что в каждый момент времени отображение f взаимно однозначно и имеет ровно две периодические траектории «список» и «свободное место». Эти траектории имеют вид

нил, f(нил), f(f(нил)), f(f(f(нил))), . . . , нил, f(нил), . . . ,

*) Символ @ (официальное название — «at» коммерческое, обычно произносится «блямба») считается такой же буквой, как a, b, c и др. Мы будем начинать с этого символа имена объектов-ссылок, т. е. если имя объекта начинается с @, то состоянием объекта является индекс какого-то элемента вектора.

не пересекаются, покрывают все множество индексов и начинаются с 1 (нил списка) и 2 (нил свободного места).

(*Траекторией* отображения $f: X \to X$, начинающейся с точки a, мы называем множество тех точек $x \in X$, которые можно получить из a, многократно применяя f, т. е. $\{x \in X: \exists n \in Z+ : x = f^n(a)\}$, где $f^0(a)=a$, $f^1(a)=f(a)$, $f^2(a)=f(f(a))$, ..., $f^n(a)=f(f(\ldots f(a)\ldots))$ (n раз). Траектория называется *периодической*, если существует $n>0$ такое, что $f^n(a)=a$.)

Таким образом, чтобы определить, есть ли в векторе свободное место, надо посмотреть значение ссылки с индексом 2. Если это значение равно 2 (т. е. траектория «свободное место» не содержит элементов, отличных от нил свободного места), то свободного места нет. В противном случае это значение указывает на свободный

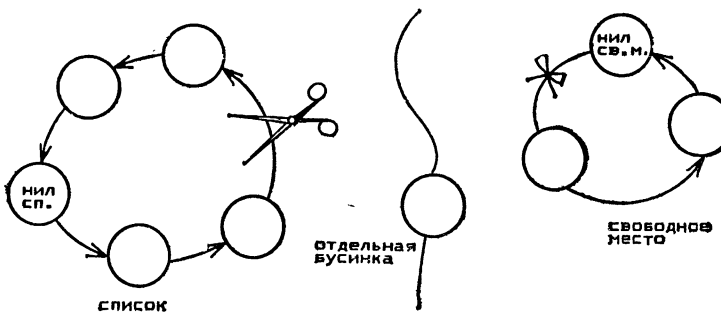


Рис. 13.5

элемент вектора, в который можно поместить новый элемент списка при имитации его добавления. Заметьте, что в последнем случае надо исключить индекс свободного элемента вектора из траектории «свободное место» и включить этот индекс в траекторию «список». Наоборот, при имитации удаления элемента из списка необходимо исключить индекс этого элемента из траектории «список» и включить его в траекторию «свободное место».

Включение (или исключение) индекса в ту или иную траекторию означает «стирание» какого-то количества старых и «рисование» какого-то количества новых стрелок. В программе стрелкам соответствуют ссылки (т. е. элементы вектора ссылок). «Рисование» новой стрелки означает изменение состояния соответствующей ссылки и автоматическое «стирание» старой стрелки с тем же началом.

Будем представлять элементы вектора бусинками, а стрелки — веревочками. Тогда весь вектор распадется на две цепочки бус (для элементов имитируемого списка и свободного места соответственно). Переброс бусинки из свободного места в список означает, что надо

а) взять какую-нибудь бусинку из свободного места, т. е. разрезать веревочки справа и слева от бусинки и связать их между собой. После этого у нас будут две цепочки бус и одна бусинка «сама по себе» (рис. 13.5);

б) разрезать веревочку в том месте списка, куда нужно вставлять новую бусинку, и связать получившиеся концы с бусинкой (рис. 13.6). Разрезание веревочек (или, что что же самое, стирание стрелок) не более чем аналогия — на программном уровне веревочку

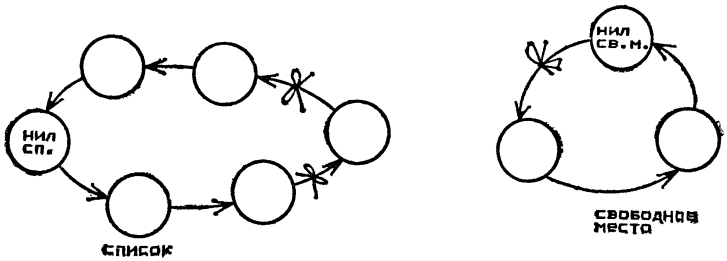


Рис. 13.6

нельзя разрезать, не связав ее одновременно с какой-то другой. Связывание же веревочек, т. е. установку ссылки с индексом И1 в состояние И2, мы выделим в отдельную программу

связать ⟨**вх** : И1, И2 : индекс⟩

Кроме того, выделим в отдельные программы все действия, связанные с траекторией «свободное место». Получение (захват) бусинки из свободного места, т. е. действия пункта а) выше, будем выполнять с помощью программы

захватить место ⟨**вых** : И : индекс⟩

где И — это индекс «бусинки» в векторе, а обратное действие — включение «отдельно висящей» бусинки в свободное место — с помощью программы

освободить место ⟨**вх** : И : индекс⟩

исполнитель Ограниченный Л1 список | на базе вектора
СП:
1. начать работу
2. сделать список пустым
3. список пуст : да/нет
4. есть свободное место : да/нет
5. нет свободного места : да/нет
6. установить указатель в начало списка
7. указатель в конце списка : да/нет
8. передвинуть указатель списка вперед
9. добавить элемент ⟨**вх** : Е⟩ за указателем списка

```
. 10. взять элемент списка за указателем в (вых : Е)
. 11. элемент списка за указателем                    ;: Е
. 12. удалить элемент списка за указателем
.
. — связать (вх : И1, И2 : индекс)
. — захватить место (вых : И : индекс)
. — освободить место (вх : И : индекс)
.
. 13. кончить работу
.
. константы:
. максинд = 100 | число элементов в векторе
.
. типы:
. индекс = 1..максинд
.
. объекты:
. указатель : запись(@до, @за : индекс)
.
. используемые исполнители:
. информация : вектор элементов типа Е с индексом 3..максинд
. @следующий : вектор элементов типа индекс
.                  с индексом типа индекс
.
. обозначения:
. нил списка          == 1
. нил свободного места == 2
.
. идеи реализации:
. Каждое состояние вектора определяет отображение f
. ("@следующий") множества индексов в себя:
.
.                  f : И -> @следующий(И)
.
. Мы реализуем список так, что в каждый момент времени ото-
. бражение f взаимно однозначно и имеет равно две периодические
. траектории „"список" и "свободное место". Эти траектории
. имеют вид:
.
. ...нил, f (нил), f (f (нил)), f (f (f (нил))),...,нил,...
.
. не пересекаются, покрывают все множество индексов и начи-
. наются с 1 (нил списка) и 2 (нил свободного места)
.
конец описаний |----------
```

```
программа начать работу
. дано      :  максинд > 2
. получить: | траектория "список" = нил списка ]
.           | траектория "свободное место" =
.           |              нил св. места, 3, 4, ..., максинд
. --------
.  информация.начать работу
.  @следующий.начать работу
.
.  связать       ⟨нил списка, нил списка⟩ | траектория «список»
.  указатель := (нил списка, нил списка)
.
.  цикл ∀ И ∈ нил свободного места..максинд—1 выполнять
.  .   связать ⟨И,И + 1⟩
.  конец цикла
.  связать ⟨максинд, нил свободного места⟩
конец программы

программа сделать список пустым
. дано      :
. получить: список пуст
. --------
.  установить указатель в начало списка
.  цикл пока список не пуст выполнять
.  .   удалить элемент списка за указателем
   конец цикла
конец программы

программа список пуст: да/нет ==
                        (@следующий(нил списка) = нил списка)

программа есть свободное место: да/нет ==
      (@следующий(нил свободного места) ≠ нил свободного места)
программа нет свободного места: да/нет ==
       (@следующий(нил свободного места) = нил свободного места)
программа установить указатель в начало списка ==
             указатель := (нил списка, @следующий(нил списка))
программа указатель в конце списка: да/нет ==
                                    (указатель. @за = нил списка)

программа передвинуть указатель списка вперед
. дано      : указатель не в конце списка
. получить:
. --------
.  указатель.@до := указатель.@за
.  указатель.@за := @следующий(указатель.@за)
конец программы
```

```
программа добавить элемент ⟨вх : Е⟩ за указателем списка
. дано     : есть свободное место
. получить:
. --------
.  захватить место ⟨вых : И⟩   | из траектории "свободное место"
.
.  связать ⟨указатель.@до, И⟩ | добавить элемент с индексом И
.  связать ⟨И, указатель.@за⟩ | в траекторию "список".
.  указатель.@за := И         | установить указатель
.  информация(И) := Е         | разместить информацию
конец программы

программа взять элемент списка за указателем в ⟨вых : Е⟩
. дано     : указатель не в конце списка
. получить:
. --------
.  Е := элемент списка за указателем
.  удалить элемент списка за указателем
конец программы

программа элемент списка за указателем :: Е
. дано     : указатель не в конце списка
. получить:
. --------
.  ответ == информация(указатель.@за)
конец программы

программа удалить элемент списка за указателем
. дано     : указатель не в конце списка
. получить:
. --------
.  И := указатель.@за                    | удалить элемент век-
.  указатель.@за := @следующий(И)        | тора за указателем из
.  связать ⟨указатель.@до,указатель.@за⟩ | траектории "список"
.  освободить место ⟨вх : И⟩  | добавить его в свободное место
конец программы

программа связать ⟨вх : И1,И2 : индекс⟩ == @следующий(И1) := И2

программа захватить место ⟨вых : И : индекс⟩
. дано     : есть свободное место
. получить:
. --------
.  И := @следующий(нил свободного места)
.  связать ⟨нил свободного места, @следующий(И)⟩
конец программы
```

```
программа освободить место (вх: И: индекс)
. дано      :
. получить:
. --------
. связать (И, @следующий(нил свободного места))
. связать (нил свободного места, И)
конец программы

программа кончить работу ==
          информация.кончить работу; @следующий.кончить
                                                работу
конец исполнителя | ==========================================
```

ЗАДАЧИ И УПРАЖНЕНИЯ

1. Каково максимальное число элементов Л1-списка в приведенной выше реализации?

2. Пусть дано, что указатель находится в конце списка. Реализуйте при этом условии программу «сделать список пустым» так, чтобы число элементарных операций с вектором не зависело ни от числа элементов списка, ни от числа элементов вектора (т. е. не пользуясь ни циклами, ни рекурсией).

3. Модифицируйте реализацию Л1-списка так, чтобы получить реализацию ограниченного Л2-списка.

Указание. Введите еще один вектор ссылок «@предыдущий» и реализуйте Л2-список так, чтобы отображения «@следующий» и «@предыдущий» были взаимно обратны. Постарайтесь, чтобы при реализации Л2-списка как можно большее число программ Л1-списка осталось без изменений.

4. В рамках реализации Л2-списка (задача 3) реализуйте программу «сделать список пустым» так, чтобы число операций было константой, т. е. не пользуясь ни циклами, ни рекурсией.

В задачах 5—8 указанные в задаче структуры требуется реализовать ссылочным образом на базе одного вектора элементов типа Е и одного или двух векторов ссылок. Слова «ограниченные в совокупности» означают, что эти структуры имеют общие для всех них предписания «начать работу»/«кончить работу» и «есть свободное место»/«нет свободного места». Если надо реализовать несколько одноименных структур (три стека, 133 стека, 100 множеств), то это значит, что во всех предписаниях кроме «есть свободное место»/«нет свободного места» и «начать работу»/«кончить работу» есть еще один параметр — номер структуры (аналогично задаче 13 предыдущего раздела). Кроме вектора элементов типа Е и одного или двух векторов ссылок, как обычно, можно использовать любое количество объектов простых типов и записей из них.

5. Три стека, ограниченные в совокупности.

6. 133 стека, ограниченные в совокупности.

7. Стек, очередь и Л2-список, ограниченные в совокупности.

8. 100 множеств, ограниченные в совокупности.

Указание. Объедините элементы отдельных структур в отдельные траектории, а оставшиеся элементы — в траекторию «свободное место». Другими словами, для реализации n структур используйте n + 1 траектории (например, в задаче 6 используйте 134 траектории, а в задаче 8 — 101 траекторию).

14. Три способа реализации множества на базе вектора. Последовательный поиск, бинарный поиск, хеширование

Непрерывная реализация. Последовательный поиск. Основная идея непрерывной реализации, как мы знаем, состоит в том, чтобы хранить элементы имитируемой структуры в векторе единым слитным куском, а для задания порядка элементов структуры использовать порядок их расположения в векторе. Для множества порядок элементов не существен, и остается лишь расположить элементы множества в векторе непрерывным куском, например «прижав» их к началу вектора (аналогично расположению элементов стека в разд. 12). Правую границу этого непрерывного куска (т. е. число элементов множества) легко задать с помощью объекта

мощность : 0..максинд

Определение принадлежности элемента множеству сводится к просмотру начального куска вектора — просмотру элементов с индексами от 1 до мощность. При имитации удаления элемента из множества нужно прежде всего выяснить, есть ли удаляемый элемент в множестве, и если есть, то в каком элементе вектора он расположен. Другими словами, перед удалением принадлежащего множеству элемента нужно сначала определить индекс соответствующего элемента вектора. Мы будем это делать с помощью программы

искать ⟨**вх** : е, **вых** : нашли : да/нет, **вых** : и : индекс⟩

Кроме того, надо позаботиться, чтобы при удалении элемента из середины оставшиеся элементы множества по-прежнему лежали слитным куском. Естественным кажется решение переместить все элементы множества, лежащие правее удаляемого, на один элемент влево по вектору. Эта операция, однако, является массовой (число выполняемых при этом элементарных действий с вектором заранее неизвестно — оно зависит от числа перемещаемых элементов). Поэтому поступим по-другому — возьмем последний (самый правый в векторе) элемент множества и переместим его в образовавшуюся

«дырку». Заметьте, что такое решение возможно только потому, что порядок элементов множества в векторе не важен.

```
исполнитель Ограниченное множество | на базе вектора
. СП:
.  1. начать работу
.  2. сделать множество пустым
.  3. множество пусто                                  : да/нет
.  4. есть свободное место                             : да/нет
.  5. нет свободного места                             : да/нет
.  6. элемент ⟨вх : е : Е⟩ принадлежит множеству       : да/нет
.  7. добавить элемент ⟨вх : е : Е⟩ в множество
.  8. удалить элемент ⟨вх : е : Е⟩ из множества
.  9. взять какой нибудь элемент множества в ⟨вых : е : Е⟩
. —  искать ⟨вх : е : Е, вых : нашли : да/нет, вых : и : индекс⟩
. 10. кончить работу
.
. константы:
.  максинд = 100
.
. типы:
.  индекс = 1..максинд
.
. объекты:
.  мощность : 0..максинд | число элементов множества
.
. идеи реализации:
.  Множество хранится "прижатым" к началу вектора, т. е. в эле-
.  ментах вектора с индексами 1..мощность. При удалении эле-
.  мента в получившуюся дырку переставляется элемент, занимав-
.  ший место с индексом мощность
.
. используемые исполнители:
.  вектор : вектор элементов типа Е с индексом типа индекс
.
конец описаний |----------------------------------

программа начать работу == вектор.начать работу; мощность := 0
программа сделать множество пустым == мощность := 0
программа множество пусто : да/нет == (мощность = 0)
программа есть свободное место : да/нет == (мощность<максинд)
программа нет свободного места : да/нет == (мощность=максинд)

программа элемент ⟨вх : е : Е⟩ принадлежит множеству : да/нет
дано
```

```
. получить:
. --------
. искать ⟨вх: е, вых: ответ, и⟩
конец программы

программа добавить элемент ⟨вх: е: Е⟩ в множество
. дано     : есть свободное место
. получить:
. --------
. искать ⟨вх: е, вых: нашли, и⟩
. если не нашли то
. . мощность.увеличить на 1
. . вектор(мощность) := е
. конец если
конец программы

программа удалить элемент ⟨вх: е: Е⟩ из множества
. дано     :
. получить:
. --------
. искать ⟨вх: е, вых: нашли, и⟩
. если нашли то
. . вектор(и) := вектор(мощность)
. . мощность.уменьшить на 1
. конец если
конец программы

программа взять какой нибудь элемент множества в ⟨вых: е: Е⟩
. дано     : множество не пусто
. получить:
. --------
. е := вектор(мощность)
. мощность.уменьшить на 1
конец программы

программа искать ⟨вх: е: Е, вых: нашли: да/нет, вых: и: индекс⟩
. дано     :
. получить:
. --------
. цикл ∀ и ∈ 1..мощность пока вектор(и) ≠ е выполнять
. . ничего не делать
. конец цикла
. нашли := (и ≠ неопр)
конец программы

программа кончить работу == вектор.кончить работу
конец исполнителя |==========================
```

Операция «искать» в этой реализации является массовой: чтобы узнать, что элемент e не принадлежит множеству, надо перебрать все элементы множества. Если e принадлежит множеству, то число операций зависит от близости расположения e к началу вектора. Таким образом, если в множестве n элементов, то при поиске принадлежащего множеству элемента максимальное число элементарных операций с вектором равно n, среднее — n/2. При поиске не принадлежащего множеству элемента число операций всегда равно n. В приложениях, где определение принадлежности элемента множеству является основной операцией, а сами множества достаточно велики, приведенная выше реализация может оказаться неудовлетворительной. Вся последующая часть этого раздела посвящена задаче уменьшения числа операций при поиске элемента в множестве.

Непрерывная реализация. Бинарный поиск. Идею бинарного поиска (или поиска методом деления пополам) поясним на следующем примере: пусть задуманы 1000 каких-то натуральных чисел — число номер 1, число номер 2, ..., число номер 1000. По условиям игры мы имеем право попросить назвать любое задуманное число, например 5-е, 28-е, 976-е и др. Наша задача — задача поиска — узнать, есть ли среди задуманных некоторое конкретное число e, например 2718.

Если про порядок задуманных чисел ничего не известно и число e может оказаться на любом месте, то у нас нет иного способа, кроме как перебирать все задуманные числа (например, последовательно узнавать 1-е, 2-е, 3-е, ..., 999, 1000-е задуманные числа) и смотреть, не попадется ли среди них e. Это и есть последовательный поиск, при котором в наихудшем случае понадобится задать 1000 (мощность множества) вопросов, прежде чем мы узнаем, есть ли там e.

Если, однако, условия игры предполагают, что задуманные числа упорядочены по возрастанию, т. е. первое число меньше второго, второе меньше третьего и т. д., можно вести себя по-другому. Узнаем сначала 500-е задуманное число; пусть это x. Ясно, что если $e < x$, то дальше надо искать e среди чисел с 1-го по 499-е, если $e = x$, то мы его нашли, а если $e > x$, то надо искать среди чисел с 501-го по 1000-е. Тем самым, задав всего один вопрос, мы либо найдем e, либо уменьшим количество подозреваемых чисел как минимум вдвое. Следующим вопросом количество подозреваемых можно уменьшить еще вдвое. Например, если после первого вопроса остались числа с 501-го по 1000-е, надо узнать 750-е число и сравнить его с e, после чего останется не более 250 подозреваемых чисел, и т. д.

Поскольку при каждом вопросе круг подозреваемых вдвое сужается, то примерно за $\log_2(n)$ (т. е. за 10) вопросов мы либо попадем на e, либо дойдем до ситуации, когда подозреваемых чисел не осталось. Последнее означает, что число e множеству задуманных чисел не принадлежит. Заметьте, что при $n = 1000$ бинарный поиск в 100 раз быстрее последовательного, а при $n = 1000\,000$ в 50 000 раз.

То, что в предыдущем рассуждении речь шла о натуральных числах, не очень важно. Если, например, задуманы 1000 слов и нужно узнать, есть ли среди них некоторое конкретное слово, то можно применить тот же самый метод деления пополам. Только сравнивать надо будет не числа, а слова (например, в алфавитном порядке). Единственное, что действительно требуется от типа E, — это чтобы он был упорядоченным, т. е. чтобы над объектами типа E были допустимы операции $<$, $\leqslant$, $\geqslant$, $>$.

Изменим теперь реализацию множества в соответствии с описанными выше идеями. Во-первых, надо обеспечить сохранение порядка элементов в множестве при добавлении и удалении; во-вторых, надо заменить в программе «искать» последовательный поиск на бинарный.

Начнем с реализации программы добавления элемента e в множество. Если e уже принадлежит множеству, то ничего делать не надо. Если же e множеству не принадлежит, то надо найти место для e между элементами множества, т. е. понять, куда вставлять e, и сдвинуть все элементы множества правее этого места (элементы большие e) вправо по вектору (рис. 14.1).



Рис. 14.1

Поиск места для вставки e можно совместить с бинарным поиском e в множестве: если последнее подозреваемое число оказалось меньше e, то e надо вставлять правее этого числа; если равно e, то e принадлежит множеству; если больше e, то e надо вставлять левее. Поэтому изменим назначение программы «искать», а именно потребуем, чтобы в случае «не нашли» эта программа выдавала индекс того элемента вектора, куда надо вставлять e. Тогда программа «добавить» реализуется следующим образом:

```
программа добавить элемент ⟨вх: e: E⟩ в множество
. дано      : есть свободное место
. получить:
. --------
. искать ⟨вх: e, вых: нашли, ие⟩
. если не нашли то
. . цикл ∀ и ∈ ие..мощность реверс выполнять
. . . вектор(и + 1) := вектор(и)
. . конец цикла
. . мощность.увеличить на 1
. . вектор(ие) := e
. конец если
конец программы
```

В цикле сдвига элементов множества вправо стоит слово **реверс**. Это значит, что индексы из диапазона ие .. мощность перебираются в обратном порядке — справа налево, т. е. тело цикла сначала выполняется при и = мощность, затем при и = мощность — 1 и т. д., последний раз — при и = ие. Такой порядок сдвига элементов необходим, так как при переборе элементов слева направо при пересылке, например, элемента с индексом ие в элемент с индексом ие + 1 старое значение элемента с индексом ие + 1 было бы потеряно, а ведь именно его надо пересылать в элемент с индексом ие + 2.

В программе «удалить», где элементы множества надо сдвигать влево, наоборот, нужно использовать обычный цикл:

```
программа удалить элемент ⟨вх: e: E⟩ из множества
. дано      :
. получить:
. --------
. искать ⟨вх: e, вых: нашли, ие⟩
. если нашли то
. . цикл ∀ и ∈ ие..мощность — 1 выполнять
. . . вектор(и) := вектор(и + 1)
. . конец цикла
. . мощность.уменьшить на 1
. конец если
конец программы
```

Реализуем теперь программу «искать» для бинарного поиска с учетом сделанных выше замечаний.

Несмотря на простоту идеи, реализовать бинарный поиск без ошибок не так просто. Поэтому подойдем к реализации более формально, чем обычно, и явно выпишем все инварианты и утверждения

ния по схеме проектирования цикла с помощью инварианта. Мы приведем два варианта программы бинарного поиска. Первый из них в точности соответствует описанному выше процессу поиска числа e среди задуманных. В соответствии с этим описанием у нас будет диапазон a .. b номеров подозреваемых чисел (диапазон индексов подозреваемых элементов вектора). В начальный момент (до первого вопроса) элемент e может оказаться на любом месте и подозреваемыми являются все индексы элементов множества:

a = 1, b = мощность.

После каждого вопроса этот диапазон уменьшается как минимум вдвое. Процесс заканчивается либо в момент обнаружения элемента e, либо, если e множеству не принадлежит, когда подозреваемых индексов не останется (диапазон a .. b станет пустым). Выберем в качестве инварианта следующее утверждение:

b ⩾ a − 1 **и**
элементы с индексами вне диапазона a..b вне подозрений.

Вторая часть этого утверждения означает, что элементы множества, расположенные левее a, меньше e, а элементы множества, расположенные правее b, больше e. С учетом упорядоченности элементов вектор(1), вектор(2), ..., вектор(мощность) инвариант можно записать так:

b ⩾ a − 1 **и**
(a = 1 **или** e > вектор(a − 1)) **и**
(b = мощность **или** e < вектор(b + 1)).

```
программа искать ⟨вх: e: E, вых: нашли: да/нет, ие: индекс⟩
. дано      : | элементы вектора с индексами 1..мощность возрастают
. получить: | ие такое, что элементы вектора с индексами 1..ие − 1
.           | меньше e, а элементы с индексами ие..мощность ⩾ e,
.             нашли = (ие ⩽ мощность и вектор(ие) = e) | ⇔ e ∈ M
. --------
. нашли := нет
. если мощность = 0 то ие := 1 иначе
. . a := 1; b := мощность | подозреваем элементы с индексами a..b
. . цикл
. . . инв: b ⩾ a − 1                    и
. . . (a = 1 или e > вектор(a − 1)) и
. . . (b = мощность или e < вектор(b + 1))
. . . | элементы с индексами вне a..b вне подозрений
. . . пока не нашли и a ⩽ b | (не нашли и есть подозреваемые)
. . . выполнять
. . . с := частное(a + b, 2); утв: a ⩽ c ⩽ b
```

```
. . . выбор
. . . . при e < вектор(c) ⇒ b := c − 1
. . . . при e = вектор(c) ⇒ a := c; b := c; нашли := да
. . . . при e > вектор(c) ⇒ a := c + 1
. . . конец выбора
. . конец цикла
. .   утв: (a = 1                  или e > вектор(a − 1)) и
. .         (a = мощность + 1 или e ⩽ вектор(a))
. .
. .  |это утверждение можно переписать в виде:
. .
. .   утв: (a = 1                  и e ⩽ вектор(1))                  или
. .         (1 < a ⩽ мощность и вектор(a−1) < e⩽вектор(a)) или
. .         (a = мощность + 1 и вектор(мощность) < e)
. .  ие := a
. конец если
конец программы
```

Условие окончания цикла «нашли **или** a > b» можно переформулировать в виде

(a = b **и** e = вектор(a)) **или** a = b + 1.

После этого постусловие (утверждение после слов **конец цикла**) можно строго формально вывести из инварианта и условия окончания (сделайте это).

Осталось проверить, что независимо от успешности или безуспешности поиска цикл рано или поздно заканчивается. Это следует из того, что при каждом выполнении тела цикла либо обнаруживается искомый элемент, либо разность b − a уменьшается по крайней мере на 1.

Приведенная выше программа обладает тем недостатком, что вывод постусловия из инварианта и условия окончания не является очевидным. Это вызвано тем, что мы строили программу в соответствии с *интуитивным алгоритмом бинарного поиска*, а потом пытались показать ее правильность, используя схему проектирования цикла с помощью инварианта. Можно поступить по-другому — отказаться от интуитивного алгоритма и строить программу, *исходя из схемы проектирования цикла с помощью инварианта*. В соответствии с этой схемой вначале надо выписать постусловие (конечную цель):

вектор(ие − 1) < e ⩽ вектор(ие).

Это постусловие, впрочем, не вполне верно. Например, если множество пусто или e ⩽ вектор (1), то нам надо получить ие = 1, а элемента с индексом ие = 1 вообще не существует. Но пока рас-

смотрим только общий случай с выписанным выше постусловием. Наша цель сейчас — придумать общую стратегию, инвариант и условие окончания, которые приводили бы к этому постусловию. Общая стратегия у нас фиксирована — это бинарный поиск. Значит, будет какой-то подозреваемый диапазон a..b, который будет примерно вдвое уменьшаться при каждом выполнении тела цикла. Но тогда с точностью до обозначений можно переписать постусловие в терминах изменяющегося в цикле объекта — диапазона a .. b:

вектор(a) $<$ e $\leqslant$ вектор(b) **и** a $=$ b $-$ 1.

Дальше уже просто — теперь постусловие, очевидно, распадается на инвариант

вектор(a) $<$ e $\leqslant$ вектор(b)

и условие окончания

a $=$ b $-$ 1,

а полная реализация с учетом особых случаев выглядит так:

```
программа искать (вх: e: E, вых: нашли: да/нет, ие: индекс)
. дано      : | элементы вектора с индексами 1.. мощность
.               возрастают
. получить: | ие такое, что элементы вектора с индексами 1..ие — 1
.           | меньше e, а элементы с индексами ие..мощность ⩾ e
.           | нашли = да ⇔ e ∈ множеству
. --------
. выбор
. . при мощность = 0 или e ⩽ вектор(1) ⇒ ие := 1
. . при вектор(мощность) < e           ⇒ ие := мощность + 1
. . иначе ⇒
. .     утв: множество не пусто и
. .          вектор(1) < e ⩽ вектор(мощность)
. .
. .   a := 1; b := мощность
. .   цикл
. .   . инв: вектор(a) < e ⩽ вектор(b)
. .   . пока a ≠ b — 1 выполнять
. .   .  c := (a + b)/2; утв: a < c < b
. .   .  выбор
. .   .  . при вектор(c) < e ⇒ a := c
. .   .  . при вектор(c) = e ⇒ b := c; a := b — 1
. .   .  . при вектор(c) > e ⇒ b := c
. .   .  конец выбора
. .   конец цикла
```

```
. .      утв: вектор(а) < е ⩽ вектор(b) и b = a + 1
. .   ие := b
. конец выбора
. нашли := (ие ⩽ мощность и вектор(ие) = е)
конец программы
```

Завершение цикла в этой программе также обеспечивается монотонным уменьшением величины b — a.

Количество операций при бинарном поиске по сравнению с последовательным сокращается с n до $\log_2(n)$. Однако заметим, что при этом добавление и удаление элемента множества становятся массовыми операциями, требующими порядка n элементарных действий с вектором. Таким образом, реализация множества с помощью бинарного поиска эффективна только в том случае, если множество меняется редко, а принадлежность того или иного элемента множеству надо определять достаточно часто. Существуют ссылочные реализации множества на базе вектора, использующие бинарный поиск и позволяющие добавлять или удалять элементы не более, чем за $C\log_2(n)$ элементарных операций. Мы, однако, эти реализации рассматривать не будем.

Битовая реализация множества элементов типа 1 .. максинд. Итак, и последовательный, и бинарный поиск имеют и свои достоинства и недостатки. Выигрыш в скорости при бинарном поиске достигнут, образно говоря, за счет того, что мы «пожертвовали» добавлением и удалением, которые теперь стали массовыми. Кроме того, несмотря на «жертвы», сам бинарный поиск также остался массовой операцией — количество действий с вектором по-прежнему зависит от n, хотя теперь эта зависимость логарифмическая, а не линейная. Естественно, возникает вопрос: а существует ли вообще реализация множества, при которой поиск не является массовой операцией? И можно ли при этом не «жертвовать» добавлением и удалением?

Есть по крайней мере один частный случай, когда такая реализация не только существует, но и очевидна, — это случай, когда число значений типа Е невелико, и это число можно использовать в качестве максинд. Например, если Е = 1 .. максинд, то можно реализовать множество элементов типа Е на базе вектора элементов типа да/нет с индексом типа 1 .. максинд (т. е. с индексом типа Е), просто запоминая в каждой компоненте вектора, принадлежит или нет соответствующий элемент множеству:

вектор(и) = да ⟺ элемент и принадлежит множеству.

В этой реализации элементы вектора могут принимать лишь одно из двух значений — да или нет. Эти элементы часто называют

битами (от англ. binary digit — двоичная цифра), а реализацию в целом — *битовой.*

```
исполнитель множество | элементов типа 1..максинд на базе вектора
. СП:
.  1. начать работу
.  2. сделать множество пустым
.  3. множество пусто                                  :да/нет
.  4. элемент ⟨вх:е:Е⟩ принадлежит множеству           :да/нет
.  5. добавить элемент ⟨вх:е:Е⟩ в множество
.  6. удалить элемент ⟨вх:е:Е⟩ из множества
.  7. взять какой нибудь элемент множества в ⟨вых:е:Е⟩
.  8. кончить работу
.
. константы:
.  максинд = 100
.
. типы:
.  Е = 1..максинд
.
. используемые исполнители:
.  вектор : вектор элементов типа да/нет с индексом типа Е
.
. идеи реализации:
.     вектор(и) = да ⇔ и ∈ множеству
.
конец описаний | ----------------------------------

программа начать работу ==
                  вектор.начать работу; сделать множество пустым
программа сделать множество пустым
. дано      :
. получить:
. --------
. цикл ∀ и ∈ Е выполнять
. .   вектор(и) := нет
. конец цикла
конец программы

программа множество пусто:да/нет
. дано      :
. получить:
. --------
.  нашли := нет
.  цикл ∀ и ∈ Е пока не нашли выполнять
```

```
. . нашли := (вектор(и) = да)
. конец цикла
. ответ := не нашли
конец программы

программа элемент ⟨вх: е: Е⟩ принадлежит множеству: да/нет ==
                                                (вектор(е) = да)
программа добавить элемент ⟨вх: е: Е⟩ в множество ==
                                                вектор(е) := да
программа удалить элемент ⟨вх: е: Е⟩ из множества ==
                                                вектор(е) := нет

программа взять какой нибудь элемент множества в ⟨вых: е: Е⟩
. дано      : множество не пусто
. получить:
. --------
. цикл ∀ е ∈ Е пока вектор(е) = нет выполнять
. . ничего не делать
. конец цикла
.   утв: вектор(е) = да
. удалить элемент ⟨вх: е⟩ из множества
конец программы

программа кончить работу == вектор.кончить работу
конец исполнителя |==========================
```

В этой реализации массовыми являются операции «сделать пустым», «множество пусто» и «взять». Анализ пустоты множества легко сделать не массовой операцией, если ввести глобальный объект «мощность» и использовать его для хранения числа элементов в множестве аналогично тому, как это было сделано в последовательном и бинарном поиске. Операцию «взять» можно сделать не массовой, если использовать еще один вектор ссылок с индексами от 0 до максинд, с помощью этого вектора объединить элементы множества в кольцо, начинающееся со служебного элемента нил = 0, и при добавлении/удалении элементов множества включать/исключать их из этого кольца так же, как мы освобождали/захватывали свободное место в ссылочной реализации Л1-списка. Операция «сделать множество пустым» по сути реализации является массовой, но обычно она достаточно редка, и ее массовостью можно пренебречь.

Подчеркнем еще раз, что битовая реализация множества элементов типа Е применима только в случаях, когда значения типа Е можно использовать в качестве индексов некоторого вектора (т. е. когда тип Е является отрезком или перечислением). Но если уж битовую реализацию удалось применить, то добавление/удаление

и анализ принадлежности элемента множеству в ней проводятся за одну операцию с вектором — вряд ли этот результат можно превзойти!

Хеширование. Итак, пока что наша мечта — поиск, не являющийся массовой операцией, — в общем случае осталась недостигнутой. Вспомним самое начало — непрерывную реализацию множества на базе вектора (последовательный поиск). Мы говорили, что основное отличие этой реализации состоит в том, что нам приходится *искать* элемент e в векторе, в то время как при реализациях стека, очереди, дека, последовательности индекс доступного элемента структуры (т. е. индекс вершины стека, начала очереди, начала или конца дека, очередного элемента последовательности)

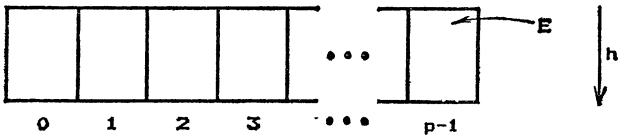


Рис. 14.2

предопределен самой структурой. Вот если бы по элементу множества можно было сразу указать индекс компоненты вектора, где этот элемент лежит! Ведь для решения именно этой задачи мы ввели программу «искать». Увы, такое возможно только в том случае, если число различных значений типа E равно количеству элементов в некотором векторе — тогда значения типа E можно (по крайней мере теоретически) взаимно однозначно отобразить на множество индексов, а в компонентах вектора хранить информацию о том, принадлежит соответствующее значение типа E множеству или нет. Это и есть битовая реализация множества (в примере выше отображение было тождественным).

Какова, однако, конечная цель, т. е. ради чего мы избегаем массовых операций при поиске? Очевидно, это скорость выполнения программ при работе с достаточно большими множествами. Но для достижения высокой скорости вовсе не обязательно, чтобы поиск *всегда* выполнялся быстро; достаточно, если он будет быстрым *почти всегда*, т. е. надо интересоваться количеством операций не в наихудшем, а в некотором типичном случае. А тогда можно воспользоваться идеей об отображении значений типа E на множество индексов, не требуя взаимной однозначности отображения.

Рассмотрим на простейшем примере, как такое отображение, называемое *хеш-функцией*, может ускорить работу с множеством.

Пусть h: E → 0 .. p — 1 отображение множества значений типа E на множество из p элементов 0, 1, 2, ..., p — 1 (рис. 14.2). Вертикальные полосы над точками 0, 1, 2, ..., p — 1 изображают

множества-слои $h^{-1}(0)$, $h^{-1}(1)$, $h^{-1}(2)$, ..., $h^{-1}(p-1)$ соответственно.

Напомним, что исполнитель «множество элементов типа Е» моделирует конечное подмножество М множества значений типа Е.

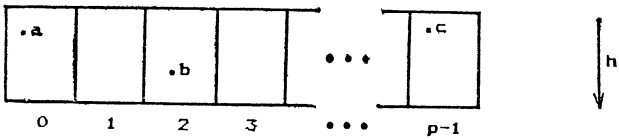


Рис. 14.3

Предположим, что М состоит из трех элементов a, b, c и что в каждом вертикальном слое лежит не более одного элемента М (рис. 14.3). Тогда можно применить идеи битовой реализации — введем вектор «бит» элементов типа «да/нет» с индексом типа

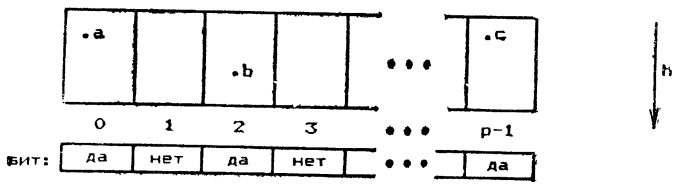


Рис. 14.4

0 .. p — 1 и запишем в элементы этого вектора да, если в слое над элементом есть точка из М, и нет в противном случае (рис. 14.4).

Пусть теперь надо определить, принадлежит ли элемент e типа Е множеству М. Можно действовать так: вычислим i = h(e) и посмотрим, чему равно бит(i). Если бит(i) = нет, то элемент e заведомо не принадлежит М (так как в слое над i элементов М нет).

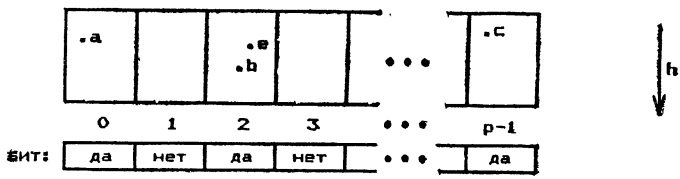


Рис. 14.5

Если же бит(i) = да, то про e ничего сказать нельзя — нужна дополнительная информация. Например, если h(e) = 2, то e может совпадать с b, а может и не совпадать (рис. 14.5).

Поскольку мы предположили, что в каждом вертикальном слое лежит не более одного элемента M, то для задания этой дополнительной информации достаточно ввести еще один вектор «инф» элементов типа E — если слой над i содержит точку $m \in M$, то инф(i) = m (рис. 14.6).

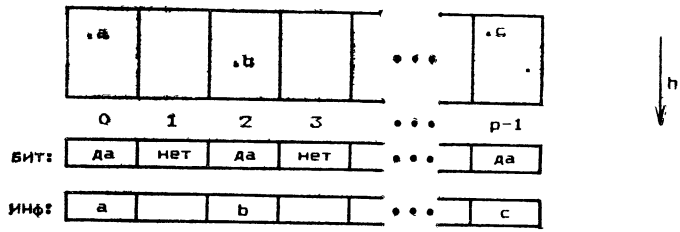


Рис. 14.6

Итак, если сужение h на M не склеивает точек, то

$e \in M \Leftrightarrow$ бит(h(e)) = да и инф(h(e)) = e.

Следовательно, число операций при поиске не зависит от числа элементов в M.

На практике, однако, h фиксируется заранее, а множество M меняется в процессе работы. Поэтому гарантировать инъективность ограничения h на M нельзя — в некоторых слоях может оказаться более одного элемента M. Например, для h, $d \in M$ может оказаться, что h(b) = h(d) = 2. Что записать в инф(2) в этом случае (рис. 14.7)?

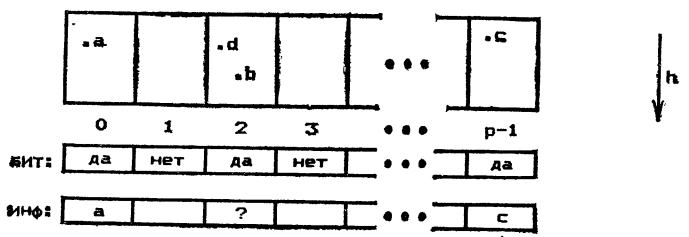


Рис. 14.7

Хотелось бы записать в инф(2) одновременно b и d, но это невозможно — элемент вектора «инф» может хранить только одно значение типа E. Выход из этой ситуации состоит в том, чтобы вместо вектора «инф» элементов типа E использовать вектор множеств элементов типа E (рис. 14.8). Вектор «бит» при этом становится ненужным, так как про множество можно узнать, пусто оно или нет.

Идею хеширования можно пояснить на примере библиотечного каталога. Пусть у нас есть некоторое множество книг (а точнее, описывающих их карточек) и нам часто приходится узнавать, есть в этом множестве некоторая конкретная книга или нет. Простейшее решение состоит в том, чтобы сложить все карточки в произвольном порядке, а при поиске конкретной книги перебирать их все друг за другом. Это и будет непрерывная реализация с последовательным поиском. Второе решение состоит в том, чтобы расположить карточки в алфавитном порядке и искать конкретную книгу методом деления пополам так, как описано в разделе про

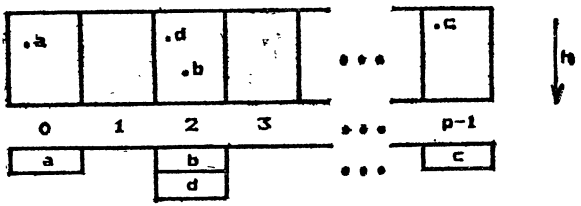


Рис. 14.8

бинарный поиск. Можно, однако, рассмотреть какую-нибудь функцию на множестве таких карточек, например сопоставить каждой карточке первую букву названия книги (это и есть хеш-функция), а сами карточки разложить по отдельным ящикам — в один ящик положить все карточки, начинающиеся с А, в другой — с Б и т. д. Тогда при поиске книги можно посмотреть на первую букву ее названия, взять соответствующий ящик и искать только в нем. (Название «хеширование» — от английского глагола hash — рассекать, рассеивать, разрезать, разбивать — как раз и означает такое разбиение множества на части, соответствующие значениям «рассеивающей» функции.)

Заметьте, что фактически получилась опять та же задача — надо найти конкретную карточку в некотором их множестве. Однако можно надеяться, что «подозреваемых» карточек стало в 30 (число значений хеш-функции) раз меньше. Эту новую задачу поиска в меньшем множестве можно решать любым образом — последовательным перебором всех карточек в ящике, упорядочением карточек в алфавитном порядке и бинарным поиском либо опять хешированием. Последнее означает, что надо придумать какую-то новую хеш-функцию, например сопоставить каждой карточке вторую букву названия или первую букву фамилии автора или число букв в названии и т. д., разгородить ящик на отделения, соответствующие значениям выбранной хеш-функции, и разложить карточки соответствующим образом.

Обратите внимание, что при применении хеш-функции можно лишь *надеяться* на уменьшение числа подозреваемых карточек. Эти надежды базируются на предположении, что если разбить данное конкретное множество карточек в соответствии со значениями данной конкретной хеш-функции, то в каждом ящике карточек окажется примерно поровну. Таким образом, эффективность хеширования целиком зависит от того, сколь равномерно данная хеш-функция разбивает данное множество. В наихудшем случае (например, если картотека содержит данные только о книгах, начинающихся со слов «К вопросу о ...») все карточки могут попасть в один ящик (с буквой К) и хеширование ни на йоту не облегчит нашу задачу. Тем не менее, как показывает пример библиотечных каталогов, если хеш-функция достаточно хороша, то применение хеширования позволяет во много раз повысить скорость поиска.

Вернемся теперь на более формальные, программистские рельсы. Пусть имеется хеш-функция h: $E \to 0 \,..\, p-1$, принимающая p различных значений от 0 до $p-1$. Описанный выше механизм хеширования означает, что при однократном применении хеш-функции множество M элементов типа E разбивается на p подмножеств:

$$M_i = \{e \in M : h(e) = i\}, \quad i = 0, 1, 2, \ldots, p-1,$$

после чего все операции над M сводятся к операциям над соответствующим M_i. Действительно, в соответствии с определением M_i

$$e \in M \Leftrightarrow e \in M_{h(e)},$$

добавление элемента e в множество M означает, что e надо добавить в $M_{h(e)}$ и т. д. Соответствующая реализация множества на базе p множеств выглядит следующим образом:

исполнитель Ограниченное множество | на базе p множеств
СП:
1. начать работу
2. сделать множество пустым
3. множество пусто :да/нет
4. есть свободное место :да/нет
5. нет свободного места :да/нет
6. элемент ⟨**вх**: e: E⟩ принадлежит множеству :да/нет
7. добавить элемент ⟨**вх**: e: E⟩ в множество
8. удалить элемент ⟨**вх**: e: E⟩ из множества
9. взять какой нибудь элемент множества в ⟨**вых**: e: E⟩
   — искать непустое ⟨**вых**: нашли: да/нет, n: номер⟩
   — h ⟨**вх**: e⟩: номер | хеш-функция
10. кончить работу

```
. константы:
.  p = 1019
.
. типы:
.  номер = 0..p — 1
.
. используемые исполнители:
.  p множеств (ПМ)
.
. идеи реализации:
.     e ∈ множеству ⇔ e ∈ ПМ.множество с номером ⟨h⟨e⟩⟩
.
конец описаний |----------------------------------

программа начать работу == ПМ.начать работу
программа сделать множество пустым
. дано     :
. получить:
. --------
.  цикл ∀ n ∈ номер выполнять
.  .  ПМ.сделать множество ⟨вх : n⟩ пустым
. конец цикла
конец программы

программа множество пусто : да/нет
. дано     :
. получить:
. --------
.  искать непустое ⟨вых : нашли, n⟩
,  ответ := не нашли
конец программы

программа есть свободное место : да/нет ==
                                  ПМ.есть свободное место
программа нет свободного места : да/нет ==
                                  ПМ.нет свободного места
программа элемент ⟨вх : e : E⟩ принадлежит множеству : да/нет ==
            ПМ.элемент ⟨вх : e⟩ принадлежит множеству ⟨вх : h⟨e⟩⟩
программа добавить элемент ⟨вх : e : E⟩ в множество ==
              ПМ.добавить элемент ⟨вх : e⟩ в множество ⟨вх : h⟨e⟩⟩
программа удалить элемент ⟨вх : e : E⟩ из множества ==
              ПМ.удалить элемент ⟨вх : e⟩ из множества ⟨вх : h⟨e⟩⟩

программа взять какой нибудь элемент множества в ⟨вых : e : E⟩
. дано     : множество не пусто
. получить:
```

```
• --------
• искать непустое ⟨вых : нашли, n⟩
•   утв: нашли
• ПМ.взять какой нибудь элемент множества ⟨вх : n⟩ в ⟨вых : e⟩
конец программы

программа искать непустое ⟨вых : нашли : да/нет, n : номер⟩
• дано      :
• получить:
• --------
• цикл ∀ n ∈ номер пока ПМ.множество ⟨вх : n⟩ пусто выполнять
• . ничего не делать
• конец цикла
• нашли := (n ≠ неопр)
конец программы

программа кончить работу == ПМ.кончить работу
конец исполнителя | ============================
```

Реализацию хеш-функции, поскольку она зависит от приложений, мы опускаем. Заметим лишь, что очень часто хеш-функция имеет вид $h(e) = f(e) \bmod p$, где p — простое, а $f(e)$ функция с целыми значениями. Типичными примерами f являются функции «сумма цифр числа e» или «само число e» (для E = Z), «первый символ», «число символов», «сумма кодов символов» (для E = последовательность элементов типа символ) и др.

Функция «сумма кодов символов» на множестве последовательностей букв французского алфавита описана у Л. Н. Толстого (Война и мир, том 3, XIX):

«...Пьеру было открыто одним из братьев-масонов следующее, выведенное из Апокалипсиса Иоанна Богослова, пророчество относительно Наполеона.

В Апокалипсисе, главе тринадцатой, стихе восемнадцатом сказано «Зде мудрость есть; иже имать ум да почтет число зверино: число бо человеческо есть и число его шестьсот шестьдесят шесть».

И той же главы в стихе пятом: «И даны быша ему уста глаголюща велика и хульна; и дана бысть ему область творити месяц четыре — десять два».

Французские буквы, подобно еврейскому число-изображению, по которому первыми десятью буквами обозначаются единицы, а прочими — десятки, имеют следующее значение:

a	b	c	d	e	f	g	h	i	k	l	m	n	o	p
1	2	3	4	5	6	7	8	9	10	20	30	40	50	60

q	r	s	t	u	v	w	x	y	z
70	80	90	100	110	120	130	140	150	160

Написав по этой азбуке цифрами слова L'mpereur Napoleon, выходит, что сумма этих чисел равна 666-ти и что поэтому Наполеон есть тот зверь

о котором предсказано в Апокалипсисе. Кроме того, написав по этой азбуке слова quarinte deux, то есть предел, который был положен зверю глаголати велика и хульна, сумма этих чисел, изображающих quarante deux, опять равна 666-ти, из чего выходит, что предел власти Наполеона наступил в 1812-м году, в котором французскому императору минуло 42 года».

Изложение собственно хеширования на этом заканчивается. В программировании довольно часто максимальное число элементов имитируемого множества M меньше p. Таким образом, при достаточно хорошей хеш-функции каждое из подмножеств M_i содержит обычно не более одного элемента и поиск элемента в множестве осуществляется мгновенно независимо от способа реализации множеств M_i.

Для завершения хеш-реализации множества осталось реализовать исполнителя «p множеств (ПМ)» на базе одного вектора. Реализацию проведем ссылочным образом. В соответствии с идеями предыдущего раздела это значит, что у нас будет p + 1 траектории — по одной на каждое множество и одна для свободного места. Траектория n-го множества будет начинаться с индекса n (нил этого множества), траектория «свободное место» будет начинаться с индекса p (нил свободного места). Внутри каждого отдельного множества будем хранить элементы неупорядоченными и использовать последовательный поиск:

исполнитель p множеств (ПМ) | ограниченных в совокупности
. **СП:**
. 1. начать работу
. 2. сделать множество ⟨**вх**: n: номер⟩ пустым
. 3. множество ⟨**вх**: n: номер⟩ пусто :да/**нет**
. 4. есть свободное место :да/**нет**
. 5. нет свободного места :да/**нет**
. 6. элемент ⟨**вх**: e⟩ принадлежит множеству ⟨**вх**: n: номер⟩ :да/**нет**
. 7. добавить элемент ⟨**вх**: e: E⟩ в множество ⟨**вх**: n: номер⟩
. 8. удалить элемент ⟨**вх**: e: E⟩ из множества ⟨**вх**: n: номер⟩
. 9. взять какой нибудь элемент множества ⟨**вх**: n⟩ в ⟨**вых**: e: E
. — искать ⟨**вх**: e: E, n: номер, **вых**: нашли: **да/нет**, указатель⟩
. — связать ⟨**вх**: И1, И2: индекс⟩
. — захватить место ⟨**вых**: И: индекс⟩
. — освободить место ⟨**вх**: И: индекс⟩
. 10. кончить работу
.
. **константы:**
. p = 1019
. максинд = p + 1000
.
. **типы:**

```
  индекс = 0..максинд
  номер  = 0..p — 1
•
• используемые исполнители:
  информация: вектор Е(p + 1..максинд)
  @следующий: вектор индекс (индекс)
• обозначения:
  нил свободного места == p
•
• идеи реализации:
  Каждое состояние вектора "@следующий" определяет
  отображение f множества индексов в себя:
                f: и → @следующий(и).
  Мы реализуем "p множеств" так, что в каждый момент времени
  отображение f взаимно однозначно и имеет ровно p + 1 траек-
  тории. Множество с номером n определяется траекторией, про-
  ходящей через точку n (нил этого множества). Каждой точке i
  этой траектории кроме точки n отвечает элемент множества,
  лежащий в информация(i).
конец описаний |----------------------------------
```

```
программа начать работу
• дано     : максинд > p
• получить: |траектория каждого множества состоит из точки нил
            |траектория свободное место=
            |          нил свободного места .. максинд
• --------
• информация.начать работу
• @следующий.начать работу
•
• цикл ∀ n ∈ номер выполнять
• .  связать ⟨n,n⟩ | траектории множеств
• конец цикла
•
• цикл ∀И ∈ нил свободного места .. максинд-1 выполнять
• .  связать ⟨И,И + 1⟩
• конец цикла
• связать ⟨максинд,нил свободного места⟩
конец программы

программа сделать множество ⟨вх: n: номер⟩ пустым
• дано
• получить: множество ⟨вх: n⟩ пусто
• --------
```

```
. цикл пока множество ⟨вх : n⟩ не пусто выполнять
. .   И := @следующий (n)
. .   связать ⟨n, @следующий (И)⟩
. .   освободить место ⟨вх : И⟩
. конец цикла
конец программы

программа множество ⟨вх : n : номер⟩ пусто : да/нет ==
                                        (@следующий (n) = n)
программа есть свободное место : да/нет ==
    (@следующий (нил свободного места) ≠ нил свободного места)
программа нет свободного места : да/нет ==
    (@следующий (нил свободного места) = нил свободного места)

программа элемент ⟨вх : e⟩ принадлежит множеству ⟨вх : n⟩ : да/нет
. дано     :
. получить:
. --------
. искать ⟨вх : e, n, вых : ответ : да/нет, указатель⟩
конец программы

программа добавить элемент ⟨вх : e : E⟩ в множество ⟨вх : n : номер⟩
. дано     : есть свободное место
. получить:
. --------
. искать ⟨вх : e, n, вых : нашли : да/нет, указатель⟩
. если не нашли то
. .  захватить место ⟨вых : И⟩        | из свободного места
. .  связать ⟨И, @ следующий (n)⟩     | включить И в начало
. .  связать ⟨n, И⟩                   | траектории n-го множества
. .  информация (И) := e              | разместить информацию
. конец если
конец программы

программа удалить элемент ⟨вх : e : E⟩ из множества ⟨вх : n : номер⟩
. дано     :
. получить:
. --------
. искать ⟨вх : e, n, вых : нашли : да/нет, указатель⟩
. если нашли то
. .  И := указатель.@ за                          | удаление эл-та с
. .  связать ⟨указатель.@ до, @ следующий (И)⟩    | инд. И из мн-ва
. .  освободить место ⟨вх : И⟩ | и включение в свободное место
. конец если
конец программы
```

```
программа взять какой нибудь элемент множества ⟨вх : п⟩ в ⟨вых : е⟩
. дано      : множество ⟨вх : п⟩ не пусто
. получить:
. --------
.  И := @ следующий (п)
.  е := информация (И)
.  связать ⟨п, @ следующий (И)⟩
.  освободить место ⟨вх : И⟩
конец программы

программа искать ⟨вх : е, п, вых : нашли, указатель⟩
. дано      : е : Е, п : номер
. получить: нашли : да/нет, указатель : запись(@до, @за : индекс)
. --------
. указатель := (п, @ следующий (п))
. цикл пока указатель.@ за ≠ п и информация (указатель.@ за) ≠ е
. . выполнять
. .  указатель.@ до := указатель.@ за
. .  указатель.@ за := @ следующий (указатель.@ за)
. конец цикла
. нашли := (указатель.@ за ≠ п)
конец программы

программа связать ⟨вх : И1, И2 : индекс⟩ ==
                                  @ следующий (И1) := И2

программа захватить место ⟨вых : И : индекс⟩
. дано      : есть свободное место
. получить:
. --------
.  И := @ следующий (нил свободного места)
.  связать ⟨нил свободного места, @ следующий (И)⟩
конец программы

программа освободить место ⟨вх : И : индекс⟩
. дано      :
. получить:
. --------
.  связать ⟨И, @ следующий (нил свободного места)⟩
.  связать ⟨нил свободного места, И⟩
конец программы

программа кончить работу ==
        информация.кончить работу; @ следующий.кончить работу

конец исполнителя I
```

Замечание. В этом разделе мы занимались только реализацией множества. На практике чаще встречаются нагруженные множества. Приведенные выше реализации легко модифицируются и для реализации нагруженных множеств — достаточно параллельно с вектором «информация» завести вектор «нагрузка» и хранить в элементах этого вектора нагрузку элементов множества.

ЗАДАЧИ И УПРАЖНЕНИЯ

1. Модифицируйте непрерывную реализацию множества (последовательный поиск) так, чтобы при выполнении подряд нескольких действий с одним и тем же элементом (например, вопрос о принадлежности, а потом добавление) поиск элемента производился только один раз.

Указание. Введите глобальные объекты, в которых будет храниться информация о последнем обрабатывавшемся элементе, о его принадлежности множеству и в случае, если элемент принадлежит множеству, его индекс в векторе.

2. Решите задачу 1 для бинарного поиска.

3. Реализуйте множество M элементов типа E на базе

а) хеш-функции $h: E \to 0 \,..\, p-1$,

б) вектора «мощность слоя» элементов типа Z+ с индексом типа $0..p-1$,

в) вектора «инф» элементов типа E с индексом типа $0..p-1$,

г) множества «остаток» элементов типа E

так, чтобы были выполнены следующие соотношения:

1) мощность слоя (i) = число элементов в $M_i = \{m \in M : h(m) = i\}$

2) если $M_i \neq \varnothing$, то
инф (i) $\in M_i$,
$M_i \setminus$ инф (i) $\in$ остаток,
инф (i) $\notin$ остаток

(другими словами, если M_i содержит один элемент, то он помещается в инф(i); если M_i содержит более одного элемента, то один из них помещается в инф(i), а остальные — в остаток).

4. Реализуйте нагруженное множество элементов типа E с нагрузкой типа Y, модифицировав непрерывную реализацию множества.

5. Реализуйте нагруженное множество элементов типа 1..максинд с нагрузкой типа Y, модифицировав битовую реализацию множества.

6. Реализуйте нагруженное множество элементов типа E с нагрузкой типа Y, модифицировав хеш-реализацию множества.

15. Двумерное хеширование по равномерной сетке. Оптимизация алгоритма построения изображения полиэдра

Вспомним проект «Построение изображения полиэдра» (разд. 11). В этом проекте при построении изображения видимой части полиэдра для каждого ребра перебирались все грани, чтобы учесть возможное загораживание гранью части ребра. Введенный в конце разд. 11 фильтр лишь ускорил обработку пары (ребро, грань) в случае, когда грань заведомо не затеняет ребро, однако не изменил качественной картины — перебор всех пар (ребро, грань) при построении изображения полиэдра сохранился.

Легко проследить аналогию между перебором всех граней для определения видимой части ребра в разд. 11 и перебором всех элементов множества для определения принадлежности этому множеству некоторого элемента. Естественно, возникает вопрос: нельзя ли воспользоваться идеями хеширования и *не искать* грани, «имеющие отношение» к ребру, а *вычислять* их? Другими словами, нельзя ли реализовать построение изображения полиэдра так, чтобы пары (ребро, грань) для далеких друг от друга ребра и грани в процессе построения изображения *вообще не рассматривались*? В этом разделе мы покажем, что это возможно, и соответствующим образом изменим проект «Построение изображения полиэдра».

Ситуация, когда нам приходится дорабатывать, улучшать уже готовую программу, типична для программирования. Данный раздел призван также на конкретном примере продемонстрировать, что для проектов, реализованных по технологии «сверху вниз», модификация проводится достаточно легко и просто: хотя новые идеи и требуют добавления в проект новых исполнителей, реализация старых исполнителей меняется незначительно или не меняется вовсе.

Идея двумерного хеширования. Рассмотрим плоскость проекции с координатами u, v. Опишем вокруг проекции полиэдра на эту плоскость *стандартный прямоугольник* (т. е. прямоугольник со сторонами, параллельными координатным осям). Разделим его средними линиями на четыре части (рис. 15.1) и назовем эти части *гнездами*.

Для каждого из четырех гнезд запомним множество граней, проекции которых задевают гнездо. Допуская вольность речи, такое множество граней мы также будем называть *гнездом* и говорить, что грань «попала» в гнездо. Предварительное размещение граней по гнездам назовем *гнездованием граней* (рис. 15.2).

После гнездования граней для построения изображения любого ребра достаточно учесть грани только из тех гнезд, которые задевает проекция этого ребра. Разумеется, грань может попасть не

в одно, а в два, три или даже четыре гнезда. Однако если число граней и ребер велико, в каждое гнездо попадает около четверти всех граней, а ребро, как правило, проецируется целиком в одно гнездо, то общее число анализируемых пар (ребро, грань) уменьшится примерно в четыре раза.

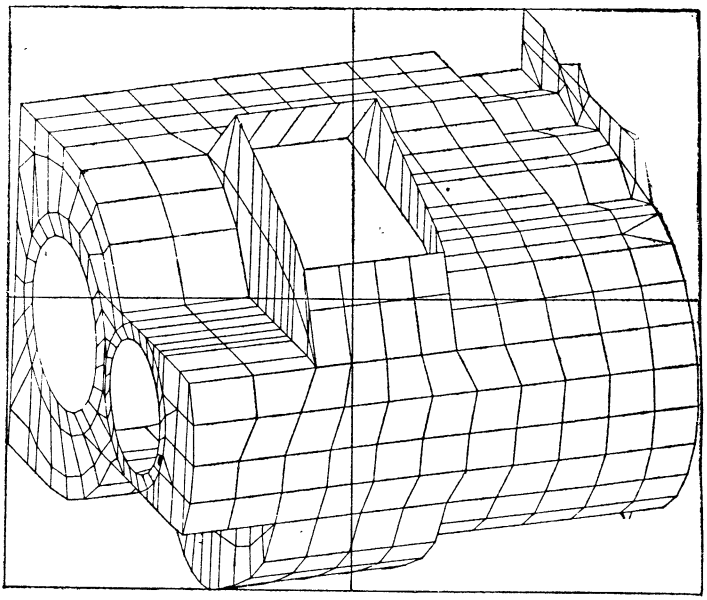


Рис. 15.1

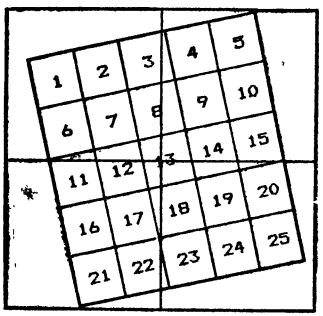
Это и есть выигрыш, который дает хеширование, — ведь если ребро проецируется целиком в одно гнездо, то грани из трех других гнезд *вообще не рассматриваются при построении изображения видимой части ребра.*

Естественно пытаться развить этот подход, увеличивая число делений прямоугольника, описанного вокруг полиэдра, вплоть до теоретически идеального случая, когда размер гнезд примерно равен размеру граней. Еще более изощренную оптимизацию можно было бы провести, рассматривая на плоскости сетку гнезд переменного размера, согласованного с размерами проекций граней. Мы, однако, не будем обсуждать этот подход, а ограничимся только случаем равномерной сетки. Более того, для простоты мы даже не будем подстраивать размер гнезда под размер грани, а реализуем двумерное хеширование по фиксированной сетке, например 30 * 30.

Таким образом, сетка — это матрица из k*k гнезд, где гнездо — это множество элементов типа «грань». Хеш-функция сопоставляет некоторому прямоугольнику множество задеваемых им гнезд, т. е.

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

Полиэдр с 25 гранями



Проекция полиэдра и сетка 2*2

1, 2, 3, 6, 7, 8, 11, 12, 13	3, 4, 5, 8, 9, 10, 13, 14, 15
11, 12, 13, 16, 17, 18, 21, 22, 23	13, 14, 15, 18, 19, 20 22,23,24,25

Результат гнездования

Рис. 15.2

какое-то множество индексов матрицы. В соответствии с этой хеш-функцией каждую грань надо поместить в те гнезда, которые задевает прямоугольник грани, а при построении изображения ребра перебирать грани только из тех гнезд, которые задевает прямоугольник ребра.

Чтобы еще раз пояснить аналогию с обычным хешированием, вспомним, что важнейшая операция при работе с множествами — это поиск. Но искать можно не только конкретный элемент. Задачу поиска в множестве M заданного элемента e можно обобщить так: найти в M подмножество MQ всех элементов $m \in M$, обладающих свойством Q(m). В задаче поиска элемента свойство Q есть совпадение с e, т. е. $Q(m) = (m = e)$. В случае рассматриваемого нами двумерного хеширования M — это множество граней

полиэдра, e — ребро, а ищется подмножество граней MQ, обладающих свойством

Q(m) = (проекции m и e на плоскость uv пересекаются).

И в обычном, и в двумерном хешировании множество MQ явно вычислить трудно. Поэтому свойство Q заменяется более слабым свойством Q′ (т. е. таким, что из Q′(m) следует Q(m)), вычислить которое более или менее просто. В обычном хешировании свойство m = e заменяется на h(m) = h(e). В двумерном хешировании свойство пересечения проекций m и e заменяется на свойство «существует гнездо, которое пересекают и прямоугольник ребра e, и прямоугольник грани m». В том и другом случае выигрыш достигается за счет того, что поиск элементов, удовлетворяющих Q в множестве M, заменяется поиском в подмножестве MQ′.

Простота вычисления MQ′ в обоих случаях обеспечивается тем, что множество M разбивается некоторым *не зависящим от e* способом на подмножества M_i ($M = \bigcup M_i$). В обычном хешировании подмножества M_i не пересекаются, а MQ′ совпадает с одним из них, т. е. задача нахождения MQ′ сводится к выбору одного из уже существующих M_i. В двумерном хешировании множества M_i могут пересекаться, MQ′ есть объединение некоторого числа M_i и нахождение MQ′ состоит в выборе нескольких из M_i.

Возвращаясь к исходной задаче, заметим, что для двумерного хеширования надо знать размеры проекции всего полиэдра. Поэтому будем считать, что в систему предписаний базового исполнителя «Полиэдр» входит еще одно предписание

4. размеры ⟨**вых** : центр : точка R3, радиус : **число**⟩, которое возвращает в выходных параметрах центр и радиус какой-нибудь сферы, содержащей полиэдр и примерно равной ему по размеру.

Реализация. В соответствии с технологией «сверху вниз» введем для всех действий с гнездами (гнездование граней, перебор граней, имеющих отношение к ребру, и т. п.) специальный исполнитель «Гнездователь». Кроме того, будем, если это понадобится, использовать исполнителей «Изобразитель ребра», «UV» и др. из разд. 11. Тогда в соответствии с идеей двумерного хеширования построение изображения полиэдра можно записать так:

программа построение изображения полиэдра
дано : | полиэдр в R3 (исполнитель "Полиэдр")
| направление проектирования (исполнитель "Проектор")
получить: | плоское изображение ребер полиэдра (параллельная
| проекция) с "удалением невидимых линий"

Гнездователь.начать работу (UV.прямоугольник полиэдра)

```
. цикл ∀ грань ∈ Полиэдр.множество граней выполнять
. . Гнездователь.добавить объект ⟨вх: грань,
. .        вх: UV.прямоугольник грани ⟨вх: грань⟩⟩
. конец цикла
. цикл ∀ ребро ∈ Полиэдр.множество ребер выполнять
. . Изобразитель ребра.начать работу ⟨вх: ребро⟩
. . Гнездователь.для каждого объекта имеющего отношение
. .       к ⟨вх: UV.прямоугольник ребра ⟨вх: ребро⟩⟩ выполнить
. .                  ⟨вх: Изобразитель ребра.учесть тень от грани⟩
. . Изобразитель ребра.кончить работу
. конец цикла
. Гнездователь.кончить работу
конец программы
```

Предписание «прямоугольник полиэдра», которое мы здесь добавили исполнителю «UV», должно вырабатывать в качестве значения прямоугольник на плоскости проекции в uv-координатах, который содержит проекцию всего полиэдра и примерно равен ей по размеру. Реализацию этого предписания проведите самостоятельно.

Здесь нам впервые встречается ситуация, когда имя программы передается в качестве фактического параметра другой программе, а именно имя программы «Изобразитель ребра.учесть тень от грани» передается одной из программ Гнездователя. Такая передача, в сущности, ничем не отличается от передачи чисел — при реализации программы

```
для каждого объекта имеющего отношение к ⟨вх: П⟩
                                выполнить ⟨вх: действие
```

формальный параметр «действие» обозначает некоторую программу. Встретив при выполнении строку

```
действие ⟨вх: e⟩,
```

Универсальный Выполнитель заменит формальный параметр «действие» на фактический, указанный в вызове, и выполнит

```
Изобразитель ребра.учесть тень от грани ⟨вх: e⟩.
```

С чисто синтаксической, языковой точки зрения параметр «действие» здесь имеет тип «имя программы» (точнее, сконструирован способом «имя программы» — см. приложение 1).

Теперь перед нами стоит задача реализовать исполнителя «Гнездователь»:

```
исполнитель Гнездователь
. СП:
. 1. начать работу ⟨вх: П: прямоугольник⟩
. 2. добавить объект ⟨вх: e: E, П: прямоугольник⟩
```

```
. 3. для каждого объекта имеющего отношение к
.                    ⟨вх: П: прямоугольник⟩ выполнить ⟨вх: действие⟩
. 4. кончить работу
.
```

В предписании «начать работу» задается максимальный прямоугольник, тот самый, который дальше разбивается равномерной сеткой на гнезда. В соответствии с идеями двумерного хеширования по предписанию «добавить объект» надо перейти от прямоугольника объекта к множеству задеваемых этим прямоугольником гнезд и далее добавить объект в каждое отдельное гнездо из этого множества. Аналогично к операциям над отдельными гнездами сводится и выполнение предписания «для каждого...». Поэтому в соответствии с технологией «сверху вниз» примем решение, что все операции над отдельными гнездами будет выполнять специальный исполнитель «Гнезда» и реализуем Гнездователя на базе этого вновь придуманного исполнителя.

```
. константы:
. k = 30 | число делений сетки гнезд
.
. типы:
. индекс гнезда  = 0 .. k − 1
. прямоугольник = запись (umin, vmin, umax, vmax: R)
.
. объекты:
. u0, v0: число | координаты левого нижнего угла сетки гнезд
. du, dv: число | размеры гнезда по горизонтали и вертикали
```

Эти объекты, вместе с константой k полностью задают сетку гнезд и должны быть установлены по предписанию «начать работу».

```
. используемые исполнители:
. Гнезда
.
конец описаний | ------------------------------

программа начать работу ⟨вх: П: прямоугольник⟩
. дано     : П: прямоугольник | всей сетки гнезд целиком
. получить: | установлены глобальные объекты Гнездователя
. --------
. u0 := П.umin
. v0 := П.vmin
. du := (П.umax − u0)/k
. dv := (П.vmax − v0)/k
. Гнезда.начать работу
конец программы
```

```
программа добавить объект ⟨вх: e: E, П: прямоугольник⟩
. дано      : | объект e и его прямоугольник П
. получить: | объект добавлен во все гнезда, которые задевает П
. --------
.  imin := минимум (k — 1, целое ( (П.umin — u0)/du) )
.  jmin := минимум (k — 1, целое ( (П.vmin — v0)/dv) )
.  imax := минимум (k — 1, целое ( (П.umax — u0)/du) )
.  jmax := минимум (k — 1, целое ( (П.vmax — v0)/dv) )
.
.  цикл ∀ i ∈ imin .. imax выполнять
.  .  цикл ∀ j ∈ jmin .. jmax выполнять
.  .  .  Гнезда добавить объект ⟨вх: e⟩ в гнездо ⟨вх: i, j⟩
.  .  конец цикла
.  конец цикла
конец программы
```

Первые четыре строки этой программы, вычисляющие imin, imax, jmin, jmax, — это и есть хеш-функция, которая по прямоугольнику объекта вычисляет «прямоугольное» множество гнезд (т. е. множество пар (i, j), $imin \leqslant i \leqslant imax$, $jmin \leqslant j \leqslant jmax$) такое, что прямоугольник объекта этим множеством покрывается целиком.

```
программа для каждого объекта имеющего отношение к
.                          ⟨вх: П: прямоугольник⟩ выполнить ⟨вх: действие⟩
. дано      : действие: имя программы | и прямоугольник П
. получить: | действие выполнено для всех гнезд, задеваемых П
. объекты:
.  imin, imax, jmin, jmax: индекс гнезда
. --------
.  imin := минимум (k — 1, целое ( (П.umin — u0)/du) )
.  jmin := минимум (k — 1, целое ( (П.vmin — v0)/dv) )
.  imax := минимум (k — 1, целое ( (П.umax — u0)/du) )
.  jmax := минимум (k — 1, целое ( (П.vmax — v0)/dv) )
.
.  цикл ∀ i ∈ imin .. imax выполнять
.  .  цикл ∀ j ∈ jmin .. jmax выполнять
.  .  .  Гнезда.для каждого объекта из гнезда ⟨вх: i, j⟩
.  .  .                                  выполнить ⟨вх: действие⟩
.  .  конец цикла
.  конец цикла
конец программы

программа кончить работу == Гнезда.кончить работу
конец исполнителя | ==========================
```

Итак, мы совершили шаг декомпозиции, получили реализацию R(Гнездователь, Гнезда), а заодно придумали и самого исполнителя «Гнезда». Поскольку, однако, исполнитель «Гнезда» не является базовым, то надо совершить следующий шаг декомпозиции и реализовать этого исполнителя на базе чего-то еще.

Мы могли бы сказать сейчас, что исполнитель «Гнезда» — это исполнитель типа матрица элементов типа гнездо с индексами типа 0..29, 0..29, где тип гнездо — это последовательность элементов типа E. При этом, правда, надо было бы изменить форму обращений к исполнителю «Гнезда», т. е. вместо

Гнезда.добавить объект ⟨**вх**: e⟩ в гнездо ⟨**вх**: i, j⟩

написать

Гнезда (i, j).добавить ⟨**вх**: e⟩ в конец последовательности,

а вместо

Гнезда.для каждого объекта из гнезда ⟨**вх**: i, j⟩
выполнить ⟨**вх**: действие⟩

написать

Гнезда (i, j).встать в начало последовательности
цикл ∀ e ∈ **непрочитанной части** Гнезда (i, j) **выполнить**
. действие ⟨**вх**: e⟩
конец цикла

(В последнем случае вызов предписания исполнителя «Гнезда» оказался замененным на встроенный в Универсальный Выполнитель цикл, перебирающий все элементы из непрочитанной части последовательности.)

Мы, однако, обещали, что такими «структурами структур», как матрица последовательностей, в этой книге пользоваться не будем. Поэтому в соответствии с традициями этой главы реализуем исполнителя «Гнезда» на базе вектора, почти дословно повторив ссылочную реализацию исполнителя «р множеств» из предыдущего раздела.

исполнитель Гнезда
. **СП:**
. 1. начать работу
. 2. добавить объект ⟨**вх**: e: E⟩ в гнездо ⟨**вх**: i, j: индекс гнезда⟩
. 3. для каждого объекта из гнезда ⟨**вх**: i, j: индекс гнезда⟩
. выполнить ⟨**вх**: действие: **имя программы**⟩
. — связать ⟨**вх**: И1, И2: индекс⟩
. 4. кончить работу
.
. **константы**
. k = 30 | **число делений сетки гнезд**

```
    р                    = k * k | общее число гнезд в сетке
    макс число объектов = 1000
    максинд              = р + макс число объектов

  типы:
   индекс гнезда = 0 .. k — 1
   индекс = 0 .. максинд
   номер  = 0 .. р — 1

  используемые исполнители:
   информация: вектор Е (р + 1 .. максинд)
   @ следующий : вектор индекс (индекс)
  идеи реализации:
  Каждое состояние вектора "@ следующий" определяет отображение f множества индексов в себя.

                f: и —> @ следующий (и)

  Мы реализуем Гнезда так, что в каждый момент времени отображение f взаимно однозначно и имеет ровно р + 1 траектории. Множество с индексами i, j определяется траекторией, проходящей через точку n = i * k + j (нил этого множества). Каждой точке t этой траектории (кроме точки n) отвечает элемент множества, лежащий в информация(t).

конец описаний |----------------------------------

программа начать работу
 дано     : максинд > р
 получить: | траектория каждого множества состоит из точки нил
           | траектория "свободное место" = р .. максинд
 --------
  информация.начать работу
  @ следующий.начать работу
  цикл ∀ n ∈ номер выполнять
  .  связать ⟨n,n⟩ | траектории множеств
  конец цикла
  цикл ∀ И ∈ р .. максинд — 1 выполнять
  .  связать ⟨И,И + 1⟩
  конец цикла
  связать ⟨максинд,р⟩
конец программы

программа добавить объект ⟨вх: е: Е⟩ в гнездо ⟨вх: i,j⟩
 дано  : i,j: индекс гнезда,
```

```
.          @следующий (p)≠ p | т. е. есть свободное место
. получить:
. --------
. n := i*k + j                | нил соответствующего множества
. И := @следующий (p)         | захватить место из траектории
. связать ⟨p, @следующий (И)⟩ |          "свободное место".
. связать ⟨И, @следующий (n)⟩ | включить И в начало
. связать ⟨n, И⟩              | траектории n-го множества
. информация (И) :=e          | добавить объект
конец программы
```

```
программа для каждого объекта из гнезда ⟨вх: i, j: индекс гнезда⟩
.                          выполнить ⟨вх: действие: имя программы⟩
. дано      :
. получить:
. --------
. n := i*k + j                | нил соответствующего множества
. И:= @следующий (n)
. цикл пока И≠n выполнять     | для каждого эл-та мн-ва
. .  действие ⟨вх: информация (И)⟩
. .  И := @следующий (И)
. конец цикла
конец программы
```

```
программа связать ⟨вх: И1, И2: индекс⟩ == @следующий (И1) := И2
программа кончить работу ==
        информация.кончить работу; @следующий.кончить работу
конец исполнителя ==========================
```

Замечание. В случае, если возникает необходимость выполнить какое-то действие для каждого из объектов, хранящихся в некотором исполнителе X, можно поступить двояко. Один путь — добавить исполнителю X предписания «начать обработку», «есть необработанные объекты», «получить очередной объект в ⟨вых: e⟩», «кончить обработку» и написать фрагмент программы вида

```
X.начать обработку
цикл пока X.есть необработанные объекты выполнять
. X.получить очередной объект в ⟨вых: e⟩
. действие ⟨вх: e⟩
конец цикла
X.кончить обработку
```

Другой путь, который и был применен выше, — «поручить» перебор объектов самому исполнителю X, т. е. ввести предписание

X.для каждого объекта выполнять ⟨вх: действие⟩.

При первом подходе в цикле обработки кроме выполнения действия происходят как минимум два вызова предписаний исполнителя X. Если исполнитель X реализован на базе исполнителя Y, который в свою очередь реализован на базе Z_i и перебираемые объекты реально хранятся в испол-

нителе Z, то при получении очередного объекта в цикле обработки эти исполнители будут «по цепочке» вызывать друг друга и передавать полученный объект е обратно. При втором подходе «по цепочке» будет передан только запрос на перебор объектов, а сам перебор будет осуществляться исполнителем Z. Таким образом, второй подход (в программировании он называется *запроцедуриванием циклов*) позволяет писать не только более короткие и более понятные, но и более эффективные программы.

Оценки эффективности двумерного хеширования. Мы реализовали построение изображения полиэдра на основе идей двумерного хеширования по равномерной сетке. Правомерно, однако, задать вопрос: а что дает такая реализация? Интуитивно кажется, что число учитываемых для каждого ребра граней должно существенно сократиться. Однако, что значит существенно? Можно ли, например, гарантировать, что в среднем для каждого ребра будет перебираться не более некоторой доли от общего числа граней? Можно ли вообще хоть как-то более или менее строго оценить эффективность полученной реализации?

Получение подобного рода оценок является не программистской, а математической деятельностью. Пригодность или непригодность новой программы в программировании обычно определяется по результатам работы программы на конкретных данных. Если, однако, новая программа оказалась неудовлетворительной и надо разобраться, почему она работает медленно, или если надо понять, на каких классах данных новая программа эффективна, то получение строгих математических оценок может оказаться полезным.

Мы сформулируем сейчас оценки эффективности двумерного хеширования по равномерной сетке и покажем, как их можно доказать. Основную ценность при этом представляют не столько сами оценки, сколько демонстрация применения математических методов рассуждения к чисто практической программистской задаче.

Итак, пусть полиэдр имеет N ребер и M граней. Нас интересует оценка времени t_p построения изображения полиэдра для больших N и M при заданном векторе проектирования p. Прежде всего пренебрежем затратами на предварительное гнездование граней — они линейно зависят от числа граней. Тогда

$$t_p = \gamma \sum_{i=1}^{N} \hat{m}_i,$$

где $\hat{m}_i$ — число граней, имеющих отношение к i-му ребру (грань *имеет отношение к ребру*, если существует гнездо, которое задевают и грань, и ребро); γ — константа, характеризующая время обработки одной пары (ребро, грань).

Максимальное значение каждого числа $\hat{m}_i$ равно общему числу M граней полиэдра. Можно придумать полиэдры, для которых

хеширование ничего не дает, — все $\hat{m}_i$ действительно равны M и $t_p = \gamma MN$. Поэтому применять хеширование (и получать какие бы то ни было оценки) надо лишь при некоторых органичениях, предположениях о полиэдре.

Мы будем предполагать, что грани полиэдра *имеют примерно одинаковый размер* и *не очень вытянуты*, а *размер гнезд близок к размеру граней*. Более формально это предположение можно сформулировать так: существуют константы $d > 0$, $C \geqslant 1$ такие, что для любой грани Г существуют круги $B_{d/C}$ и B_{dC} в плоскости грани радиусов d/C и dC соответственно такие, что

$$B_{d/C} \subset \Gamma \subset B_{dC}. \tag{1}$$

При тех же константах d и C для любого гнезда G существуют круги $B_{d/C}$ и B_{dC} такие, что

$$B_{d/C} \subset G \subset B_{dC}.$$

Кроме того, введем еще одну характеристику полиэдра (назовем ее кратностью поверхности):

$$K = \max_L K_L,$$

где L — всевозможные прямые, а K_L — число трансверсальных пересечений (т. е. пересечений без касания) прямой L с гранями полиэдра. (На практике K обычно находится в диапазоне от 2 до 6.)

Тогда время построения изображения полиэдра при векторе проектирования p (без учета линейных по числу граней затрат на предварительное гнездование граней) можно оценить как

$$t_p = \gamma \sum_{i=1}^{N} \hat{m}_i \leqslant \sum_{i=1}^{N} m_i, \tag{2}$$

где m_i — число граней, задевающих цилиндр радиуса 3dC с осью, параллельной p, выбранный так, что i-е ребро содержится в цилиндре радиуса dC с той же осью (это возможно, так как всякое ребро принадлежит некоторой грани, а для грани выполнено условие (1)).

Здесь и далее $a \leqslant b$ (меньше с точностью до константы) означает, что существует константа $q > 0$, не зависящая от полиэдра (т. е. от N, M, d, K), такая, что $a \leqslant qb$.

Формула (2) очевидна, но ничего не говорит об эффективности двумерного хеширования. Из нее, однако, уже чисто формально можно получить следующие оценки:

А) в среднем по всем направлениям проектирования при $M > 1$

$$t_{cp} \leqslant KN \ln M;$$

B) при любом направлении проектирования p

$$t_p \leqslant KN\,(D_p/d + 1),$$

где D_p — диаметр проекции модели на прямую, параллельную p.

В частности, если полиэдр не очень вытянут вдоль p, т. е. $D_p/d \leqslant \sqrt{N}$, то $t_p \leqslant KN\sqrt{N}$.

Эти оценки являются оценками сверху. Кроме того, можно привести примеры полиэдров, показывающие, что оценки A) и B) не улучшаемы, а именно:

C) клетчатый лист $n * n$, вектор p ортогонален плоскости листа, $K = 1$:

$$t_p \approx N;$$

D) клетчатый лист $n * n$, вектор p параллелен плоскости листа и одному из направлений сетки листа, $K = 1$:

$$t_p \approx N\sqrt{M};$$

E) многогранник из n^2 граней, удовлетворяющих (1), вписанный в сферу радиуса nd, p — любое, $K = 2$, $n > 5C$:

$$t_p \approx N \ln M$$

($a \approx b$ означает, что $a \leqslant b$ и $b \leqslant a$).

Доказательства оценок A), B) и E) базируются на том факте, что

$$S_{\substack{\text{поверхности}\\ \text{фигуры } \Phi}} \leqslant K \cdot S_{\substack{\text{поверхности любой фигуры,}\\ \text{содержащей } \Phi}} \qquad (3)$$

где K — кратность поверхности фигуры Ф, определяемая аналогично кратности поверхности полиэдра.

Сам этот факт очевидно следует из известной формулы интегральной геометрии (см., например, Сантало Л. Интегральная геометрия и геометрические вероятности, гл. 14, раздел 5), выражающей площадь S поверхности компактного двумерного дифференцируемого кусочно-гладкого многообразия M^2 через число K_L трансверсальных пересечений всевозможных прямых L с многообразием M^2:

$$S = \frac{1}{\pi} \int\limits_{L \cap M^2 \neq \varnothing} K_L \, dL.$$

Из A) и B) докажем как более простую оценку B). Согласно (2),

$$t_p \leqslant \sum_{i=1}^{N} m_i \leqslant N \max_i m_i.$$

Из (1) следует, что все грани j, дающие вклад в m_i, т. е. задевающие цилиндр i-го ребра вдоль p радиуса 3dC, содержатся в бесконечном цилиндре радиуса R = 4dC с той же осью, что и цилиндр i-го ребра. Так как диаметр проекции полиэдра на ось цилиндра равен D_p, то можно выделить часть цилиндра, скажем B, длины D_p такую, что все грани j из m_i лежат в этой части. Площадь поверхности этой части цилиндра

$$S_B = 2\pi R^2 + 2\pi R D_p = 2\pi R (R + D_p).$$

На основании (3) суммарная площадь S граней j из m_i удовлетворяет неравенству

$$S \leqslant K \cdot S_B = 2\pi KR (D_p + R) = 2\pi K4\ dC (D_p + 4dC).$$

С другой стороны, из (1) имеем

$$S \geqslant \pi (d/C)^2 m_i.$$

Следовательно,

$$m_i \leqslant (C/d)^2 2K4\ dC (D_p + 4dC) = 8C^2CK (D_p/d + 4C) \leqslant K (D_p/d + 1).$$

Значит,

$$t_p \leqslant N \max_i m_i \leqslant KN (D_p/d + 1).$$

Доказательства оценок A) и E), хотя и более сложны и требуют ряда дополнительных построений, проводятся в принципе таким же образом на основе формулы (3). Доказательства оценок C) и D) тривиальны и оставляются читателю.

Приведенные выше оценки показывают, что двумерное хеширование по равномерной сетке наиболее эффективно на участках поверхности полиэдра, нормальных к вектору проектирования, — на этих участках $t_p \approx KN$. На участках, почти касательных к вектору проектирования, $t_p \approx KN\sqrt{M}$ (если только поверхность не является сильно вытянутой вдоль p), а в целом для шарообразной модели (например, для аппроксимации сферы), равно как и в среднем по всем направлениям проектирования для любой модели, $t \approx KN \ln M$.

ЗАДАЧИ И УПРАЖНЕНИЯ

1. Реализуйте предписание «UV.прямоугольник полиэдра».

2. Модифицируйте построение изображения полиэдра следующим образом: введите три дополнительные сетки гнезд, такие же как и первая, но со сдвигом на половину размера гнезда по горизонтали и/или по вертикали, с тем чтобы «накрыть» гнездами

дополнительных сеток вертикальные и горизонтальные линии первой сетки. При гнездовании грани ее следует добавлять во все задеваемые гнезда всех сеток, а при обработке ребра следует выбрать сетку, в которой ребро задевает наименьшее количество гнезд, и работать так, как если бы других сеток не существовало. (Для полиэдров с гранями примерно одинакового размера такая модификация обеспечивает практически 100 %-ное попадание изображаемого ребра целиком в одно гнездо, что приводит к ускорению построения изображения полиэдра в 1.5—2 раза.)

3. Докажите оценки C) и D) эффективности двумерного хеширования для клетчатого листа из n * n граней.

4. Пусть M1 — множество отрезков в R3 и M2 — множество треугольников в R3. Реализуйте программу построения изображения отрезков из M1 с учетом загораживания их треугольниками из M2.

5. Решите задачу 4 так, чтобы в процессе построения изображения M1 каждый треугольник из M2 рассматривался не более одного раза.

6. Примените двумерное хеширование при решении задачи 5.

В задачах 7—11 требуется придумать максимально эффективную реализацию. Как минимум, программа *не должна* анализировать все пары (a, b), $a \in A$, $b \in B$.

7. Пусть A и B — два множества точек на прямой. Напишите программу, которая находит а) расстояние между множествами, б) пересечение множеств.

8. Пусть A и B — два множества отрезков на прямой. Напишите программу, которая находит а) расстояние между множествами, б) пересечение множеств.

9. Пусть A и B — два множества точек на плоскости. Напишите программу, которая находит их пересечение.

10. Пусть A и B — два множества целых точек в R3, лежащих в шаре радиуса 127. Напишите программу, которая находит их пересечение.

11. Пусть A и B — два множества точек в единичном круге на плоскости. Напишите программу, которая находит расстояние между ними с точностью до 1/10.

16. Виртуальная память

В этом разделе мы реализуем «большой» вектор из N элементов типа E на базе точно такого же «большого» вектора из N элементов типа E и еще одного «маленького» вектора из n элементов типа E, где $n \ll N$. Однако сначала поясним причины возникновения этой на первый взгляд странной задачи.

Память современных ЭВМ состоит из нескольких компонент, физически по разному реализованных и работающих с разными скоростями. Любая ЭВМ имеет внутреннюю, так называемую *оперативную* память, которая является вектором элементов специального типа байт. Байт состоит из 8 элементов типа **да/нет**, называемых *битами*, и может иметь $2^8 = 256$ различных состояний, которые обычно нумеруются от 0 до 255. Таким образом, байт можно считать объектом типа 0..255.

Кроме оперативной, ЭВМ имеет обычно внешнюю, более медленную память, которая является вектором элементов типа *блок*. Блок — это вектор элементов типа байт с индексом, как правило, от 1 до 512. В этом разделе для нас существенно только то, что внешняя память имеет на порядок больший объем и на порядок меньшую скорость работы, чем оперативная.

Кроме оперативной и внешней памяти современные ЭВМ часто имеют еще компоненту памяти, называемую *кэш* (от англ. cache — запас) — также вектор элементов типа байт. Кэш является на порядок более быстрой памятью и имеет на порядок меньший объем, чем оперативная.

Разумеется, если при работе программы необходимо хранить объем информации, превышающий суммарный объем всех видов памяти используемой ЭВМ, то ситуация предельно проста: на данной ЭВМ данную программу выполнить нельзя. Точно так же программу нельзя выполнить, если скорость работы самой быстрой памяти не обеспечивает нужную скорость выполнения программы. Чаще, однако, встречается промежуточная ситуация: суммарный объем памяти достаточен и скорость работы быстрой памяти удовлетворительна, но вся необходимая информация одновременно в быструю память не помещается.

Поскольку в каждый отрезок времени программа обычно работает не со всеми данными, а лишь с некоторой их частью, то можно попытаться разместить эти активно используемые данные в быстрой памяти. Тогда на данном отрезке времени скорость выполнения программы будет определяться скоростью работы быстрой памяти. Разумеется, в процессе выполнения программы активно используемые данные изменяются. Поэтому для обеспечения быстрой работы данные нужно регулярно перераспределять между быстрой памятью и медленной. Возникает вопрос: как это делать? Ответ, который излагается в настоящем разделе, таков: будем писать программы, не используя явно ни маленькую быструю память (вектор элементов типа E с индексом от 0 до n — 1), ни большую медленную. Вместо этого используем в программе исполнителя «Виртуальная (от англ. virtual — воображаемая) память» (вектор элементов типа E с индексом от 0 до N — 1), считая, что его ско-

рость работы близка к скорости работы быстрой памяти. Сам этот большой быстрый вектор реализуем на базе большой медленной памяти, используя для ускорения работы маленькую быструю.

Естественно, что на самом деле обеспечить высокую скорость работы исполнителя «Виртуальная память» можно только в том случае, если перераспределения данных между быстрой и медленной памятью будут происходить существенно реже, чем обращения к элементам, уже имеющимся в быстрой памяти. А это зависит не только от реализации виртуальной памяти, но и от того, как она используется. Если, например, последовательно использовать нулевой, первый,..., N — 1-й элементы виртуальной памяти, то, какова бы ни была ее реализация, скорость работы будет такой же, как скорость медленной памяти. Таким образом, реализация исполнителя «Виртуальная память» должна учитывать характер его использования.

Рассмотрим две постановки и два решения этой задачи:

а) оперативная память рассматривается как большая медленная, а кэш — как ускоритель (тип E=байт). Реализация должна быть предельно простой (она «запаивается» в аппаратуру) и обеспечивать быструю работу как минимум при многократном чтении (в произвольном порядке) n любых подряд идущих элементов;

б) оперативная память (точнее, некоторая ее часть, называемая *пулом* — от англ. pool — общий фонд) рассматривается как быстрая маленькая, т. е. как ускоритель, а основным хранилищем элементов служит внешняя (или, как ее еще называют, вторичная) память. Хотя элементом оперативной памяти является байт, а элементом вторичной — блок, мы для простоты будем считать, что они состоят из элементов одного типа — из блоков. Виртуальная память должна работать с блоками и обеспечивать быструю работу и при чтении, и при записи любых (не обязательно идущих подряд) n элементов.

Кроме основных базовых структур в реализациях разрешается использовать «небольшие» (из n элементов) векторы объектов типа да/нет, чисел и ссылок.

Поскольку изменение элемента виртуальной памяти — потенциально более долгая операция, чем его использование, то доступ к элементу вектора разделим на две операции — чтение и запись — и будем реализовывать исполнителя «Виртуальная память» со следующей системой предписаний:

1. начать работу
2. элемент с индексом ⟨вх : i⟩ :E
3. записать ⟨вх : E⟩ в элемент с индексом ⟨вх : i⟩
4. кончить работу

Замечание. Обычно мы формулируем требования к реализациям до их создания. На практике это не всегда так — довольно часто сначала придумывается интересная реализация, а потом выясняется, что она может существенно помочь при решении тех или иных задач.

Кэш-память. Пусть стоит задача реализовать воображаемый (виртуальный) большой быстрый вектор (назовем его «Кэш память») элементов типа E (E=байт) с индексом от 0 до N — 1 на базе маленького быстрого вектора «Кэш» с индексом от 0 до n — 1 и большого медленного вектора «Память» с индексом от 0 до N — 1. Идея реализации очень проста: будем хранить элементы виртуального вектора в соответствующих элементах вектора «Память», а вектор «Кэш» используем в качестве ускорителя. Для этого отобразим множество индексов 0..N — 1 хеш-функцией h(i) = i **mod** n в множество 0..n — 1. При записи элемента с индексом i будем записывать его и в память, и в элемент кэш с индексом h(i). При чтении элемента сначала посмотрим, не находится ли этот элемент уже в векторе «Кэш». Если да, то к памяти обращаться вообще не будем (за счет чего и должны получить выигрыш в скорости). Если же элемента в кэш нет, то прочтем его из памяти и одновременно поместим в кэш в элемент с индексом h(i).

В ЭВМ устройства «кэш» и «память» конструируются так, что при всяком чтении из памяти или записи в память элемент попадает в кэш автоматически, образно говоря, по «параллельным проводам». Таким образом, работа с кэш не требует никакого дополнительного времени по сравнению с работой с памятью. Если, однако, требуемый элемент уже есть в кэш, то при его чтении обращения к памяти не происходит. За счет этого среднее быстродействие ЭВМ повышается в 1.5—2 раза.

исполнитель Кэш память
. **СП:**
. 1. начать работу
. 2. элемент с индексом ⟨**вх**: i⟩ : E
. 3. записать ⟨**вх**: E⟩ в элемент с индексом ⟨**вх**: i⟩
. 4. кончить работу
.
. **константы:**
. N = 65536
. n = 2048
.
. **типы:**
. индекс = 0 .. N — 1

```
.  кэш_индекс = 0 .. n — 1
. объекты:
.  Индекс в памяти: вектор индекс (кэш_индекс)
. используемые исполнители:
.  Кэш      : вектор E (кэш_индекс)
.  Память : вектор E (индекс)
.
конец описаний |--------------------------------

программа начать работу
. дано     :
. получить:
. --------
. Кэш.начать работу
. Индекс в памяти.начать работу
. Память.начать работу
. цикл ∀ k ∈ кэш_индекс выполнять
. . Кэш (k) := Память (k)
. . Индекс в памяти (k) := k
. конец цикла
конец программы

программа элемент с индексом ⟨вх: i: индекс⟩: E
. дано     :
. получить:
. --------
. k := i mod n
. если Индекс в памяти (k) ≠ i то
. . Кэш (k) := Память (i)
. . Индекс в памяти (k) := i
. конец если
. ответ := Кэш (k)
конец программы

программа записать ⟨вх: e: E⟩ в элемент с индексом ⟨вх: i⟩
. дано     :
. получить:
. --------
. k := i mod n
. Кэш (k) := e
. Индекс в памяти (k) := i
. Память (i) := e
конец программы

программа кончить работу
. дано     :
```

```
| получить:
| ---------
| Кэш.кончить работу
| Индекс в памяти.кончить работу
| Память.кончить работу
конец программы

конец исполнителя |==============================
```

Напомним, что реализация кэш-памяти дает выигрыш, если в каждый конкретный момент времени активно читается небольшое (не превосходящее кэш) множество *рядом расположенных* элементов вектора. В этом случае все они попадают в кэш, и при работе с ними память трогать не приходится. Если, однако, поочередно работать всего лишь с двумя элементами вектора, индексы которых отличаются на n, то придется постоянно обращаться к памяти, и никакого ускорения по сравнению просто с работой с памятью не будет.

Виртуальная память. Решим теперь задачу б). Здесь у нас тип E — это блок, маленький быстрый вектор «Пул» — это некоторая часть оперативной памяти ЭВМ, большой медленный вектор «Вторичная память» — это некоторая область во внешней памяти. Будем считать, что пул является обычным вектором, а исполнитель «Вторичная память», по сути также являющийся вектором, имеет следующую систему предписаний:

1. начать работу
2. прочесть блок ⟨**вх** : i : номер блока⟩ в ⟨**вых** : e : блок⟩
3. записать ⟨**вх** : e : блок⟩ в блок ⟨**вх** : i : номер блока⟩
4. кончить работу

Постановка этой задачи образно изображена на рис. 16.1. Исполнитель «Вторичная память» изображен мальчиком на стремянке. Приведенная выше реализация кэш-памяти в точности повторяет эту картину в миниатюре: вместо блоков — байты, вместо пула — кэш, вместо вторичной памяти — память.

Ниже мы будем использовать знак # (который считается такой же буквой, как A, B, C и др.) в качестве альтернативы для слова «индекс», т. е. имена вида «#блока» будем читать и использовать как «индекс блока».

```
исполнитель Виртуальная память
| СП:
| 1. начать работу
| 2. элемент с индексом ⟨вх : i⟩ : E
| 3. записать ⟨вх : e : E⟩ в элемент с индексом ⟨вх : i⟩
```

— поместить блок ⟨**вх** : i⟩ в пул ⟨**вых** : k⟩
— освободить блок пула ⟨**вых** : k⟩
4. кончить работу

константы:
N = 2048
n = 128
длина блока = 512 | байт

типы:
#блока = 0 .. N — 1 | во вторичной памяти
#блока пула = 0 .. n — 1
E = блок = **вектор** байт (0 .. длина блока — 1)

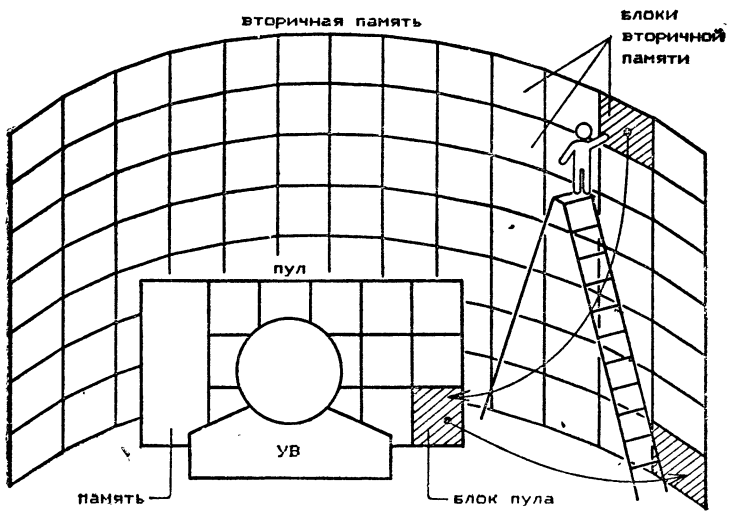


Рис. 16.1

Идеи реализации. Реализуем исполнителя «Виртуальная память» (вектор блоков с индексом 0..N — 1) на базе ряда объектов в памяти и исполнителя «Вторичная память», который является точно таким же вектором блоков, только очень медленным. В процессе работы информация из некоторых блоков вторичной памяти располагается в буферном пуле. Если при обращении к тому или иному блоку виртуальной памяти этот блок уже находится в пуле, то производятся требуемые действия с соответствующим блоком пула. Если же блока в пуле нет, то он в пул помещается. При

этом какой-то другой блок может быть «вытолкнут» из пула обратно во вторичную память (рис. 16.1). В конце работы все блоки пула, «выталкиваются» обратно во вторичную память.

Таким образом, действия, производимые при имитации работы с блоками виртуальной памяти, можно разделить на чтение/запись блоков в пуле и/или во вторичной памяти, и на работу с ограниченным множеством пар (индекс блока в пуле, индекс блока во вторичной памяти) для всех находящихся в пуле блоков. (Число пар в этом множестве ограничено числом блоков пула.) В соответствии с технологией «сверху вниз» введем для работы с этим множеством промежуточного исполнителя «Множество пар» и «переложим на него» анализ наличия или отсутствия блока виртуальной памяти в пуле, определение индекса выталкиваемого блока пула, определение по индексу блока его индекса в пуле и т. п.:

```
. используемые исполнители:
. Множество пар
. Пул: вектор элементов типа E с индексом типа #блока пула
. Вторичная память
.
конец описаний | ---------------------------------
программа начать работу
. дано     :
. получить:
. --------
. Множество пар.начать работу
. Пул.начать работу
. Вторичная память.начать работу
конец программы

программа элемент с индексом (вх: i: #блока): E
. дано     :
. получить:
. --------
. поместить блок (вх: i) в пул (вых: k)
. ответ := Пул (k)
конец программы
```

Введенная здесь локальная программа «поместить блок в пул» должна поместить i-й блок виртуальной памяти в пул (если его там не было раньше) и выдать в выходном параметре k индекс соответствующего блока пула.

```
программа записать (вх: e: E) в элемент с индексом (вх: i)
. дано     : i: # блока
. получить:
```

```
. --------
. поместить блок ⟨вх : i⟩ в пул ⟨вых : k⟩
. Пул (k) := e
конец программы

программа поместить блок ⟨вх : i⟩ в пул ⟨вых : k⟩
. дано      : i : # блока
. получить: k : # блока пула
. --------
. k := Множество пар.индекс блока в пуле ⟨вх : i⟩
. если k = неопр то | блока в пуле нет
. . освободить блок пула ⟨вых : k⟩
. . Вторичная память.прочесть блок ⟨вх : i⟩ в ⟨вых : Пул (k)⟩
. . Множество пар.добавить пару ⟨вх : k, i⟩
. конец если
конец программы
```

Предписание «Множество пар.индекс блока в пуле» должно вырабатывать в качестве значения либо индекс блока пула, в котором расположен i-й блок виртуальной памяти, либо **неопр**, если i-го блока виртуальной памяти в данный момент в пуле нет (т. е. в Множестве пар не существует пары вида (k, i) с данным i).

```
программа освободить блок пула ⟨вых : k⟩
. дано      :
. получить: k : # блока пула
. --------
. Множество пар.вытолкнуть пару ⟨вых : k, i⟩
. если i ≠ неопр то | блок пула был занят
. . Вторичная память.записать ⟨вх : Пул (k)⟩ в блок ⟨вх : i⟩
. конец если
конец программы
```

Мы предполагаем, что предписание «Множество пар.вытолкнуть пару» либо удаляет какую-то пару (k, i) из множества, выдавая величины k и i в качестве выходных параметров наружу, либо выдает пару вида (k, **неопр**), где k — индекс свободного блока пула (множество пар при этом не меняется). Кроме того, будем предполагать (это понадобится в программе «кончить работу»), что при применении n раз подряд предписания «вытолкнуть пару» получаются все n различных значений k.

```
программа кончить работу
. дано      :
. получить:
. --------
. цикл n раз | т. е. число блоков пула раз
```

```
. . выполнять
. .  освободить блок пула ⟨вых: k⟩
. конец цикла
. Вторичная память.кончить работу
. Пул.кончить работу
. Множество пар.кончить работу
конец программы

конец исполнителя |==========================
```

Прежде чем двигаться дальше, т. е. в соответствии с технологией «сверху вниз» реализовывать промежуточного исполнителя «Множество пар», заметим, что в приведенной реализации при выталкивании блока виртуальной памяти из пула он *всегда* записывается во вторичную память. Очевидно, что если содержимое блока не менялось, то в этом нет необходимости. Поскольку запись блока во вторичную память — весьма длительная операция, изменим реализацию так, чтобы блоки пула записывались во вторичную память только тогда, когда это действительно нужно, т. е. когда содержимое блока менялось. Для хранения информации об изменениях в блоках добавим в реализацию исполнителя «Виртуальная память» глобальный объект

были изменения: вектор да/нет (# блока пула)

и изменим реализацию программ «записать», «поместить блок» и «освободить блок»:

```
программа записать ⟨вх: e: E⟩ в элемент с индексом ⟨вх: i⟩
. дано     : i: # блока
. получить:
. --------
. поместить блок ⟨вх: i⟩ в пул ⟨вх: k⟩
. Пул (k) := e
. были изменения (k) := да
конец программы

программа поместить блок ⟨вх: i⟩ в пул ⟨вых: k⟩
. дано     : i: # блока
. получить: k: # блока пула
. --------
. k := Множество пар.индекс блока в пуле ⟨вх: i⟩
. если k = неопр то | блока в пуле нет
. .  освободить блок пула ⟨вых: k⟩
. .  Вторичная память.прочесть блок ⟨вх: i⟩ в ⟨вых: Пул (k)⟩
. .  Множество пар.добавить пару ⟨вх: k, i⟩
. .  были изменения (k) := нет
```

```
. конец если
конец программы

программа освободить блок пула (вых: k)
. дано :
. получить: k: # блока пула
. ---------
. Множество пар.вытолкнуть пару (вых: k, i)
. если i≠неопр и были изменения (k) то
. . Вторичная память.записать (вх: Пул (k)) в блок (вх: i)
. конец если
конец программы
```

Второй шаг декомпозиции. Займемся теперь реализацией исполнителя «Множество пар». Система предписаний этого исполнителя выглядит так:

```
исполнитель Множество пар
. СП:
.  1. начать работу
.  2. вытолкнуть пару (вых: k: # блока пула, i: # блока)
.  3. добавить пару (вх: k: # блока пула, i: # блока)
.  4. индекс блока в пуле (вх: i: # блока) : # блока пула
.  5. кончить работу
.
. константы:
.  N          = 2048
.  n          = 128
.
. типы:
.  # блока      = 0 .. N — 1
.  #блока пула = 0 .. n — 1
.
```

Поскольку число пар (k, i) не превосходит числа блоков в пуле и k соответствуют индексам блоков пула, то простейшее решение состоит в том, чтобы хранить информацию о парах (k, i) в глобальном объекте

индекс блока : **вектор** #блока (#блока пула),

а именно в индекс блока (k) записать i, если пара (k, i) есть в множестве пар, и **неопр**, если пары (k, i) с данным k в множестве нет.

Простейшая стратегия выталкивания, удовлетворяющая требованию «за n выталкиваний — все n различных значений k», состоит в том, чтобы последовательно выталкивать блоки сначала с $k = 0$, затем с $k = 1$, $k = 2$, ..., $k = n - 1$, потом опять с

k = 0, k = 1 и т. д. Заметим, что если при всяком добавлении пары (k, i) k содержит индекс только что вытолкнутого блока (а в реализации Виртуальной памяти это так), то применение этой стратегии приводит к выталкиванию из пула «самого старого» блока — того, который находится в пуле дольше всех. Для реализации этой стратегии достаточно хранить индекс последнего вытолкнутого блока пула и перемещать его по «кольцу» 0..n — 1 при очередном выталкивании. Мы будем хранить этот индекс в глобальном объекте

#вытолкнутого блока : #блока пула.

Таким образом, продолжение реализации исполнителя «Множество пар» выглядит так:

```
. объекты:
. индекс блока : вектор # блока (# блока пула)
. # вытолкнутого блока : # блока пула
конец описаний |----------------------------

программа начать работу == # вытолкнутого блока := n — 1
программа вытолкнуть пару ⟨вых : k : # блока пула, i : # блока⟩
. дано      :
. получить:
. --------
. k := (# вытолкнутого блока + 1) mod n
. i := индекс блока (k)
. индекс блока (k) := неопр
. # вытолкнутого блока := k
конец программы

программа добавить пару ⟨вх : k : # блока пула, i : # блока⟩
. дано      : индекс блока (k) = неопр
. получить:
. --------
. индекс блока (k) := i
конец программы

программа индекс блока в пуле ⟨вх : i : # блока⟩ : # блока пула
. дано      :
. получить: ответ = неопр |если блока i в пуле нет
.           или индекс блока (ответ) = i
. --------
. цикл ∀ k ∈ # блока пула пока индекс блока (k) ≠ i
. . выполнять
. . ничего не делать
```

```
. конец цикла
. ответ := k
конец программы

программа кончить работу == ничего не делать
конец исполнителя |==========================
```

Хеш-реализация виртуальной памяти. Приведенная выше реализация виртуальной памяти (а точнее, исполнителя «Множество пар») обладает одним недостатком — анализ наличия или отсутствия нужного блока в пуле, а также определение его индекса в пуле (предписание «Множество пар.индекс блока в пуле») является массовой операцией, так как осуществляется последовательным перебором всех номеров блоков пула. От этого недостатка можно избавиться, если воспользоваться идеями хеширования, т. е. не искать индекс блока в пуле, а вычислять его по индексу блока в памяти. Введем константу

```
p = n | число хеш-значений
```

и глобальные объекты

```
хеш_таблица: вектор # блока пула (0 .. p — 1)
@след, @пред: вектор # блока пула (# блока пула)
```

Анализ наличия и определение месторасположения блока i в пуле изменим следующим образом: все множество блоков, находящихся в пуле, разобьем на подмножества, соответствующие различным значениям хеш-функции h(i) = i **mod** p. Номера блоков пула в каждом подмножестве провяжем ссылками «@след» и «@пред» в отдельные кольца. Индекс какого-нибудь элемента кольца (т. е. номер блока пула) запишем в хеш_таблица(h), где h — значение хеш-функции для данного кольца. Если для некоторого значения h в пуле нет блоков с таким значением хеш-функции, то элементу хеш-таблицы с индексом h присвоим значение **неопр**:

```
исполнитель Множество пар
. СП:
.   1. начать работу
.   2. вытолкнуть пару ⟨вых: k: # блока пула, i: # блока⟩
.   3. добавить пару ⟨вх: k: # блока пула, i: # блока⟩
.   4. индекс блока в пуле ⟨вх: i: # блока⟩ : # блока пула
.   5. кончить работу
.
. константы:
.   N          = 2048
.   n          = 128
.   p          = n | число хеш-значений
```

```
.
. типы:
. # блока        = 0 .. N — 1
. # блока пула = 0 .. n — 1
.
. объекты:
.  индекс блока : вектор # блока (# блока пула)
.  # вытолкнутого блока : # блока пула
.  хеш_таблица : вектор # блока пула (0 .. p — 1)
.  @ след, @пред : вектор # блока пула (#блока пула)
.
конец описаний |----------------------------
```

```
программа начать работу == # вытолкнутого блока := n — 1
программа вытолкнуть пару (вых : k : # блока пула, i : # блока)
. дано      :
. получить:
. --------
. k := (# вытолкнутого блока + 1) mod n
. i := индекс блока (k)
. индекс блока (k) := неопр
. # вытолкнутого блока := k
```

В новой реализации, однако, в случае, если выталкивается занятый блок (i ≠ неопр), надо исключить его из соответствующего кольца:

```
. если i ≠ неопр то
. .  связать (@ пред (k), @ след (k))
```

а также, если блок в кольце один или если блоков несколько, но в хеш-таблице записан индекс именно этого блока, — модифицировать хеш-таблицу:

```
. .  h := i mod p
. .  выбор
. .  . при @ след (k) = k          ⇒ хеш_таблица (h) := неопр
. .  . при хеш_таблица (h) = k ⇒ хеш_таблица (h) := @след (k)
. .  конец выбора
. конец если
конец программы
```

```
программа добавить пару (вх : k : # блока пула, i : # блока)
. дано      : индекс блока (k) = неопр
. получить:
. --------
.  индекс блока (k) := i
```

Кроме того, новый блок надо вставить в соответствующее кольцо, если оно было (т. е. если хеш_таблица(h) ≠ **неопр**), либо сформировать из блока новое кольцо, если такого кольца не было:

```
. h := i mod p
. k1: = хеш_таблица (h)
. если k1 = неопр
. . то | блоков с хеш_значением h раньше не было
. .  связать ⟨k,k⟩
. .  хеш_таблица ( h) := k
. . иначе | вставляем блок в старое кольцо:
. .  связать ⟨@ пред (k1), k >
. .  связать ⟨k,          k1⟩
```

В этот момент все уже сделано. Мы, однако, ускорим еще многократные обращения к одному и тому же блоку в случае, если блок в кольце не один. Для этого воспользуемся тем, что в хеш-таблице может быть записан индекс любого элемента кольца, а поиск начинается именно с этого элемента. Таким образом, если в хеш-таблицу записать индекс нового, только что прочитанного блока, то информация о блоках в пуле останется корректной, а при повторном обращении к этому блоку искать в кольце не придется — мы сразу получим индекс требуемого блока. Поэтому напишем еще

```
. . хеш_таблица (h) := k
. конец если
конец программы
```

```
программа индекс блока в пуле ⟨вх: i: # блока⟩: # блока пула
. дано     :
. получить: ответ = неопр | если блока i в пуле нет
.           или индекс блока (ответ) = i
. --------
```

В новой реализации для получения ответа достаточно просмотреть только элементы кольца с соответствующим значением хеш-функции. Определим сначала индекс k первого элемента кольца:

```
. h := i mod p
. k := хеш_таблица (h)
```

Теперь надо просмотреть само кольцо в поисках требуемого блока. С учетом того, что кольцо может быть пусто (в этом случае k = **неопр**), поиск блока можно записать так:

```
. если k≠неопр то | есть блоки с таким значением хеш-функции
. .  k1 := k
. .  цикл пока индекс блока (k)≠i и @ след (k)≠k1
. .  .  выполнять
```

```
. . .   k := @след(k)
. . конец цикла
. . если индекс блока(k)≠i то k := неопр конец если
. конец если
.  утв: k = неопр | если i-го блока нет в пуле
.       или индекс блока(k) = i
. ответ := k
конец программы
```

Использованная выше локальная подпрограмма «связать» реализуется, как обычно:

```
программа связать ⟨вх: ka, kb⟩ == @след(ka) := kb;
                                  @пред(kb) := ka
программа кончить работу == ничего не делать
конец исполнителя | ==========================
```

Поскольку число хеш-значений совпадает с максимальным числом блоков в пуле, то можно надеяться, что в кольце, как правило, будет один-два элемента (если, например, работать с n подряд расположенными блоками вторичной памяти, то в каждом кольце будет по одному элементу). Так как число хеш-значений является константой, то его легко увеличить, например положить равным 2n. В последнем случае в каждом кольце, как правило, будет всего один элемент (или не будет элементов вовсе) и поиск элемента будет мгновенным.

Замечание. Слова «виртуальная память», вообще говоря, означают любую реализацию вектора на базе других структур данных. Приведенные выше реализации в этом классе задач не единственные — широко распространенными и практически используемыми являются, например, реализации многих (например, 256) динамических векторов, ограниченных в совокупности на базе одного большого вектора, или реализации «большого разреженного» вектора с индексом от 0 до N — 1 на базе «маленького» вектора с индексом от 0 до n — 1, где $n \ll N$. Слова «разреженный» означают, что, хотя в большом векторе N элементов, в каждый момент времени не более n из них отлично от некоторой константы e0. Как и выше, тип элементов здесь — это обычно байт или блок, а в реализациях кроме «основных» базовых структур используются «небольшие» вспомогательные векторы, элементами которых являются числа или ссылки.

ЗАДАЧИ И УПРАЖНЕНИЯ

Во всех реализациях ниже кроме указанных базовых структур можно использовать также «небольшие» вектора чисел и ссылок, если это понадобится.

1. Реализуйте большой разреженный вектор элементов типа Б на базе маленького вектора элементов типа Е.

2. Реализуйте исполнителя «р динамических векторов блоков» на базе одного вектора блоков.

3. Реализуйте исполнителя «р динамических векторов блоков» на базе быстрого маленького буферного пула блоков и исполнителя «Вторичная память».

4. Модифицируйте исполнителя «Виртуальная память» так, чтобы он работал не с байтами, а с элементами типа Е, где тип Е = вектор байт (1..длина) и величина «длина» делит величину «длина блока».

17. Простейшая файловая система

В этом разделе мы реализуем новую структуру данных — простейшую файловую систему — на базе исполнителя «Диск». Однако прежде чем формально ставить основную задачу программирования, т. е. выписывать системы предписаний искомого исполнителя «Простейшая файловая система» (А) и базового исполнителя «Диск» (Е), кратко поясним, откуда эта задача возникла.

При работе с электронно-вычислительными машинами (ЭВМ), которые выступают у нас в роли Универсальных Выполнителей, возникает необходимость хранения большого количества разнообразной информации, в частности, текстов программ. До сих пор мы могли считать, что Универсальному Выполнителю можно вручить кипу исписанных листов бумаги и далее он автоматически выполнит заданную программу. Эта модель, однако, хороша только в том случае, если Универсальный Выполнитель является человеком. На практике, при работе с ЭВМ тексты программ должны быть закодированы с помощью последовательностей нулей и единиц. Такие последовательности можно запомнить, хранить и при необходимости воспроизвести, например на магнитной ленте подобно тому, как хранятся и воспроизводятся на магнитофонных кассетах последовательности звуков.

Использование магнитофона для хранения информации, однако, не очень удобно: для записи или воспроизведения требуемой информации ленту придется перематывать к нужному месту, что не очень быстро. Иное дело проигрыватель — там иглу можно прямо поставить куда надо! Поэтому при работе с ЭВМ для хранения информации используют специальное устройство, называемое *диском*, которое объединяет идеи магнитной записи/воспроизведения с возможностью быстрого доступа к любому месту пластинки. Диск можно представлять в виде покрытой магнитным слоем вращаю-

щейся пластинки, над которой по радиусу перемещается магнитная головка, подобная магнитофонной.

В отличие от бытовой техники, предназначенной для записи и воспроизведения музыки, речи и других аналоговых сигналов, запись информации на диск проводится фиксированными, строго отмеренными порциями, называемыми *блоками*. Диск обеспечивает запись/чтение любого блока по его номеру и, таким образом, по сути дела является вектором блоков. В зависимости от устройства один диск может содержать от нескольких сотен до нескольких миллионов блоков.

Блок в свою очередь — это вектор элементов типа байт с индексом от 1 до 512, где байт — это элемент, который может принимать 256 различных состояний. Мы можем приписать разным состояниям байта различные символы, например считать, что 65-е состояние означает букву А, 66-е — букву В, 67-е — букву С и т. п., т. е. считать, то байт — это просто символ. Такое отождествление байт с символами и будет использовано в этом разделе. Числа 65, 66, 67 при этом называются *кодами символов* А, В и С соответственно.

Итак, на одном диске может быть записано до миллиарда символов! Даже на скромном по возможностям гибком диске для персональной ЭВМ можно хранить около миллиона символов. Для сравнения заметим, то вся книга, которую Вы сейчас читаете, вместе со всеми текстами программ составляет также всего около миллиона символов.

Поэтому работать с диском на уровне блоков не очень удобно — объекты слишком велики, для того чтобы помнить, в каких блоках записана та или иная интересующая нас информация. Хотелось бы иметь какие-то более крупные логические единицы хранения информации на диске, соответствующие проектам в целом, реализациям отдельных исполнителей, программам и пр. Эти более крупные единицы принято называть *файлами*, и именно для работы с ними нужен исполнитель «Простейшая файловая система».

Постановка задачи. Надо реализовать исполнителя «Простейшая файловая система», который должен работать с некоторым количеством файлов на диске. Неформально файл — это множество определенным образом организованных блоков. Поскольку, однако, в файлах надо хранить отдельные программы, реализации тех или иных исполнителей и т. п., другими словами тексты, то файловая система должна обеспечивать работу с файлом на уровне строк. Точнее говоря, на базе файла должны легко реализовываться последовательность и стек строк — две самые распространенные (после вектора) структуры в программировании.

Более формально, требуется реализовать исполнителя «Простейшая файловая система» со следующей системой предписаний:

0. инициализировать диск
1. начать работу
2. можно создать файл : да/нет
3. создать файл ⟨**вых** : ф : ключ⟩
4. файл ⟨**вх** : ф : ключ⟩ пуст : да/нет
5. можно добавить строку ⟨**вх** : ф : ключ, **вх** : С : строка⟩ : да/нет
6. добавить строку в конец файла ⟨**вх/вых** : ф, **вх** : С : строка⟩
7. взять строку из конца файла ⟨**вх/вых** : ф, **вых** : С : строка⟩
8. в начале файла ⟨**вх** : ф : ключ⟩ : да/нет
9. в конце файла ⟨**вх** : ф : ключ⟩ : да/нет
10. встать в начало файла ⟨**вх/вых** : ф : ключ⟩
11. встать в конец файла ⟨**вх/вых** : ф : ключ⟩
12. прочесть строку вперед ⟨**вх/вых** : ф : ключ, **вых** : С : строка⟩
13. прочесть строку назад ⟨**вх/вых** : ф : ключ, **вых** : С : строка⟩
14. удалить файл ⟨**вх/вых** : ф : ключ⟩
15. кончить работу

Предписание «инициализировать диск» должно очищать диск от всякой информации (т. е. делать пустым). Особенность этого предписания связана с тем, что и сам диск — это особый исполнитель, который сохраняет свое состояние между «кончить» и «начать» работу (аналогично тому, как выключение и включение магнитофона не влияет на содержимое кассеты). Поэтому предписание «начать работу» в файловой системе должно начинать работу с тем диском, который уже есть. Дополнительные детали, связанные с инициализацией диска, будут ясны из реализации.

Предписания «создать файл»/«удалить файл» относятся к файлу в целом как к единому и неделимому объекту. По предписанию «создать» файловая система создает новый пустой файл и возвращает в выходном параметре уникальный *ключ файла*. Какая именно информация закодирована в ключе — внутреннее дело файловой системы. Все другие операции с файлом вплоть до его удаления можно производить, только указав ключ файла. Образно говоря, когда некто просит, например, добавить строку в конец файла, он должен «вручить», файловой системе ключ от файла и добавляемую строку. После этого файловая система «забирает ключ», добавляет строку к файлу, меняет ключ и возвращает новый измененный ключ. Следующую операцию с файлом надо проводить уже с помощью этого нового измененного ключа.

Изменение ключа при каждой операции с файлом не является самоцелью. Дело в том, что фактически в ключе содержится информация о текущем состоянии файла: расположение на диске,

пуст файл или нет и т. п. Поскольку при изменениях файла информация о нем также может измениться, то во всех предписаниях, меняющих файл, ключ является входно-выходным параметром.

Предписания 4—13 предназначены для работы внутри файла и являются симбиозом предписаний последовательности (встать в начало, прочесть вперед, добавить в конец) и стека (добавить в конец, взять из конца). Кроме того, они позволяют читать последовательность в обратном порядке (встать в конец, прочесть назад). Фигурирующий в системе предписаний тип «строка» — это

запись (длина : 0 .. 132,
текст : **вектор символ** (1 .. 132))

Другими словами, строка — это вектор символов некоторой фиксированной длины (например, длины 132) вместе с указанием, какая часть этого вектора является информационной. Например, для пустой строки поле «длина» будет иметь значение 0, а содержимое поля «текст» неважно. Наоборот, для строки максимально возможной длины все содержимое вектора в поле «текст» значимо.

От файловой системы требуется, чтобы она использовала место на диске, насколько это возможно, экономно, например чтобы пустые строки «почти» не занимали места.

Так как диск содержит ограниченное количество блоков, то в систему предписаний файловой системы включены предписания «можно создать файл» и «можно добавить строку», которые выполняют роль предписания «есть свободное место». Наличие двух предписаний, одно из которых к тому же имеет параметры, вызвано тем, что для создания файла или для добавления той или иной строки требуется разное «количество» места, и может оказаться, что одна из этих операций невыполнима, а другая выполнима.

Исполнителя «Простейшая файловая система» надо реализовать над заданным базовым исполнителем «Диск» со следующей системой предписаний:

1. начать работу
2. записать ⟨**вх** : В : блок⟩ в блок ⟨**вх** : k : номер блока⟩
3. прочесть блок ⟨**вх** : k : номер блока⟩ в ⟨**вых** : В : блок⟩
4. кончить работу

типы:

блок = **вектор** байт(1 .. 512)
байт = **символ**
номер блока = 0 .. 4095

Как уже отмечалось, особенность диска в том, что он сохраняет свое состояние между «кончить» и «начать» работу.

Обратите внимание, что система предписаний этого исполнителя совпадает с системой предписаний исполнителя «Вторичная память» из предыдущего раздела. Это связано с тем, что обычно под вторичную память отводится либо отдельный диск, либо некоторая фиксированная часть того же диска, на котором хранятся и тексты. Во втором случае файловая система либо не должна трогать отведенную под вторичную память область на диске, либо должна обеспечивать реализацию вторичной памяти над собой. Мы, однако, этими вопросами заниматься не будем.

Основные идеи реализации. Итак, задача состоит в том, чтобы реализовать файловую систему на базе диска. Поскольку файлов может быть много, а размеры файлов (измеряемые в блоках) могут меняться в процессе работы, то, чтобы избежать массовых операций с блоками, воспользуемся ссылочной реализацией. Здесь, однако, нельзя завести кроме диска вспомогательный вектор ссылок — ведь содержимое диска без информации о ссылках представляет просто «кучу обрывков от разных файлов», а требуется, чтобы файлы сохранялись между концом и началом работы файловой системы. Таким образом, ссылки тоже надо хранить где-то на диске. Имеется два естественных решения: 1) собрать все ссылки и хранить их в специально выделенном месте диска, например в начале; 2) хранить ссылки прямо в том блоке, к которому они относятся.

Мы выберем второе решение и будем в каждом блоке в первых 4 байтах хранить ссылку на следующий блок, а во вторых 4 байтах — ссылку на предыдущий. Ссылку будем записывать в виде четырехзначного десятичного числа, т. е. с помощью цифр от 0 до 9. Например, если следующим за блоком 3544 является блок 1673, то запишем в первые 4 байта 3544-го блока число 1673: в первый байт — цифру 1, во второй — 6, в третий — 7 и в четвертый — 3. Если ссылка меньше 1000, то дополним ее слева незначащими нулями до 4 цифр; например, 38 запишем как 0038.

Как обычно, число траекторий, на которые разбиваются блоки диска, будет равно числу файлов, увеличенному на 1 (т. е. на траекторию «свободное место»). В отличие от предыдущих ссылочных реализаций число траекторий файлов здесь не фиксировано, а меняется в процессе работы. Поэтому решим, что траектория «свободное место» будет начинаться с нулевого блока (нил свободного места), а траектории файлов специально выделенных служебных блоков нил иметь не будут — следующим за последним блоком в файле будет первый, а предыдущим для первого — последний.

При выполнении некоторых операций нам понадобится не только знать, есть или нет свободные блоки на диске, но и знать, сколько их. Поэтому введем в исполнителе «Простейшая файловая

система» глобальный объект, в котором будем хранить число свободных блоков. Это число, впрочем, между концом и началом работы файловой системы тоже надо хранить где-то на диске. Мы будем хранить его в нулевом блоке в байтах с номерами 9..12 сразу вслед за ссылками.

Наконец, надо понять, что же такое ключ файла. Взглянем на производимые с файлом операции. У файла кроме содержимого есть начало, конец и граница между прочитанной и непрочитанной частью. Примем решение, что ключ будет содержать всю эту информацию. Поскольку строки файла будут как-то располагаться в блоках траектории файла, то указание на границу между строками — это пара: указание на блок и указание на границу между байтами в этом блоке. Мы будем границу между байтами задавать индексом i левого (младшего) байта. Этот индекс удобен тем, что определен и для крайних случаев. Например, для положения перед первой строкой файла i = 8 (строки в любом блоке начинаются с 9 байта — байты 1..8 заняты ссылками). Таким образом, решим, что тип ключ будет выглядеть так:

ключ = запись (первый блок : номер блока,
прочитанный блок : номер блока,
прочитанный байт : номер байта,
последний блок : номер блока,
последний байт : номер байта,
число блоков в файле : число блоков)

Заметьте, что хранится только номер блока начала файла — номер байта хранить не надо, так как известно, что строки файла начинаются с 9-го байта в первом блоке. Число блоков в файле является полезной информацией, которая понадобится нам в дальнейшем.

Реализация. Мы решили все основные вопросы, кроме одного — как же именно будут располагаться строки вутри файла. Отложим пока решение этого вопроса и реализуем работу с файлами в целом, т. е. работу с файлами как с траекториями блоков.

исполнитель Простейшая файловая система
. **СП:**
. 0. инициализировать диск
.
. 1. начать работу
. 2. можно создать файл : да/нет
. 3. создать файл ⟨**вых** : ф : ключ⟩
.
. 4. файл ⟨**вх** : ф : ключ⟩ пуст : да/нет
. 5. можно добавить строку ⟨**вх** : ф : ключ, **вх** : С : строка⟩ : да/нет

```
. 6. добавить строку в конец файла ⟨вх/вых : ф : ключ,
.        вх : С : строка⟩
. —   добавить ⟨вх : d⟩ символов из ⟨вх : t⟩ в конец ⟨вх/вых : ф⟩
. 7. взять строку из конца файла ⟨вх/вых : ф : ключ,
.        вых : С : строка⟩
. —   взять ⟨вх : d⟩ символов из конца ⟨вх/вых : ф⟩ в ⟨вых : t⟩
. 8. в начале файла ⟨вх : ф : ключ⟩                          : да/нет
. 9. в конце файла ⟨вх : ф : ключ⟩                           : да/нет
. 10. встать в начало файла ⟨вх/вых : ф : ключ⟩
. 11. встать в конец файла ⟨вх/вых : ф : ключ⟩
. 12. прочесть строку вперед ⟨вх/вых : ф : ключ, вых : С : строка⟩
. —   прочесть вперед ⟨вх : d⟩ символов из ⟨ф⟩ в ⟨вых : t⟩
. 13. прочесть строку назад ⟨вх/вых : ф : ключ, вых : С : строка⟩
. —   прочесть назад ⟨вх : d⟩ символов из ⟨ф⟩ в ⟨вых : t⟩
. 14. удалить файл ⟨вх/вых : ф : ключ⟩
.
. — захватить блок ⟨вых : k : номер блока⟩
. — освободить блок ⟨вх : k : номер блока⟩
. — связать ⟨вх : k1, k2 : 0 .. макс номер блока на диске⟩
. — установить поле ⟨вх : i⟩ в блоке ⟨вх : k⟩ в ⟨вх : n⟩
. — прочесть число из поля ⟨i⟩ в ⟨вых : n⟩
. — записать число ⟨вх : n⟩ в поле ⟨i⟩
. — число ⟨вх : t⟩ : 0 .. макс число
. — текст ⟨вх : n⟩ : вектор символ (1 .. длина числа)
. 15. кончить работу
.
. константы:
.  макс номер блока на диске   = 4095
.  число байт в блоке          =  512
.  макс число символов в строке =  132
.  макс число                  = 4095 | т. е. максимум (4095, 132)
.  длина числа                 =    4 | т. е. число цифр в числе
.
. обозначения:
.  нил == 0
.  @ след    == 1                          | начала служебных полей
.  @ пред    == @ след + длина числа       | в любом блоке диска
.  последний служебный байт блока == 2 * длина числа
.  # свободных == @ пред + длина числа | только в нулевом блоке
.
```

Другими словами, в нулевом блоке под служебную информацию заняты первые 12 байт (и больше в нулевом блоке ничего нет), а во всех остальных блоках — первые 8 байт. Например, если

файл состоит из трех блоков 1000, 0038 и 2017, то первые байты этих блоков будут содержать следующие символы:

блок 1000	0	0	3	8	2	0	1	7	...
блок 0038	2	0	1	7	1	0	0	0	...
блок 2017	1	0	0	0	0	0	3	8	...
	поле «@след»				поле «@пред»				

```
. типы:
. номер символа = 1 .. макс число символов в строке
. строка = запись (длина: 0..макс число символов в строке,
.                  текст: вектор символ (номер символа))
. байт          = символ
. номер байта   = 1..число байт в блоке
. блок          = вектор байт (номер байта)
. номер блока   = 1..макс номер блока на диске
. число блоков  = 0..макс номер блока на диске
. ключ = запись (первый       блок: номер блока,
.                прочитанный  блок: номер блока,
.                прочитанный  байт: номер байта,
.                последний    блок: номер блока,
.                последний    байт: номер байта,
.                число блоков в файле: число блоков)
.
. объекты:
. первый свободный блок: 0..макс номер блока на диске
. число свободных      : число блоков
. буфер                : блок
.
```

Наличие объекта буфер требует пояснений. Система предписаний исполнителя диск позволяет прочесть или записать блок только целиком. Поэтому для получения любой информации из блока его сначала надо целиком прочесть в память. Именно для этого и предназначен объект буфер. При изменении части информации в блоке надо сначала весь блок прочесть в буфер, потом изменить соответствующую часть буфера и, наконец, записать весь буфер на диск.

```
. используемые исполнители:
. Диск
конец описаний |--------------------------------

программа инициализировать диск
. дано      :
. получить:
. --------
. Диск.начать работу
. первый свободный блок := нил
. число свободных       := 0
. связать ⟨нил,нил⟩
. цикл ∀k ∈ номер блока реверс выполнять
. .   освободить блок ⟨вх:k⟩
. конец цикла
конец программы

программа начать работу
. дано      :
. получить:
. --------
. Диск.начать работу
. Диск.прочесть блок ⟨нил⟩ в ⟨вых:буфер⟩
. прочесть число из поля ⟨@ след⟩ в ⟨вых:первый свободный блок⟩
. прочесть число из поля ⟨# свободных⟩ в ⟨вых:число свободных⟩
конец программы
```

Программа «прочесть число из поля ⟨вх : i⟩ в ⟨вых : n⟩» читает 4 (длина числа) байта из буфера, начиная с индекса i, и преобразует эту символьную информацию (последовательность из 4 цифр) в целое число — значение объекта n.

```
программа можно создать файл:да/нет == (число свободных > 0)
программа создать файл ⟨вых:ф:ключ⟩
. дано      : можно создать файл
. получить:
. --------
. захватить блок ⟨вых:k⟩
. связать ⟨k,k⟩
. ф.первый       блок := k
. ф.прочитанный блок := k
. ф.прочитанный байт := последний служебный байт блока
. ф.последний    блок := k
. ф.последний    байт := последний служебный байт блока
. ф.число блоков в файле := 1
конец программы
```

```
программа удалить файл ⟨вх/вых: ф: ключ⟩
. дано     :
. получить:
. --------
.  связать ⟨ф.последний блок, первый свободный блок⟩
.  первый свободный блок := ф.первый блок
.  число свободных.увеличить на ф.число блоков в файле
.  связать ⟨нил, ф.первый блок⟩
конец программы

программа захватить блок ⟨вых: k: номер блока⟩
. дано     : число свободных > 0
. получить:
. --------
.  k := первый свободный блок
.  Диск.прочесть блок ⟨вх: k⟩ в ⟨вых: буфер⟩
.  прочесть число из поля ⟨@ след⟩ в ⟨вых: первый свободный блок⟩
.  число свободных.уменьшить на 1
.  связать ⟨нил, первый свободный блок⟩
конец программы

программа освободить блок ⟨вх: k: номер блока⟩
. дано     :
. получить:
. --------
.  связать ⟨k, первый свободный блок⟩
.  первый свободный блок := k
.  число свободных.увеличить на 1
.  связать ⟨нил, k⟩
конец программы

программа связать ⟨вх: k1, k2: 0..макс номер блока на диске⟩
. дано     :
. получить: |установить ссылки между блоками k1 и k2 на диске,
.           |а кроме того, если попадется нулевой блок, то
.           |записать в него текущее состояние глобального
.           |объекта "число свободных"
. --------
.  установить поле ⟨вх: @ след⟩ в блоке ⟨вк: k1⟩ в ⟨вх: k2⟩
.  установить поле ⟨вх: @ пред⟩ в блоке ⟨вх: k2⟩ в ⟨вх: k1⟩
конец программы

программа установить поле ⟨вх: i⟩ в блоке ⟨вх: k⟩ в ⟨вх: n⟩
. дано   :  k: 0..макс номер блока на диске,
.           i: номер байта,
.           n: 0..макс число
```

```
. получить: | в блок k, начиная с байта i, записано число n
.           | в символьном виде. Если блок k нулевой, то
.           | кроме того, надо поле "# свободных" в нулевом
.           | блоке установить в состояние объекта
.           | "число свободных"
. --------
.  Диск.прочесть блок ⟨k⟩ в ⟨вых : буфер⟩
.  записать число ⟨n⟩ в поле ⟨i⟩
.    если k = нил то
.    . записать число ⟨число свободных⟩ в поле ⟨# свободных⟩
.    конец если
.  Диск.записать ⟨вх : буфер⟩ в блок ⟨k⟩
конец программы
```

Таким образом, мы реализовали всю работу с блоками. Заметьте, что всякий раз, когда число свободных блоков изменяется (по предписаниям «удалить файл», «захватить блок», «освободить блок»), измененное число свободных блоков записывается и в нулевой блок на диск. Так что после выполнения любого вызванного извне предписания файловой системы диск находится в корректном состоянии, т. е. содержит всю информацию о траекториях и числе свободных блоков. Поэтому по предписанию «кончить работу» ничего делать не надо:

```
программа кончить работу == Диск.кончить работу
```

Глобальные объекты «первый свободный блок» и «число свободных» являются, таким образом, просто копией соответствующих полей нулевого блока и введены только для оптимизации, чтобы не читать нулевой блок, когда нужно посмотреть на эту информацию.

Реализуем теперь преобразования чисел в последовательность из 4 цифр и обратно.

```
программа прочесть число из поля ⟨i⟩ в ⟨вых : n⟩
. дано     : i : номер байта
. получить: n : 0..макс число
. --------
.  n := число ⟨буфер (i..i + длина числа)⟩
конец программы

программа записать число ⟨вх : n⟩ в поле ⟨i⟩
. дано     : i : номер байта,
.            n : 0..макс число
. получить:
. --------
.  буфер (i..i + длина числа) := текст ⟨вх : n⟩
конец программы
```

Использованная в этих двух программах запись

буфер (i..i + длина числа),

называемая *вырезкой*, служит обозначением для части вектора буфер, начиная с элемента с индексом i и кончая элементом с индексом i + длина числа (т. е. состоит из 4 элементов). С этой вырезкой можно работать так, как если бы это был отдельный 4-элементный вектор символов; в частности, можно присвоить вырезке значение другого вектора символов из 4 элементов — вектора, который вырабатывается в качестве значения предписания «текст».

```
программа число ⟨вх : t⟩ : 0..макс число
. дано      : t : вектор символ (1..длина числа)
. получить:
. --------
. ответ := 0
. цикл ∀ i ∈ 1.. длина числа выполнять
. . ответ := ответ * 10 + (код (t (i)) — код ("0"))
. конец цикла
конец программы
```

Функция *код* сопоставляет каждому символу целое число — его код (например, букве A — число 65, букве B — число 66 и т. п.). Коды цифр идут подряд, т. е. код("1") = код("0") + 1, код("2") = = код("1") + 1 и т. п. Таким образом, разность

код (t (i)) — код ("0")

это и есть значение типа число, соответствующее цифре t(i).

Обратное преобразование из числа, являющегося кодом некоторого символа, в сам символ осуществляет функция «символ». Обе эти функции являются стандартными (см. приложение 1).

```
программа текст ⟨вх : n⟩ : вектор символ (1..длина числа)
. дано      : n : 0..макс число
. получить:
. --------
. s := n
. цикл ∀ i ∈ 1..длина числа реверс выполнять
. . ответ (i) := символ (s mod 10)
. . s := частное (s, 10)
. конец цикла
конец программы
```

Итак, мы реализовали все программы, работающие с файлами в целом, включая и необходимые подпрограммы. Теперь надо реализовать предписания 4—13, работающие внутри файла. Некоторые

из этих предписаний, очевидно, могут быть реализованы на базе ключа файла вообще без каких-либо действий с диском и без знания расположения строк файла в блоках:

программа файл ⟨**вх** : ф : ключ⟩ пуст : **да/нет** ==
(ф.последний блок = ф.первый блок **и**
ф.последний байт = последний служебный байт блока)
программа встать в начало файла ⟨**вх/вых** : ф : ключ⟩ ==
ф.прочитанный блок := ф.первый блок
ф.прочитанный байт := последний служебный байт блока
программа в начале файла ⟨вх : ф : ключ⟩ : **да/нет** ==
(ф.прочитанный блок = ф.первый блок **и**
ф.прочитанный байт = последний служебный байт блока)
программа встать в конец файла ⟨**вх/вых** : ф : ключ⟩ ==
ф.прочитанный блок := ф.последний блок
ф.прочитанный байт := ф.последний байт
программа в конце файла ⟨**вх** : ф : ключ⟩ : **да/нет** ==
(ф.прочитанный блок = ф.последний блок **и**
ф.прочитанный байт = ф.последний байт)

Осталось реализовать предписания, которые собственно и составляют основу работы с файлом: добавление и взятие строк в конце файла, чтение строк вперед и назад, а также анализ, можно ли добавить строку. Таким образом, пора разобраться с последним отложенным вопросом и решить, как же именно будут храниться строки в блоках. Поскольку в требования к файловой системе входило, насколько это возможно, экономное использование места на диске, то будем хранить только информационные части строк, т. е. части с индексами от 1 до длины строки, а «мусор» — компоненты строк с индексами большими длины строки — никуда не записывать. Поскольку, кроме того, надо уметь читать строки и вперед, и назад, то длину строки (4 байта в таком же виде, как и ссылки) будем записывать на диск до и после ее информационной части. При этом способе пустая строка будет занимать 8 байт на диске: 4 байта длины (т. е. четыре символа 0), потом информационная часть, которая в этом случае не занимает ни одного байта, и затем еще раз 4 байта длины.

Если при чтении строки мы дойдем до границы блока, то следующие байты будем читать из начала следующего блока. Другими словами, при выбранном способе хранения строк на диске строка может начинаться в одном блоке, а заканчиваться в другом. В частности, 4 байта для длины строки, о которых говорилось выше, могут оказаться в разных блоках. Если граница блока встретится при добавлении строки к файлу, то в траекторию файла надо добавить новый блок и продолжать запись в него.

Вот несколько примеров размещения строк в блоках файла. Файл, состоящий из двух строк

Я Вас любил,
Любовь еще, быть может,

размещенный в блоке с номером 1000 (точки обозначают символы, значения которых неизвестны или неважны):

1000100000012Я Вас любил,00120023Любовь еще, быть может,0023...

файл, состоящий из трех строк

0666
1611011
1986

размещенный в блоке с номером 1000:

1000100000040666000400071611011000700041986000 4...

С содержательной точки зрения эта последовательность символов состоит из следующих групп:

@сл	@пр	d_1	t_1	d_1	d_2	t_2	d_2	d_3	t_3	d_3	
1000	1000	0004	0666	0004	0007	1611011	0006	0004	1986	0004	...

где @сл, @пр — ссылки на следующий и предыдущий блоки в кольце блоков файла; d_1, t_1 — длина и информационная часть 1-й строки текста.

Наконец, приведем пример размещения строки "++++A++++A++++" длины 14, переходящей из блока 0038 в блок 1000:

блок 0038	1000	...				...	001
блок 1000	....	0038	4	++++A++++A++++	0014	...	

программа можно добавить строку
⟨вх : ф : ключ, С : строка⟩ : да/нет ==
((число байт в блоке — ф.последний байт) +
число свободных *
(число байт в блоке — последний служебный байт блока) ⩾
С.длина + 2 * длина числа)
программа добавить строку в конец файла
⟨вх/вых : ф, вх : С : строка⟩

```
. дано     : можно добавить строку ⟨вх : ф : ключ, вх : С : строка⟩
. получить:
. --------
.  t := текст ⟨С.длина⟩
.  добавить ⟨длина числа⟩ символов из ⟨t⟩          в конец ⟨ф⟩
.  добавить ⟨С.длина⟩     символов из ⟨С.текст⟩ в конец ⟨ф⟩
.  добавить ⟨длина числа⟩ символов из ⟨t⟩          в конец ⟨ф⟩
конец программы

программа добавить ⟨вх : d⟩ символов из ⟨вх : t⟩ в конец ⟨вх/вых : ф⟩
. дано     : t : вектор элементов типа символ с индексом 1 .. d
. получить:
. --------
.  Диск.прочесть блок ⟨ф.последний блок⟩ в ⟨вых : буфер⟩
.  цикл ∀ i ∈ 1..d выполнять
. .  ф.последний байт.увеличить на 1
. .  если ф.последний байт > число байт в блоке то
. . .  записать число ⟨первый свободный блок⟩ в поле ⟨@ след⟩
. . .  Диск.записать ⟨вх : буфер⟩ в блок ⟨ф.последний блок⟩
. . .
. . .  захватить блок ⟨вых : k⟩
. . .  записать число ⟨ф.первый блок⟩     в поле ⟨@ след⟩
. . .  записать число ⟨ф.последний блок⟩ в поле ⟨@ пред⟩
. . .                                                  -
. . .  ф.последний блок := k
. . .  ф.последний байт := последний служебный байт блока
. . .                                                  +1
. .  конец если
. .  буфер (ф.последний байт) := t(i)
.  конец цикла
.  Диск.записать ⟨вх : буфер⟩ в блок ⟨ф.последний блок⟩
конец программы

программа взять строку из конца файла
.                                  ⟨вх/вых : ф, вых : С : строка⟩
. дано     : файл ⟨вх : ф⟩ не пуст
. получить:
. объекты :
.  t : вектор элементов типа символ с индексом 1..длина числа
. --------
.  взять ⟨длина числа⟩ символов из конца ⟨ф⟩ в ⟨вых : t⟩
.  С.длина := число ⟨вх : t⟩
.  взять ⟨С.длина⟩     символов из конца ⟨ф⟩ в ⟨вых : С.текст⟩
.  взять ⟨длина числа⟩ символов из конца ⟨ф⟩ в ⟨вых : t⟩
конец программы
```

```
программа взять ⟨вх : d⟩ символов из конца ⟨вх/вых : ф⟩ в ⟨вых : t⟩
. дано      :
. получить: t : вектор элементов типа символ с индексом 1..d
. --------
.  если в конце файла ⟨вх : ф⟩ то были в конце := да конец если
.  Диск.прочесть блок ⟨ф.последний блок⟩ в ⟨вых : буфер⟩
.  цикл ∀ i ∈ 1..d реверс выполнять
.  .   утв: ф.последний байт > последний служебный байт блока
.  .   t(i) := буфер (ф.последний байт)
.  .   ф.последний байт.уменьшить на 1
.  .   если ф.последний байт = последний служебный байт блока
.  .   . то
.  .   .  k := ф.последний блок
.  .   .  прочесть число из поля ⟨@ пред⟩ в ⟨ф.последний блок⟩
.  .   .  ф.последний байт := число байт в блоке
.  .   .
.  .   .  освободить блок ⟨вх : k⟩
.  .   .  связать ⟨ф.последний блок, ф.первый блок⟩
.  .   .  Диск.прочесть блок ⟨ф.последний блок⟩ в ⟨вых : буфер⟩
.  .   конец если
.  конец цикла
.  Диск.записать ⟨вх : буфер⟩ в блок ⟨ф.последний блок⟩
.  если были в конце то встать в конец файла ⟨ф⟩ конец если
конец программы

программа прочесть строку вперед
.                                  ⟨вх/вых : ф : ключ, вых : С : строка⟩
. дано      : не в конце файла ⟨вх : ф : ключ⟩
. получить:
. объекты :
.  t : вектор элементов типа символ с индексом 1..длина числа
. --------
.  прочесть вперед ⟨длина числа⟩ символов из ⟨ф⟩ в ⟨t⟩
.  С.длина := число ⟨вх : t⟩
.  прочесть вперед ⟨С.длина⟩     символов из ⟨ф⟩ в ⟨С.текст⟩
.  прочесть вперед ⟨длина числа⟩ символов из ⟨ф⟩ в ⟨t⟩
конец программы

программа прочесть вперед ⟨вх : d⟩ символов из ⟨ф⟩ в ⟨вых : t⟩
. дано      :
. получить: t : вектор символ (1..d)
. --------
.  Диск.прочесть блок ⟨ф.прочитанный блок⟩ в ⟨вых : буфер⟩
.  цикл ∀ i ∈ 1..d выполнять
.  .   ф.прочитанный байт.увеличить на 1
```

```
. . если ф.прочитанный байт > число байт в блоке то
. . . прочесть число из поля ⟨@след⟩ в ⟨ф.прочитанный блок⟩
. . . ф.прочитанный байт := последний служебный байт блока
. . .                                                +1
. . . Диск.прочесть блок ⟨ф.прочитанный блок⟩ в ⟨вых: буфер⟩
. . конец если
. . t(i) := буфер (ф.прочитанный байт⟩
. конец цикла
конец программы

программа прочесть строку назад ⟨вх/вых: ф: ключ, вых: С: строка⟩
. дано     : не в начале файла ⟨вх: ф: ключ⟩
. получить:
. объекты :
. t: вектор элементов типа символ с индексом 1..длина числа
. --------
. прочесть назад ⟨длина числа⟩ символов из ⟨ф⟩ в ⟨t⟩
. С.длина := число ⟨вх: t⟩
. прочесть назад ⟨С.длина⟩      символов из ⟨ф⟩ в ⟨С.текст⟩
. прочесть назад ⟨длина числа⟩ символов из ⟨ф⟩ в ⟨t⟩
конец программы

программа прочесть назад ⟨вх: d⟩ символов из ⟨ф⟩ в ⟨вых: t⟩
. дано     :
. получить: t: вектор символ (1..d)
. --------
. Диск.прочесть блок ⟨ф.прочитанный блок⟩ в ⟨вых: буфер⟩
. цикл ∀ i ∈ 1..d реверс выполнять
. . t(i) := буфер (ф.прочитанный байт)
. . ф.прочитанный байт.уменьшить на 1
. . если ф.прочитанный байт = последний служебный байт блока
. . . то
. . . прочесть число из поля ⟨@пред⟩ в ⟨ф.прочитанный блок⟩
. . . ф.прочитанный байт := число байт в блоке
. . . Диск.прочесть блок ⟨ф.прочитанный блок⟩ в ⟨вых: буфер⟩
. . конец если
. конец цикла
конец программы
конец исполнителя |==========================
```

ЗАДАЧИ И УПРАЖНЕНИЯ

1. Модифицируйте проект так, чтобы в случае, когда нужный блок уже находится в буфере, он с диска не читался, чтобы изменения информации сначала происходили только в буфере и чтобы буфер записывался на диск только тогда, когда в этом действи-

тельно есть необходимость (аналогично проекту «Виртуальная память» из предыдущего раздела).

2. Модифицируйте проект так, чтобы можно было добавлять и брать строки не только в конце, но и в начале файла (т. е. чтоб над файловой системой можно было реализовать и дек тоже).

3. Существующий проект работает только с диском фиксированного размера, заданного заранее с помощью константы «макс номер блока на диске». Модифицируйте проект так, чтобы он работал с диском любых размеров, считывая действительное число блоков на диске из специально отведенного для этой цели места в нулевом блоке. В предписании «инициализировать» размер диска в блоках следует указывать во входном параметре.

4. Измените реализацию Простейшей файловой системы так, чтобы попытка выполнить любую операцию с файлом после его удаления приводила к отказу.

Указание. Добавьте в предписания, у которых ключ файла является входным или входно-выходным параметром, какой-нибудь анализ корректности состояния ключа, например проверку, что

ключ.число блоков в файле $>$ 0.

А в программу удаления файла добавьте «разрушение» ключа, т. е. установку в некорректное состояние.

5. Модифицируйте проект так, чтобы все ссылки хранились одним «куском» в начале диска, считывались в память по предписанию «начать работу» файловой системы и записывались обратно по предписанию «кончить работу».

6. Используя исполнителя «Простейшая файловая система» и, если надо, изменяя его, реализуйте исполнителя «Именная файловая система», в которой каждый файл имеет *имя* (вектор символов фиксированной длины, например длины 12). Между концом и началом работы файловой системы должны сохраняться не только сами файлы, но и их имена. При создании или удалении файла в качестве входного параметра указывается его имя. Предписания для работы внутри файла используют не имя, а ключ (как и в реализации выше); ими можно пользоваться только между «начать работу с файлом ⟨вх : имя, вых : ключ⟩» и «кончить работу с файлом ⟨вх : ключ⟩». Таким образом, ключ существует только на время работы с файлом и получается по имени файла предписанием «начать работу с файлом». (Именно такие файловые системы обычно используются при работе на ЭВМ.)

7. Сделайте из буфера буферный пул (т. е. увеличьте его размеры с одного блока до нескольких). Реализуйте файловую систему, используя для ускорения ее работы идеи реализации виртуальной памяти из предыдущего раздела.

18. Иерархия структур данных при разработке программ

При решении практических задач обычно не удается обойтись одними лишь стандартными структурами данных (стек, дек, очередь, список и т. д.), перечисленными в разд. 7. Реализация достаточно сложных исполнителей требует введения структур данных, максимально отражающих существо выполняемой исполнителем задачи. Такие специальные структуры данных обычно оформляются как вспомогательные исполнители и разрабатываются по технологии «сверху вниз». В этом разделе мы реализуем структуру данных, естественно возникающую при реализации текстового редактора.

При работе на ЭВМ недостаточно уметь записывать информацию в файл и читать ее из файла. Очень часто возникает задача *редактирования* (т. е. изменения) текстового файла. Эта задача обычно решается человеком с помощью специальной программы — экранного текстового редактора. Хотя к этому моменту Вы, наверное, достаточно поработали за терминалом и представляете себе внешний вид и возможности такого редактора, опишем его основные черты.

При работе человека с редактором на экране с помощью специальной отметки (*курсора*) изображается некоторое положение в тексте и умещающаяся на экране окрестность текста вокруг этого положения (назовем ее *окном*). Набор стандартных операций редактирования включает, в частности, вставку, удаление, изменение символов и строк. Перемещая курсор по тексту и изменяя текст возле курсора, можно внести в текст любые желаемые изменения.

Займемся реализацией структуры данных, соответствующей тексту с отмеченным положением и изображаемым окном, т. е. реализуем некоторую часть экранного текстового редактора. Для простоты не будем рассматривать операции изменения строки и горизонтальные перемещения курсора, а ограничимся только работой по вертикали — будем считать строки элементарными неделимыми объектами и интересоваться только вертикальной компонентой положения курсора. Удобной и отвечающей представлениям человека о тексте структурой является список строк — вертикальной компоненте положения курсора соответствует положение указателя, в районе указателя можно вставлять и удалять строки. Существуют, впрочем, и отличия текста от обычного списка: в тексте должна быть выделена постоянно изображаемая на экране терминала окрестность указателя (окно); в начале работы текст должен содержать информацию из изменяемого файла; после окончания работы новый текст должен записываться куда-то на диск и т. п.

Реализованная нами простейшая файловая система (как, впрочем, и почти все «настоящие» файловые системы) не позволяет

вставлять и удалять строки в середине файла. Таким образом, структуру текста строк с изображаемым окном и указателем надо как-то реализовывать над другими структурами данных и исполнителями. Среди этих структур и исполнителей кроме файловой системы можно сразу выделить исполнителя «Экран», который должен попавшие в окно строки текста каким-то образом изображать на экране терминала.

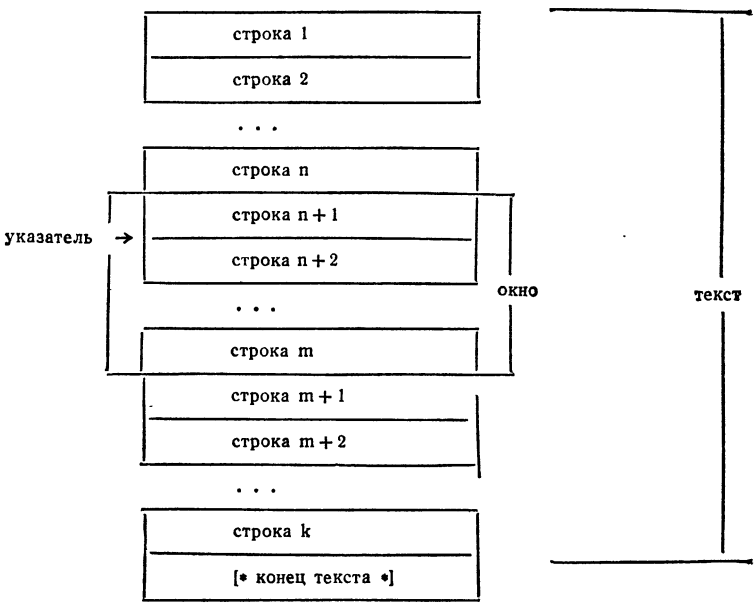
Мы будем считать, что исполнитель «Экран» с внешней точки зрения является вектором элементов типа строка с индексом от 0 до макс Yэкрана, где строки-элементы вектора соответствуют строкам, изображаемым на экране терминала. Например, чтобы изменить верхнюю строку на экране терминала, надо изменить элемент вектора с индексом 0, а чтобы узнать состояние последней (нижней) строки экрана, надо проанализировать элемент вектора с индексом макс Yэкрана. Реализация исполнителя «Экран» зависит от конкретной ЭВМ и от конкретного терминала. На персональных ЭВМ, например, на экране обычно изображается содержимое некоторого фиксированного участка оперативной памяти (этот участок называется видеопамятью). Реализация исполнителя «Экран» при этом сводится к преобразованию действий над строками в действия над соответствующими фрагментами видеопамяти. Мы будем считать исполнителя «Экран» базовым и реализовывать его не будем.

Итак, пусть стоит задача реализовать исполнителя «Текст строк с изображаемым окном и указателем» со следующей системой предписаний:

исполнитель Текст строк с изображаемым окном и указателем
. **СП:**
. 1. начать работу ⟨**вх**: ф: ключ⟩
. 2. установить указатель и окно в начало текста
. 3. установить указатель и окно в конец текста
. 4. указатель в начале текста : **да/нет**
. 5. указатель в конце текста : **да/нет**
. 6. передвинуть указатель назад возможно с окном
. — роллировать вниз ⟨**вх**: i1, i2: 0..макс Yэкрана⟩
. 7. передвинуть указатель вперед возможно с окном
. — роллировать вверх ⟨**вх**: i1, i2: 0..макс Yэкрана⟩
. 8. добавить строку ⟨**вх**: C: строка⟩ за указателем
. 9. удалить строку за указателем
. 10. строка за указателем :: строка
. 11. Yокна максимальное : 0..макс Yэкрана
. 12. получить положение указателя и окна в ⟨**вых**: Yтекста, Yокна⟩
. 13. установить указатель и окно в положение ⟨**вх**: Yтекста, Yокна⟩

14. кончить работу аварийно ⟨**вых** : ф : ключ⟩
15. кончить работу ⟨**вых** : ф : ключ⟩

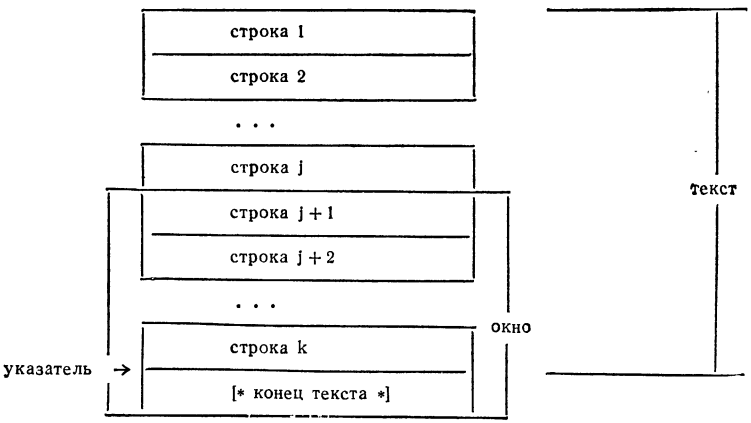
Предписания 2—10 образуют, если можно так выразиться, «Л1.5-список» — возможности по установке и перемещению указателя ссвпадают с возможностями Л2-списка, однако все действия проводятся только за указателем, как в Л1-списке. Предписаний, делающих список пустым и проверяющих пустоту или непустоту, нет вовсе. Предписания «роллировать вверх»/«роллировать вниз» являются локальными и будут объяснены ниже. При попытке с помощью предписания 6 или 7 вывести указатель за пределы окна окно смещается в соответствующем направлении на одну строку, чтобы новое положение указателя оказалось в новом окне. Таким образом, можно представлять себе текст в виде



Здесь текст состоит из k строк, окно содержит строки с n + 1-й по m-ю, указатель расположен между строками n + 1 и n + 2.

Обратите внимание на так называемую *холостую строку* ниже текста. Это строка не входит в текст. Дело в том, что, хотя ука-

затель располагается *между* строками, на экране он изображается в виде курсора (заштрихованного прямоугольника) *в строке* под указателем. Поскольку возможных положений указателя на 1 больше, чем строк в тексте, то положение указателя после последней строки текста изображается курсором в следующей за ней строке экрана. Чтобы отличить эту строку от строк текста, изобразим в ней текст «[* конец текста *]». Таким образом, возможно, например, следующее положение окна и указателя:



Обратите внимание на то, что границы окна, как и положение указателя, всегда располагаются между строками текста.

Предписание «Yокна максимальное» вырабатывает в качестве значения текущее число строк в окне без единицы. Мы будем хранить это число в глобальном объекте макс Yокна. В общем случае макс Yокна совпадает с константой макс Yэкрана. Однако в пустом файле, например, макс Yокна будет равно нулю, что означает, что в окне находится только одна строка — холостая. Yтекста в предписаниях получения/установки положения указателя и окна — это число строк текста над указателем, Yокна — это число строк окна над указателем. На изображенной выше картинке Yтекста = k, Yокна = макс Yокна = k — j. На предыдущей картинке Yтекста = n + 1, Yокна = 1, а макс Yокна = m — n — 1.

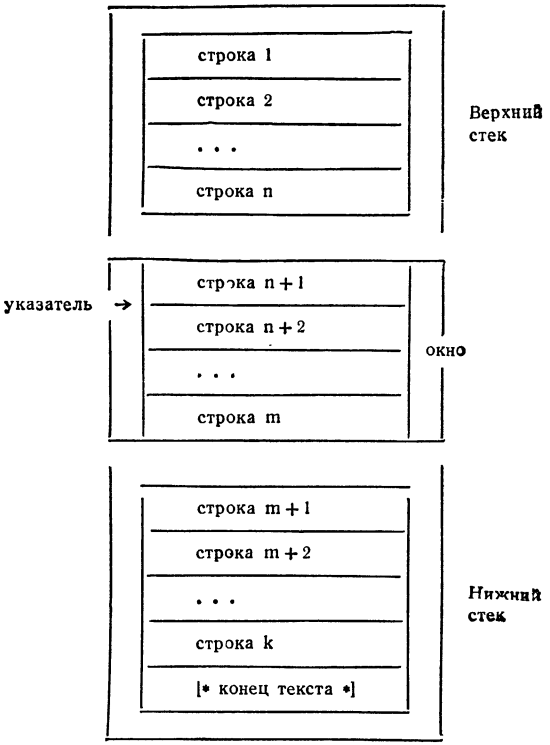
Наконец, кончить работу с исполнителем «Текст» можно двояко — аварийно и нормально. При аварийном окончании все внесенные человеком изменения аннулируются, исходный текст остается нетронутым и в выходном параметре возвращается ключ файла,

содержащего исходный нетронутый текст. При нормальном окончании новый измененный текст запоминается в некотором файле на диске, старый файл уничтожается, а в выходном параметре возвращается ключ файла с новым текстом.

```
. константы:
. число строк на экране = 24
. макс Yэкрана          = число строк на экране — 1
. макс длина строки     = 132 | символа
.
. типы:
. ключ = ...
. строка = запись (длина: 0..макс длина строки,
.                  текст: вектор символ (1..макс длина строки))
.
. объекты:
. макс Yокна : —1..макс Yэкрана | —1 — когда в окне нет ничего,
.                               | даже холостой строки
. Yокна      : 0..макс Yэкрана  | точнее 0..макс Yокна
.
. используемые исполнители:
. Экран : вектор строка (0..макс Yэкрана),
. Верхний стек, Нижний стек,
. Простейшая файловая система (ФС)
.
конец описаний |--------------------------------
```

Идеи реализации. Прежде всего заметим, что редактор должен уметь редактировать любые файлы, в том числе и те, размеры которых превышают размеры памяти. Таким образом, простейшее решение — реализовать весь текст целиком в памяти, в начале работы считывать в память текст из исходного файла, а в конце работы новый текст из памяти записывать в файл на диск — не проходит. Поэтому будем действовать по технологии «сверху вниз» — реализуем искомую структуру над удобными промежуточными, далее реализуем эти промежуточные структуры и т. д., пока не дойдем до базовых.

На первом шаге декомпозиции реализуем Текст на базе исполнителей «Нижний стек», «Экран» и «Верхний стек» таким образом, что расположенные над окном строки текста будут храниться в исполнителе «Верхний стек», строки окна — в исполнителе «Экран», а строки ниже окна — в исполнителе «Нижний стек»:



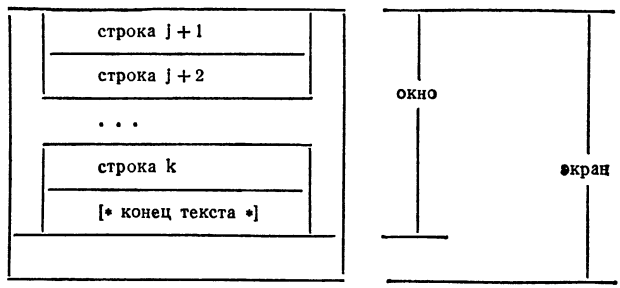
Положение указателя в тексте и в окне будем хранить в виде двух величин: Yокна (т. е. числа строк окна над указателем) и числа строк в верхнем стеке. Таким образом, потребуем, чтобы исполнитель «Верхний стек» имел предписание «число строк», вырабатывающее соответствующее значение. При окончании работы с верхним стеком его содержимое должно записываться в какой-то файл на диск.

При начале работы с нижним стеком он должен содержать строки исходного файла, заканчивающиеся холостой строкой «[* конец текста *]». В частности, предписание «пуст» нижнего стека вырабатывает значение да только в том случае, когда из стека взята и эта строка тоже. Напомним, что конец текста находится перед холостой строкой.

Поскольку по предписаниям «кончить работу» исполнитель «Текст строк с изображаемым окном и указателем» должен выдавать наружу ключ исходного или нового файла, то потребуем,

чтобы предписание «кончить работу» нижнего стека возвращало ключ исходного файла, а «кончить работу» верхнего — ключ нового файла.

Окно, естественно, может иногда занимать не весь экран. В этом случае (при макс Yокна < макс Yэкрана) будем размещать строки окна в верхней части экрана:



Мы реализуем исполнителя «Текст» так, что после вызова любого предписания Текста *извне* этого исполнителя будет выполнено следующее условие (*инвариант исполнителя*):

если макс Yокна < макс Yэкрана **то** Нижний стек.пуст

т. е. если нижний стек не пуст, то окно полно (занимает весь экран).

Поскольку положение указателя изображается курсором в строке под указателем, то реализуем Текст так, что указатель никогда не будет расположен за последней строкой окна. Иными словами,

$0 \leqslant$ Yокна $\leqslant$ макс Yокна = число строк окна $- 1$.

```
программа начать работу (вх : ф : ключ)
. дано      :
. получить: указатель в начале текста | а окно имеет максимально
.           | возможные размеры и прижато к верхнему краю
.           | текста
. --------
. Нижний стек.начать работу (вх : ф : ключ)
. Экран.начать работу
. Верхний стек.начать работу
. макс Yокна := -1
. цикл пока Нижний стек.не пуст и макс Yокна < макс Yэкрана
. . выполнять
. .  макс Yокна.увеличить на 1
```

```
. .  Нижний стек.взять строку в ⟨вых: Экран (макс Yокна)⟩
. конец цикла
. Yокна := 0
конец программы
```

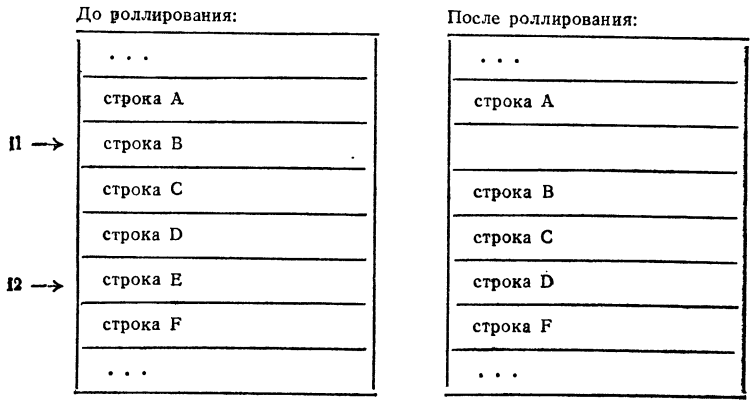
После выполнения этой программы макс Yокна окажется неотрицательным, так как цикл хотя бы один раз выполнится: даже при пустом тексте нижний стек содержит холостую строку и, значит, не пуст.

```
программа установить указатель и окно в начало текста
. дано      :
. получить: указатель в начало текста
. --------
. цикл пока указатель не в начале текста выполнять
. .  передвинуть указатель назад возможно с окном
. конец цикла
конец программы

программа установить указатель и окно в конец текста
. дано      :
. получить: указатель в конце текста
. --------
. цикл пока указатель не в конце текста выполнять
. .  передвинуть указатель вперед возможно с окном
. конец цикла
конец программы

программа указатель в начале текста : да/нет ==
                                (Верхний стек.пуст и Yокна = 0)
программа указатель в конце текста : да/нет ==
                              (Нижний стек.пуст и Yокна = макс Yокна)
программа передвинуть указатель назад возможно с окном
. дано      : указатель не в начале текста
. получить:
. --------
. если Yокна > 0 то Yокна.уменьшить на 1 иначе
. .  если макс Yокна < макс Yэкрана то
. .  .  макс Yокна.увеличить на 1
. .  .  иначе
. .  .  Нижний стек.добавить строку ⟨вх: Экран (макс Yокна)⟩
. .  конец если
. .  роллировать вниз ⟨0, макс Yокна⟩
. .  Верхний стек.взять строку в ⟨вых: Экран (0)⟩
. конец если
конец программы
```

Мы воспользовались здесь локальным предписанием «роллировать вниз ⟨**вх**: i1, i2⟩», которое сдвигает вниз на одну строку все строки окна с i1-й по 12-ю включительно. При этом самая нижняя строка — строка с номером 12 — пропадает, верхняя строка — строка с номером i1 — становится пустой, а строки с номерами вне диапазона i1 .. i2 не изменяются, например



(при i1 = i2 это просто очистка строки).

программа роллировать вниз ⟨**вх**: i1, i2: 0..макс Yэкрана⟩
. **дано** : i1 ⩽ i2
. **получить**: | строки окна в диапазоне i1..i2 сдвинуты на одну
. | строку вниз. Старая строка с номером i2 пропала,
. | новая строка с номером i1 пуста
. ————————
. **цикл ∀** i ∈ i1 + 1..i2 **реверс выполнять**
. . Экран (i) := Экран (i — 1)
. **конец цикла**
. Экран (i1).длина := 0
конец программы

программа передвинуть указатель вперед возможно с окном
. **дано** : указатель не в конце текста
. **получить**:
. ————————
. **если** Yокна < макс Yокна **то** Yокна.**увеличить на** 1 **иначе**
. . Верхний стек.добавить строку ⟨**вх**: Экран (0)⟩
. . роллировать вверх ⟨0, макс Yокна⟩
. . Нижний стек.взять строку в ⟨**вых**: Экран (макс Yокна)⟩
. **конец если**
конец программы

Программа «роллировать вверх» действует аналогично программе «роллировать вниз», но сдвигает строки с i1-й по i2-ю включительно на одну строку вверх. При этом старая строка с номером i1 пропадает, а новая строка с номером i2 становится пустой:

```
программа роллировать вверх (вх: i1, i2: 0..макс Yэкрана)
. дано     : i1 ⩽ i2
. получить: | строки окна в диапазоне i1..i2 сдвинуты на одну
.           | строку вверх. Старая строка с номером i1 пропала,
.           | новая строка с номером i2 пуста
. --------
. цикл ∀ i ∈ i1..i2 — 1 выполнять
. .  Экран (i) := Экран (i + 1)
. конец цикла
. Экран (i2).длина := 0
конец программы

программа добавить строку (вх: С: строка) за указателем
. дано     :
. получить:
. --------
. если макс Yокна = макс Yэкрана то
. .  Нижний стек.добавить строку (вх: Экран (макс Yокна))
. . иначе
. .  макс Yокна.увеличить на 1
. конец если
. роллировать вниз (Yокна, макс Yокна)
. Экран (Yокна) := С
конец программы

программа удалить строку за указателем
. дано     : указатель не в конце текста
. получить:
. --------
. роллировать вверх (Yокна, макс Yокна)
. если Нижний стек.не пуст то
. .  Нижний стек.взять строку в (вых: Экран (макс Yокна))
. . иначе
. .  макс Yокна.уменьшить на 1
. конец если
конец программы

программа строка за указателем :: строка
. дано     : указатель не в конце текста
```

```
. получить:
. --------
. ответ == Экран (Yокна)
конец программы

программа Yокна максимальное :0..макс Yэкрана == макс Yокна

программа получить положение указателя и окна в ⟨Yтекста, Yo⟩
. дано    :
. получить:
. --------
. Yтекста := Верхний стек.число строк + Yокна
. Yo      := Yокна
конец программы

программа установить указатель и окно в положение ⟨Yтекста, Yo⟩
. дано    :
. получить:
. --------
. dY := Yтекста—Yo—Верхний стек.число строк
. выбор
. . при dY > 0 ⇒
. .      цикл dY раз выполнять
. .      . передвинуть указатель вперед возможно с окном
. .      конец цикла
. . при dY < 0 ⇒
. .      цикл — dY раз выполнять
. .      . передвинуть указатель назад возможно с окном
. .      конец цикла
. конец выбора
.   утв: Yтекста—Yo=Верхний стек.число строк и Yo ⩽макс Yокна
. Yокна := Yo
конец программы

программа кончить работу аварийно ⟨вых : ф : ключ⟩
. дано    :
. получить: | ф — ключ старого нетронутого файла
. --------
. Верхний стек.кончить работу ⟨вых : фв : ключ⟩
. ФС.удалить файл ⟨вх : фв⟩
. Экран.кончить работу
. Нижний стек.кончить работу ⟨вых : ф : ключ⟩
конец программы

программа кончить работу ⟨вых : ф : ключ⟩
. дано    :
. получить: | ф — ключ нового измененного файла
```

```
. --------
. установить указатель и окно в конец текста
.   утв: Нижний стек.пуст | и холостая строка на экране
. цикл ∀ i ∈ 0..макс Yокна — 1 | т. е. без холостой строки
. . выполнять
. .  Верхний стек.добавить строку ⟨вх : Экран (i)⟩
. конец цикла
.
. Верхний стек.кончить работу ⟨вых : ф : ключ⟩
. Экран. кончить работу
. Нижний стек.кончить работу ⟨вых : фи : ключ⟩
. ФС.удалить файл ⟨вх : фи⟩
конец программы

конец исполнителя |==========================
```

Мы совершили первый шаг декомпозиции и получили реализацию структуры данных (исполнителя) «Текст строк с изображаемым окном и указателем» на базе исполнителей «Экран», «Простейшая файловая система», «Верхний стек» и «Нижний стек». Исполнителя «Экран» мы считаем базовым. Простейшая файловая система была реализована в предыдущем разделе. Таким образом, осталось реализовать верхний и нижний стеки.

Реализация исполнителя «Верхний стек» над файловой системой тривиальна, поэтому приведем ее без каких-либо пояснений:

```
исполнитель Верхний стек
. СП:
.   1. начать работу
.   2. пуст              : да/нет
.   3. число строк       : Z+
.   4. добавить строку ⟨вх : С : строка⟩
.   5. взять строку в ⟨вых : С : строка⟩
.   6. кончить работу ⟨вых : фв : ключ⟩
.
. объекты:
.   ф : ключ | ключ файла, на котором имитируется стек
.   n : Z+   | число строк в имитируемом стеке
.
. используемые исполнители:
.   Простейшая файловая система (ФС)
конец описаний |-----------------------------

программа начать работу == ФС.создать файл ⟨вых : ф : ключ⟩;
                                                  n := 0

программа пуст          : да/нет == (n = 0)
```

```
программа число строк: Z+      == n
программа добавить строку (вх : С : строка) ==
        ФС.добавить строку в конец файла (вх/вых : ф, вх : С);
                                                n := n + 1
программа взять строку в (вых : С : строка)
. дано      : n > 0
. получить:
. --------
. ФС.взять строку из конца файла (вх/вых : ф, вых : С : строка)
. n := n - 1
конец программы

программа кончить работу (вых : фв : ключ) == фв := ф
конец исполнителя |==========================
```

Осталось реализовать исполнителя «Нижний стек». В отличие от верхнего стека, который является более или менее обычным, нижний стек в момент начала работы должен содержать все строки исходного файла. Конечно, можно было бы прочесть их все от конца к началу и добавить в таком инвертированном виде в новый файл, после чего имитировать нижний стек на новом файле точно так же, как имитировали верхний. Однако предварительное инвертирование файла — длительная операция, а редактирование должно начинаться мгновенно. (Человек психологически готов к тому, чтобы нормальное завершение редактирования занимало какое-то время, но абсолютно не готов к этому в начале — когда он еще ничего не сделал.)

Поэтому поступим по-другому — проимитируем нижний стек так, как будто он в начальный момент содержит весь исходный файл, *но физически никаких строк никуда копировать не будем.* Для этого кроме исходного файла, который будем только последовательно читать, используем еще один файл, на котором проимитируем вспомогательный стек строк. Основная идея реализации заключается в том, что содержимое нижнего стека в любой момент времени представляется в виде содержимого вспомогательного стека, за которым следует непрочитанная часть исходного файла и, если она еще не взята, холостая строка (рис. 18.1):

```
нижний стек = вспомогательный стек +
                    непрочитанная часть исходного файла
                    (+, быть может, холостая строка)
```

При добавлении строки в нижний стек она всегда добавляется во вспомогательный стек. При взятии строки производится анализ: если вспомогательный стек не пуст, то строка берется из него, если же пуст, то читается очередная строка из исходного файла. Таким

образом, в начальный момент времени достаточно сделать вспомогательный стек пустым и встать в начало исходного файла.

Наконец, поскольку нижний стек должен в начальный момент содержать не только строки исходного файла, но и еще холостую строку «[* конец текста *]», будем имитировать эту строку с помощью глобального объекта

холостая строка взята : да/нет.

В начале работы установим «холостая строка взята := нет» в знак того, что за непрочитанной частью исходного файла идет

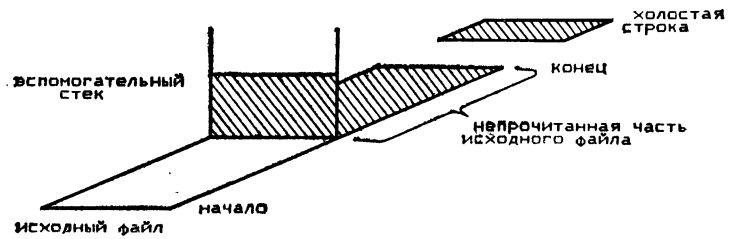


Рис. 18.1

холостая строка, а в предписании «взять» в случае, если вспомогательный стек пуст и исходный файл полностью прочитан, выдадим наружу холостую строку и установим «холостая строка взята := да».

```
исполнитель Нижний стек
. СП:
.   1. начать работу ⟨вх : ф : ключ⟩
.   2. пуст    : да/нет
.   3. добавить строку ⟨вх : С : строка⟩
.   4. взять строку в ⟨вых : С : строка⟩
.   5. кончить работу ⟨вых : ф : ключ⟩
.
. объекты:
.   фи : ключ | исходного файла
.   фв : ключ | файла для вспомогательного стека
.   холостая строка взята : да/нет
.
. используемые исполнители:
.  Простейшая файловая система (ФС)
.
конец описаний |----------------------------------
```

```
программа начать работу (вх:ф:ключ)
. дано      :
. получить:
. --------
.  фи := ф
.  ФС.встать в начало файла (вх/вых:фи)
.  ФС.создать файл (вых:фв)
.  холостая строка взята := нет
конец программы

программа пуст:да/нет == (ФС.файл (фв) пуст и
                ФС.в конце файла (фи) и холостая строка взята)
программа добавить строку (вх:С:строка) ==
     ФС.добавить строку в конец файла (вх/вых:фв,вх:С:строка)

программа взять строку в (вых:С:строка)
. дано      : не пуст
. получить:
. --------
.  выбор
.  .  при ФС.файл (фв) не пуст =>
.  .    ФС.взять строку из конца файла (вх/вых:фв,вых:С)
.  .  при ФС.не в конце файла (фи) =>
.  .    ФС.прочесть строку вперед (вх/вых:фи,вых:С:строка)
.  .  при холостая строка не взята =>
.  .    С.длина                 := 18 | число симв.холостой строки
.  .    С.текст (1..С.длина) := ''[* конец текста *]''
.  .    холостая строка взята := да
.  .  иначе => отказ
.  конец выбора
конец программы

программа кончить работу (вых:ф:ключ) ==
                              ФС.удалить файл (вх:фв); ф := фи
конец исполнителя |==========================
```

Мы реализовали исполнителя «Текст строк с изображаемым окном и указателем» с использованием реализованного нами в предыдущем разделе исполнителя «Простейшая файловая система». Иерархия структур данных в этой реализации оказалась достаточно простой (рис. 18.2).

Стрелки, идущие вниз от исполнителя, показывают, на базе каких других исполнителей он реализован. По сути дела, эта картинка является технологической цепочкой программирования «сверху

вниз» для проекта «Текст строк с изображаемым окном и указателем», нарисованной без использования «векторных обозначений».

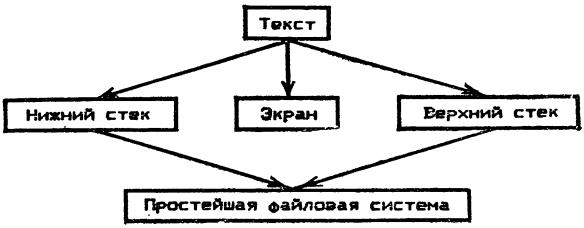


Рис. 18.2

Реально используемые файловые системы, однако, как правило, не позволяют прямо реализовать стек строк, и иерархия промежуточных исполнителей усложняется: верхний стек строк приходится реализовывать на базе блока в памяти и стека блоков на диске, а стек блоков — на базе динамического вектора блоков, который

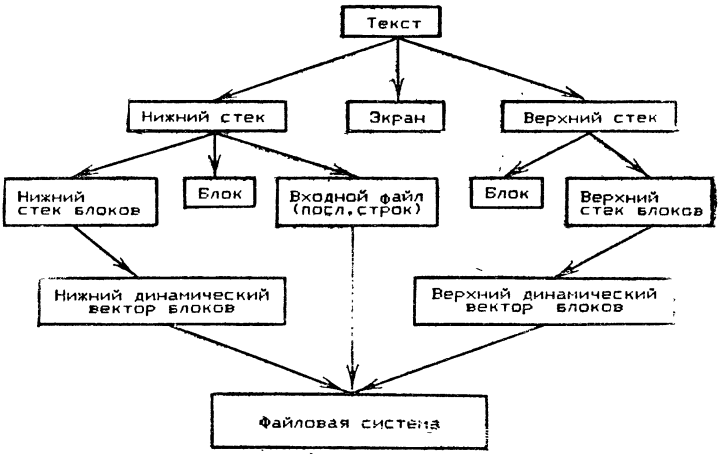


Рис. 18.3

обычно уже реализуется над файловой системой. Аналогично декомпозируется и верхний стек. В итоге иерархия структур данных принимает вид, изображенный на рис. 18.3.

ЗАДАЧИ И УПРАЖНЕНИЯ

1. Может ли при редактировании текста в некоторый момент оказаться, что в нем 100 строк, а на экране изображена только холостая строка?

2. Модифицируйте проект «Текст строк с изображаемым окном и указателем», добавив предписание «внести изменения в файл». Смысл этого предписания состоит в том, что аварийное завершение работы должно возвращать ключ файла с тем текстом, который был в момент последнего вызова предписания «внести изменения в файл».

3. Модифицируйте проект «Текст строк с изображаемым окном и указателем», считая, что и в начальный, и в конечный момент времени текст расположен в двух стеках (двух файлах) на диске и что аварийного завершения работы нет. При этом положение в тексте и в окне должно сохраняться между «кончить» и «начать» работу.

Указание. Введите предписания «добавить число в конец файла ⟨**вх/вых** : Ф, **вх** : п⟩» и «взять число из конца файла ⟨**вх/вых** : Ф, **вых** : п⟩», которые будут преобразовывать число в строку символов и наоборот, а также добавлять/брать соответствующую строку файла.

4. Модифицируйте проект «Текст строк с изображаемым окном и указателем», добавив предписание «откатить последнее изменение». По этому предписанию текст и положение указателя в нем должны принимать те состояния, в которых они находились перед последним удалением или вставкой строки. Вызов этого предписания до первого изменения текста не должен вызывать никаких действий.

5. Решите задачу 4, считая, что предписание «откатить последнее изменение» можно вызывать неоднократно: второй вызов откатывает второе по давности изменение и т. д. Другими словами, последовательное применение откатки должно восстанавливать предыдущие состояния текста вплоть до исходного состояния перед редактированием.

Указание. Используйте еще один вспомогательный файл (файл откатки) в качестве стека для хранения необходимой для откатки информации.

6. В рамках задачи 5 добавьте еще предписание «откатить откатку», которое должно выполнять действия, обратные к «откатить последнее изменение».

Указание. Используйте два вспомогательных файла.

ГЛАВА 4

ЛОГИЧЕСКОЕ УСТРОЙСТВО И ПРИНЦИПЫ РАБОТЫ ЭПВМ

До сих пор метафора Универсального Выполнителя использовалась для объяснения семантики тех или иных форм записи программ. В этой главе мы рассмотрим устройство реальных Универсальных Выполнителей, в роли которых на практике обычно выступают электронно-программные выполняющие машины (ЭПВМ). Как и следует из названия, ЭПВМ состоит из двух основных частей: электронной (ЭВМ) и программной.

Логическому устройству и принципам работы ЭВМ на примере ЭВМ серии PDP-11 фирмы DEC посвящен разд. 19 (аналогично устроены и многие другие ЭВМ, например отечественные СМ-3, СМ-4, СМ-1420, ДВК, БК, УКНЦ, Электроника-60, Электроника-100/25, Электроника-79, Электроника-85). ЭВМ сама по себе также является Универсальным Выполнителем программ, записанных на специальном языке (их называют *программами в кодах ЭВМ*). Именно на примере этих программ мы объясним, как ЭВМ «читает» программу и как обеспечивается автоматизм ее работы. Наши объяснения, впрочем, будут носить чисто логический характер и не будут касаться не менее интересных вопросов о том, из каких электронных компонент (сопротивлений, транзисторов, микросхем и т. п.) и как именно «спаяна» ЭВМ.

Вопросам о том, наличие каких программ и как именно превращает ЭВМ в ЭПВМ, т. е. в Универсального Выполнителя программ, записанных на том или ином языке программирования, а также описанию работы программиста на ЭВМ посвящен разд. 20.

19. Логическое устройство и принципы работы ЭВМ

С логической точки зрения ЭВМ состоит из нескольких согласованно работающих частей (компонент). Среди них можно выделить:

— *процессор* — устройство, которое, собственно, и производит все операции, своего рода «мозг» ЭВМ;

— *оперативную память* (или просто *память*) — устройство, предназначенное для хранения разнообразной информации. В качестве модели этого устройства раньше выступала «доска» УВ, на

которой писались программы, рисовались прямоугольники объектов, программ и исполнителей;

— *внешние устройства.* К ним относятся, по-видимому, хорошо Вам знакомые клавиатура и экран терминала, устройства для хранения информации на магнитных дисках (см. разд. 17) и другие стандартные (связанные с хранением и переработкой только информации) и нестандартные устройства. В число последних входят, например, подключаемые к ЭВМ станки с ЧПУ (скажем, Резчик металла), разнообразные датчики, устройства управления физическими объектами и т. п.



Рис. 19.1. Общая схема ЭВМ архитектуры PDP-11

Отдельные компоненты ЭВМ взаимодействуют между собой посредством специального устройства, называемого *магистралью* (рис. 19.1).

Память. Память ЭВМ состоит из элементов, называемых *битами.* Каждый бит может находиться в двух различных состояниях, которые принято обозначать цифрами 0 и 1. Перевод бита в состояние 0 принято называть *очисткой* бита, а в состояние 1— *установкой* бита. С помощью одного бита можно представить в памяти объект типа да/нет — достаточно выбрать какое-то соответствие, например считать, что 1 означает да, а 0 — нет, и переформулировать все операции над объектом типа да/нет в терминах бита и его состояния.

Бит, однако, является слишком мелкой единицей для представления объектов других типов. Поэтому биты объединяют в группы по 8 штук в каждой. Такая группа из 8 бит называется *байтом.* Байт может находиться в $2^8 = 256$ различных состояниях, которые принято обозначать числами от 0 до 255. Если обозначить биты в байте индексами i_7, i_6, i_5, i_4, i_3, i_2, i_1, i_0, то состояние x байта связывается с состояниями i_j отдельных битов формулой

$$x = i_7 2^7 + i_6 2^6 + i_5 2^5 + i_4 2^4 + i_3 2^3 + i_2 2^2 + i_1 2 + i_0.$$

Другими словами $i_7 i_6 i_5 i_4 i_3 i_2 i_1 i_0$ есть не что иное, как представление числа x в двоичной системе счисления. Например, набор состояний битов 10010010 соответствует десятичному числу 146, а набор 00100100 — десятичному числу 36. При работе с битами, однако, удобнее использовать *восьмеричную систему счисления,* в

которой состояние каждой тройки битов записывается одним числом k от 0 до 7:

k	биты	k	биты	k	биты	k	биты
0	000	1	001	2	010	3	011
4	100	5	101	6	110	7	111

Состояние битов 10010010 при этом записывается в виде восьмеричного числа 222, а состояние 00100101 — в виде восьмеричного числа 045.

Поскольку байт имеет 256 различных состояний, то с помощью одного байта можно представить объект типа «символ» — достаточно установить соглашение о том, какое состояние байта какой символ означает. Состояние байта (десятичное число от 0 до 255 или восьмеричное число от 0 до 377) при этом называется *кодом символа*. Само соглашение задается в виде *таблицы кодов*, где для каждого кода указывается соответствующий этому коду символ. Фрагмент такой таблицы с восьмеричными кодами приведен ниже:

Код	Символ	Код	Символ	Код	Символ	Код	Символ
040	пробел	060	0	100	@	120	P
041	!	061	1	101	A	121	Q
042	"	062	2	102	B	122	R
043	#	063	3	103	C	123	S
044	$	064	4	104	D	124	T
045	%	065	5	105	E	125	U
046	&	066	6	106	F	126	V
047	'	067	7	107	G	127	W
050	(	070	8	110	H	130	X
051	)	071	9	111	I	131	Y
052	*	072	:	112	J	132	Z
053	+	073	;	113	K	133	[
054	,	074	<	114	L	134	\
055	-	075	=	115	M	135	]
056	.	076	>	116	N	136	^
057	/	077	?	117	O	137	—

Память в целом является вектором байт с индексом от 0 до некоторого максимального значения. Индекс байта в памяти называется его *адресом*. Адреса мы будем записывать в восьмеричной системе счисления.

Для представления чисел, однако, байт тоже маловат. Поэтому байты объединяют в пары: нулевой с первым, второй с третьим,

четвертый с пятым и т. д. Такая пара байт с соседними адресами, из которых меньший адрес четный, называется *словом*. Сам меньший (четный) адрес называется *адресом слова*. Память, таким образом, можно представлять в виде вектора слов с четными индексами (рис. 19.2).

Так как слово состоит из двух байт (16 бит), то оно может принимать $256^2 = 2^{16} = 65\,536$ различных состояний. Биты в слове

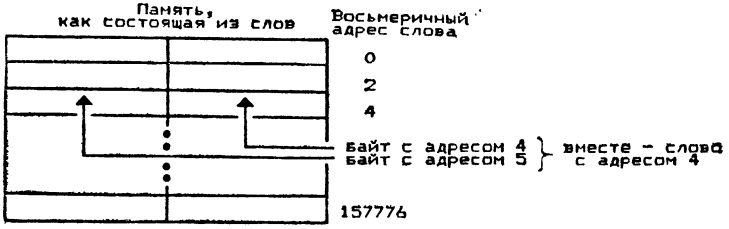


Рис. 19.2

нумеруются с нуля и справа налево, т. е. слово представляется последовательностью из 16 бит

$$i_{15}i_{14}i_{13}i_{12}i_{11}i_{10}i_9i_8i_7i_6i_5i_4i_3i_2i_1i_0.$$

С помощью таких последовательностей (т. е. одним словом) принято представлять неотрицательные целые числа (Z+) в диапазоне 0 .. 65 535 ($x = i_{15}2^{15} + i_{14}2^{14} + \ldots + i_0$), целые числа со знаком (Z) — в диапазоне −32 768 .. 32 767 ($x = i_{14}2^{14} + i_{13}2^{13} + \ldots + i_0 - i_{15}2^{15}$) либо другие объекты, число различных состояний которых не превосходит 65 536. Обратите внимание, что представления целых чисел Z+ и Z — это *два разных представления кольца вычетов по модулю* 65 536. Код результата сложения двух слов в этом кольце вычетов определяется однозначно по кодам слов и не зависит от того, выполняется ли сложение неотрицательных чисел (Z+) или чисел со знаком (Z). Обратите также внимание, что знак целого числа определяется состоянием старшего бита. Если $i_{15} = 0$, то число больше либо равно 0, а если $i_{15} = 1$ — то меньше 0.

Замечание. Битом, байтом, словом называют не только элементы памяти, но и (даже чаще) информацию, которую они содержат. Так, *бит* — это либо 0, либо 1; *байт* — последовательность из 8 нулей и единиц; *слово* — последовательность из 16 нулей и единиц. В результате словом, например, называется и элемент памяти ЭВМ, и состояние этого элемента (конкретная последовательность из 16 нулей и единиц), и любая другая конкретная последовательность из 16 нулей и единиц, и вообще «объем»

порции информации из 16 нулей и единиц (16 бит). Обычно это не приводит ни к каким недоразумениям и мы также будем использовать термин «слово» во всех этих смыслах, например будем говорить «процессор читает слово с адресом x из памяти» наряду с «процессор читает содержимое (состояние) слова с адресом x из памяти».

Слово является основной единицей хранения и обработки информации — представляемый с помощью слова диапазон неотрицательных целых чисел и целых чисел со знаком в большинстве случаев оказывается достаточным; с помощью слова (в кодировке Z+) представляются адреса байт и слов памяти; код каждой команды процессора занимает ровно одно слово; при обмене информацией через магистраль (например, между процессором и памятью) единицей обмена также является слово.

Если для представления каких-то объектов слова не хватает, то используют несколько слов (или несколько байт). Например, действительные числа (R) кодируются в двух словах. Пара слов может принимать $2^{32} = 65\,536^2$ различных состояний. Таким образом, парой слов можно закодировать всего около 4 млрд различных действительных чисел. Этого, однако, оказывается достаточно, чтобы любое число в диапазоне $(-10^{38}, -10^{-38}) \cup \{0\} \cup (10^{-38}, 10^{38})$ представить приближенно с точностью до 7 значащих цифр.

Процессор. Процессор — это «мозг» ЭВМ, который выполняет все команды. Он имеет небольшую собственную внутреннюю память, называемую *регистрами процессора*. Каждый регистр состоит из 16 бит, поэтому содержимое регистра называется словом. Поскольку регистров мало и они расположены прямо внутри процессора, то работа с ними происходит существенно быстрее, чем со словами оперативной памяти. Среди регистров процессора нас будут интересовать восемь так называемых общих регистров R0, R1, ..., R7, а также 4 младших бита в еще одном регистре PS — Processor Status, т. е. регистр состояния процессора (рис. 19.3).

Регистры R6 и R7 играют особую роль и поэтому имеют специальные обозначения SP (Stack Pointer — *указатель стека*) и PC (Program Counter — *программный счетчик* или, как его еще часто называют, *счетчик команд*). Четыре младших бита в регистре PC обозначаются буквами N, Z, V, C от слов Negative (отрицательный), Zero (нуль), oVerflow (переполнение) и Carry (перенос). Эти биты устанавливаются в 0 или в 1 при выполнении большинства команд следующим образом:

— если результат выполнения команды меньше нуля, то бит N устанавливается в 1 (в противном случае чистится);

— если результат равен 0, то бит Z устанавливается в 1 (в противном случае чистится);

— если при выполнении команды возникло переполнение, т. е. результат команды выходит за представимый диапазон целых чисел (например, при сложении двух положительных чисел получилось отрицательное), то бит V устанавливается в 1 (в противном случае чистится);

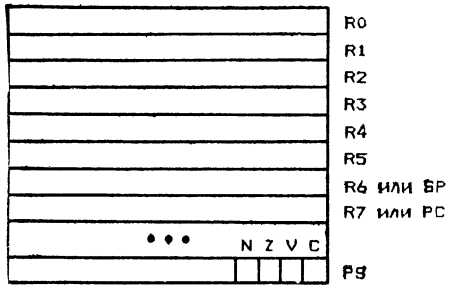


Рис. 19.3. Некоторые регистры процессора

— если при выполнении команды над словами (16 бит) возник перенос в 17-й бит, или при выполнении команды над байтами (8 бит) возник перенос в 9-й, то бит C устанавливается в 1, в противном случае чистится.

Работа процессора. В общих чертах работу процессора можно описать следующим основным циклом:

цикл выполнять
. прочесть слово памяти с адресом ⟨**вх**: PC⟩ в ⟨**вых**: x⟩
. PC.**увеличить на** 2
. выполнить команду с кодом ⟨**вх**: x⟩
конец цикла

Другими словами, процессор последовательно читает из памяти команды программы (каждая команда занимает одно слово) и выполняет их. Очередная команда программы всякий раз читается из слова памяти, адрес которого в данный момент находится в регистре PC. В процессе выполнения команды процессор может читать еще какие-то слова из памяти, записывать в память результат, опрашивать состояние каких-то внешних устройств или посылать им какие-то команды.

Обратите внимание, что при чтении команды регистр PC сразу же увеличивается на 2. Поэтому при следующем выполнении тела цикла процессор получит и выполнит следующую команду, потом

еще одну и т. д. Именно увеличение PC обеспечивает автоматизм работы ЭВМ как Универсального Выполнителя.

Пусть, например, в начальный момент PC=1000, а в памяти, начиная с адреса 1000, расположена следующая программа в кодах (адреса и содержимое памяти изображены в восьмеричном виде):

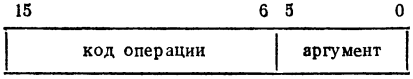
Память	Адрес	Что делает процессор	Мнемоника
010002	1000	R 2:= R0	MOV R0, R2
060102	1002	R2.увеличить на R1	ADD R1, R2
000000	1004	стоп	HALT

Первая команда здесь заставляет процессор переслать содержимое регистра R0 в регистр R2, вторая добавляет к содержимому регистра R2 содержимое регистра R1, а третья останавливает процессор (работа вообще прекращается). Таким образом, после выполнения этой программы в регистре R2 окажется сумма содержимого регистров R0 и R1, а в регистре PC — 1006 (адрес команды HALT + 2). В дальнейшем мы будем изображать содержимое памяти только в мнемоническом виде:

MOV	R0	R1	1000	R2 := R0
ADD	R1	R2	1002	R2.увеличить на R1
HALT			1004	стоп

Команды процессора. Разберемся теперь более детально с тем, какие бывают команды. Каждая команда процессора занимает ровно одно слово (16 бит). Эти 16 бит можно разбить на несколько групп. Одна из групп задает *код операции* (что именно должен сделать процессор), другие группы задают аргументы команды. Количество групп и распределение битов между ними для разных команд могут быть различными. Мы условно разделим команды на 5 классов: одноаргументные, двухаргументные, команды работы с битами NZVC, переходы, все остальные.

Одноаргументные команды. Структура этих команд такова:



Биты с 15-го по 6-й содержат код операции, а биты с 5-го по 0-й задают аргумент команды. В коде операции самый левый

(старший) бит обычно указывает, относится ли команда к слову ($i_{15} = 0$) или к байту ($i_{15} = 1$). Например, команда CLR (очистить слово, т. е. установить все биты в 0) имеет восьмеричный код операции 0050 (биты 0000101000) и в целом может быть представлена в виде .050DD, где символ «.» обозначает старший бит (0 для команды «очистить слово», 1 для команды «очистить байт»), а буквы DD (от слова Destination — получатель) обозначают две восьмеричные цифры, задающие аргумент команды. Как именно задается аргумент, мы разберем позже. Сейчас важно лишь, что для его задания используются 6 бит. Например, DD = 01 означает, что аргументом является содержимое регистра R1 процессора. Таким образом, команда «очистить слово — содержимое регистра R1» будет иметь восьмеричный код 005001 (биты 0000101000000001).

Перечень одноаргументных команд приведен в табл. 19.1.

Таблица 19.1

Код	Команда	Действие	NZVC
.050DD	CLR (B) — clear	d := 0	0 1 0 0
.051DD	COM (B) — complement	d := ~d	↕ . 0 1
.052DD	INC (B) — increment	d := d + 1	↕ ↕ ↕ –
.053DD	DEC (B) — decrement	d := d – 1	↕ ↕ ↕ –
.054DD	NEG (B) — negate	d := –d	↕ ↕ ↕ ↕
.055DD	ADC (B) — add carry	d := d + C	↕ ↕ ↕ ↕
.056DD	SBC (B) — subtract carry	d := d – C	↕ ↕ ↕ ↕
.057DD	TST (B) — test	d := d	↕ ↕ 0 0
.060DD	ROR (B) — rotate right	→C, d	↕ ↕ ↕ ↕
.061DD	ROL (B) — rotate left	C, d←	↕ ↕ ↕ ↕
.062DD	ASR (B) — arithmet. shift right	d := d/2	↕ ↕ ↕ ↕
.063DD	ASL (B) — arithmet. shift left	d := 2*d	↕ ↕ ↕ ↕
0003DD	SWAB — swap bytes	d := sb (d)	↕ ↕ ↕ 0

В табл. 19.1:

d — аргумент команды (например, содержимое R1, если DD = 01);

~ — операция изменения состояний всех битов на противоположные (1 на 0, 0 на 1);

→C,d — циклический сдвиг бита C процессора и всех битов слова или байта d на одну позицию вправо (рис. 19.4, *а*);

C,d← — циклический сдвиг бита C процессора и всех битов слова или байта d на одну позицию влево (рис. 19.4, *б*);

sb(d) — слово, полученное из d перестановкой байтов.

В колонке битов NZVC:

0 — в результате выполнения команды бит чистится;

1 — в результате выполнения команды бит устанавливается в состояние 1;

. — в результате выполнения команды бит устанавливается в 1 или 0 в зависимости от результата и хода выполнения команды (как описано на с. 304—305);

— — состояние бита в результате выполнения команды не изменяется.

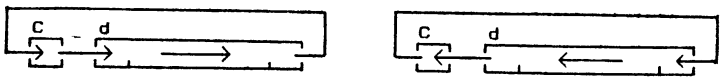
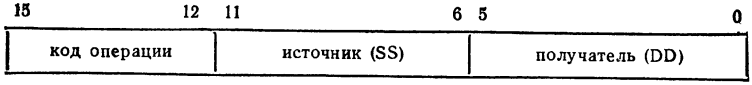


Рис. 19.4

Двухаргументные команды. В двухаргументных командах аргументы принято называть *источником* и *получателем*:



В табл. 19.2 двухаргументных команд они обозначены соответственно s (source) и d (destination)

Таблица 19.2

Код	Команда	Действие	N Z V C
.1SSDD	MOV (B) — move	d := s	. . 0 —
.2SSDD	CMP (B) — compare	:= s — d	. . . .
.3SSDD	BIT (B) — bit test	:= s & d	. . 0 —
.4SSDD	BIC (B) — bit clear	d := d & ~s	. . 0 —
.5SSDD	BIS (B) — bit set	d := d ! s	. . 0 —
06SSDD	ADD — add	d := d + s	. . . .
16SSDD	SUB — subtract	d := d — s	. . . .

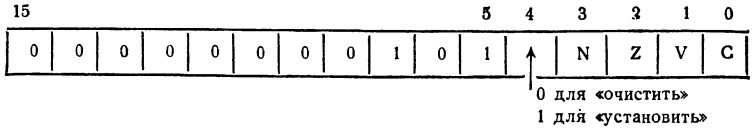
В табл. 19.2:

& — операция «побитное и», т. е. $(s \& d)_i = 1 \Leftrightarrow s_i = 1$ и $d_i = 1$ (s_i, d_i, $(s \& d)_i$ — состояния бита с номером i в s, d, s & d соответственно);

! — операция «побитное или»: $(s ! d)_i = 1 \Leftrightarrow s_i = 1$ или $d_i = 1$

Если при описании действия слева от знака присваивания ничего не написано (команды CMP, CMPB, BIT, BITB), то это значит, что результат вычисления правой части сказывается лишь на установке битов NZVC, а аргумент-получатель не изменяется.

Команды работы с битами NZVC. Эти безаргументные команды позволяют явно очистить/установить те или иные из битов NZVC процессора. Их общая структура такова:



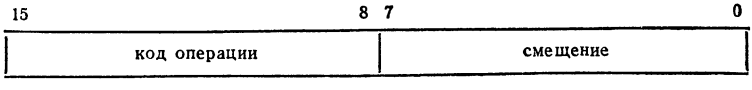
По сути дела, все эти команды работают одинаково, а именно копируют состояние 4 бита кода команды в те из битов NZVC процессора, для которых соответствующие биты в коде команды установлены в 1 (табл. 19.3).

Таблица 19.3

Код	Команда	NZVC
000241	CLC — clear C	---0
000242	CLV — clear V	--0-
000244	CLZ — clear Z	-0--
000250	CLN — clear N	0---
000257	CCC — clear NZVC	0000
000261	SEC — set C	---1
000262	SEV — set V	--1-
000264	SEZ — set Z	-1--
000270	SEN — set N	1---
000277	SCC — set NZVC	1111

(CC в командах CCC и SCC — это аббревиатура слов Condition Code bits — биты кодов условий.)

Команды переходов. Единственный результат выполнения этих команд — изменение (или неизменение) содержимого регистра PC в зависимости от выполнения (или невыполнения) некоторых условий. Формально все условия в этих командах — это некоторые утверждения о состоянии битов NZVC процессора. Содержательно, однако, большинство этих условий имеет смысл утверждения о результате выполнения предыдущей команды. Все команды (табл. 19.4) имеют вид



Младший байт команды (биты с 7 по 0) трактуется как целое число в диапазоне от —128 до 127 ($x = i_6 2^6 + i_5 2^5 + \ldots + i_0 - i_7 2^7$), называемое *смещением*. Если условие, определяемое кодом опера-

ции, выполнено, то результат выполнения команды перехода можно записать в виде

PC.**увеличить на** (2 * смещение)

Поскольку в момент выполнения команды перехода регистр PC содержит адрес команды перехода + 2, то результат перехода можно записать также в виде

PC := адрес команды перехода + 2 + (2 * смещение)

Если же условие, определяемое кодом операции, не выполнено, то PC не изменяется (остается равным адресу команды перехода + 2). Таким образом, в этом случае очередная команда будет получена из слова памяти, следующего за командой перехода.

Таблица 19.4

Код	Команда	Условие перехода	
		Неформально	Формально
000400	BR —branch (br)	всегда	
001000	BNE —br if not equal	≠0	Z=0
001400	BEQ —br if equal	=0	Z=1
100000	BPL —br if plus	+	N=0
100400	BMI —br if minus	−	N=1
102000	BVC —br if V is clear		V=0
102400	BVS —br if V is set		V=1
103000	BCC —br if C is clear		C=0
103400	BCS —br if C is set		C=1
002000	BGE —br if greater or equal	≥0	N#V=0
002400	BLT —br if less than	<0	N#V=1
003000	BGT —br if greater than	>0	Z! (N#V)=0
003400	BLE —br if less or equal	≤0	Z! (N#V)=1
101000	BHI —br if higher	>	Z! C=0
101400	BLOS—br if lower or same	≤	Z! C=1
103000	BHIS—br if higher or same	≥	C=0
103400	BLO —br if lower	<	C=1

В табл. 19.4:

— операция «исключающее или» (N # V = 1 ⇔ (N = 1 и V = 0) или (N = 0 и V = 1), в остальных случаях N # V = 0);

— в колонке «Код» приведен код команды с нулевым смещением;

— колонки битов NZVC не приведены, так как команды переходов их не изменяют.

С помощью команд перехода можно заставить процессор выполнить ту или иную группу команд в зависимости от некоторых

условий, повторить некоторую группу команд и т. п., т. е. смоделировать конструкции ветвления и повторения. Соответствующие примеры программ в кодах приведены ниже.

Команды BGE, BLT, BGT, BLE применяются для анализа результатов операций над целыми числами со знаком (Z), а команды BHIS, BLO, BHI, BLOS — для анализа результатов операций над неотрицательными целыми (Z+).

Другие команды. Приведенный выше перечень команд процессора не является полным. Среди других команд можно отметить следующие (табл. 19.5). Команда JMP — это еще одна команда

Таблица 19.5

Код	Команда	Действие
0001DD	JMP —jump	PC := d
000240	NOP —no operation	ничего
000000	HALT—halt	стоп

перехода, которая просто помещает в PC свой аргумент. Выполнение команды NOP не приводит ни к каким действиям процессора. По команде HALT процессор останавливается (прекращает выполнение программы). Некоторые другие команды (вызов подпрограммы, возврат из подпрограммы, возврат из прерывания) будут введены и объяснены ниже.

Задание аргументов команд. Аргументы команд задаются в коде команды с помощью шести бит или, что то же самое, двух восьмеричных цифр. Вторая из этих двух цифр является номером одного из общих регистров процессора (от 0 до 7), а первая (также от 0 до 7) называется *видом адресации* и описывает, как именно, используя содержимое соответствующего регистра, получить аргумент команды. Всего, таким образом, существует восемь различных способов получения аргумента команды (табл. 19.6).

Рассмотрим, например, выполнение процессором команды

MOV	(PC)+	R0	1000	R0 :=
		3000	1002	3000

(код команды — 012700, SS = 27, DD = 00).

В соответствии с основным циклом работы процессора к моменту, когда процессор прочтет эту команду из памяти и начнет выполнять ее, PC = 1002. Команда MOV — это двухаргументная команда пересылки слова из аргумента-источника в аргумент-получатель. Сначала процессор получит аргумент-источник. Так как

последний задан в виде (PC)+(вид адресации 2, регистр PC), то в качестве источника (s) будет взято содержимое слова по адресу,

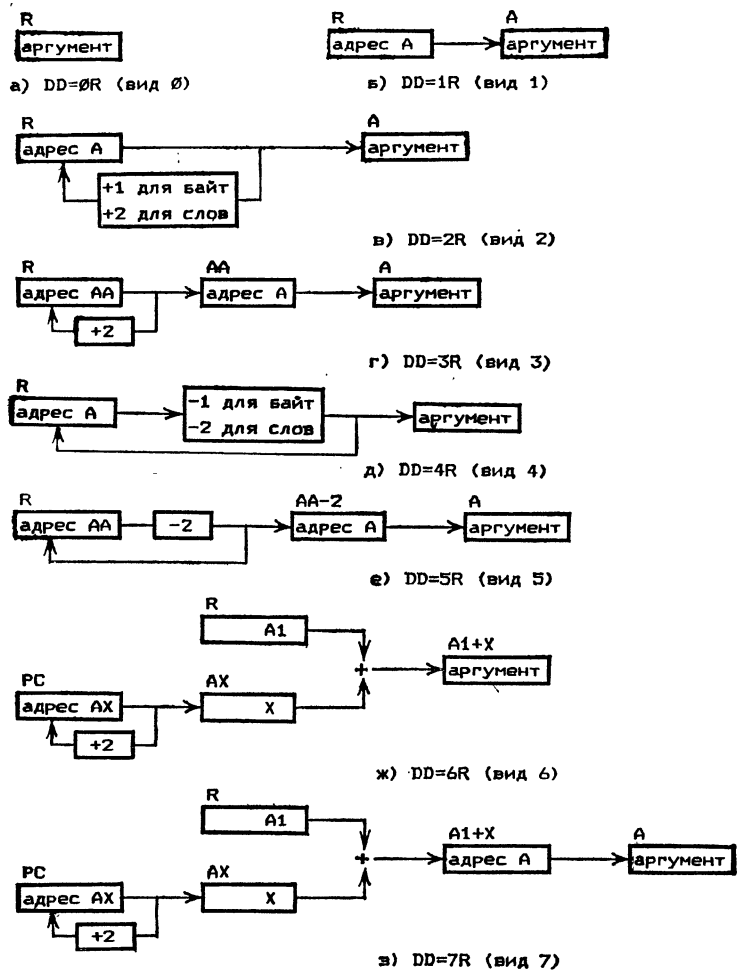


Рис. 19.5. Виды адресации аргументов команд

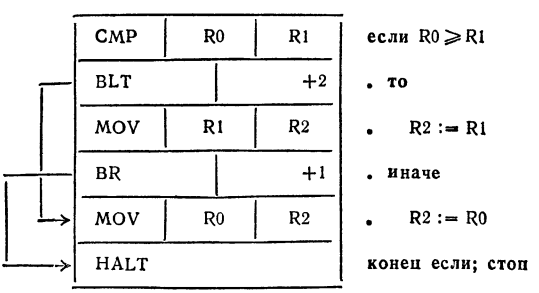
находящемуся в PC, а сам PC будет увеличен на 2. В данном случае PC = 1002, s = 3000, следовательно, PC будет увеличен до 1004. Получатель (d) команды задан в виде R0 (вид адресации 0, регистр R0), т. е. d — это содержимое регистра R0. Поэтому в результате выполнения команды MOV (d := s) в регистр R0 будет

Таблица 19.6

Вид	Мнемоника	Описание (R) обозначает содержимое регистра R)
0	R	(R)—аргумент (рис. 19.5, *а*)
1	(R)	(R)—адрес аргумента (рис. 19.5, *б*)
2	(R)+	(R)—адрес аргумента; (R) увеличивается на 1 (если аргумент—байт) или на 2 (если аргумент—слово) (рис. 19.5, *в*)
3	@(R)+	(R)—адрес адреса аргумента: (R) увеличивается на 2 (рис. 19.5, *г*)
4	—(R)	(R) уменьшается на 1 (если аргумент—байт) или на 2 (если аргумент—слово); новое (R)—адрес аргумента (рис. 19.5, *д*)
5	@—(R)	(R) уменьшается на 2; новое (R)—адрес адреса аргумента (рис. 19.5, *е*)
6	X (R)	(R)+X—адрес аргумента, где X—содержимое слова с адресом PC; PC увеличивается на 2 (рис. 19.5, *ж*)
7	@X (R)	(R)+X—адрес адреса аргумента, где X—содержимое слова с адресом PC; PC увеличивается на 2 (рис. 19.5, *з*)

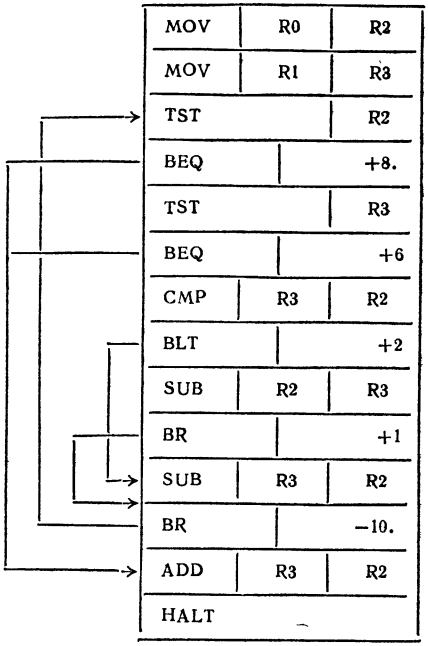
записано число 3000. Обратите внимание, что после выполнения команды PC = 1004 и следующей будет выполняться команда, расположенная в слове с адресом 1004.

Примеры программ. Пример 1, R2 := min(R0, R1):



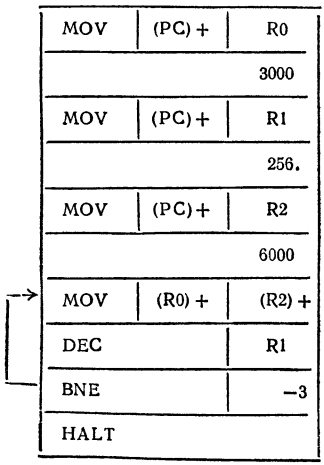
Пример 2. R2 := НОД(R0, R1), считая, что R0 ⩾ 0, R1 ⩾ 0, R0 + R1 > 0 (точка после числа означает, что число записано в

десятичной системе счисления):



MOV	R0	R2	R2 := R0
MOV	R1	R3	R3 := R1
TST		R2	цикл пока R2≠0
BEQ		+8.	. и
TST		R3	. R3≠0
BEQ		+6	. выполнять
CMP	R3	R2	. если R3≥R2
BLT		+2	. . то
SUB	R2	R3	. . R3 := R3—R2
BR		+1	. . иначе
SUB	R3	R2	. . R2 := R2—R3
			конец если
BR		−10.	конец цикла
ADD	R3	R2	R2 := R2 + R3
HALT			стоп

Пример 3. Пересылка 256 слов памяти, расположенных, начиная с адреса 3000 (восьмеричное), в память, начиная с адреса 6000 (восьмеричное):



MOV	(PC)+	R0	R0 :=
		3000	3000
MOV	(PC)+	R1	R1 :=
		256.	256.
MOV	(PC)+	R2	R2 :=
		6000	6000
MOV	(R0)+	(R2)+	цикл выполнять . переслать (R0)+ в (R2)+
DEC		R1	. R1 уменьшить на 1
BNE		−3	. пока R1≠0
HALT			конец цикла стоп

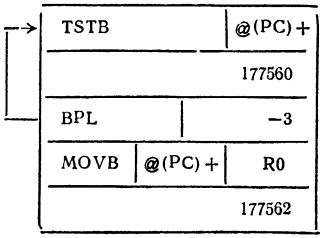
Пример 4. Сложение длинных целых чисел. Пусть необходимо работать с целыми числами, выходящими за представимый с помощью одного слова диапазон. В этом случае можно хранить число x в паре слов с адресами A+2 и A (A будем называть адресом числа x) и использовать 32 бита этой пары слов для представления x в двоичной системе счисления (старшие 16 бит — в слове A+2, младшие — в слове A). Процессор ЭВМ, однако, не умеет выполнять операции над такими *длинными целыми* числами. Поэтому сложение, например, двух длинных целых надо представлять в виде последовательности команд процессора, т. е. в виде программы. Приведем пример программы, которая добавляет длинное целое число, адрес которого записан в регистре R0, к длинному целому числу, адрес которого записан в регистре R1:

ADD	(R0)+	(R1)+	сложение младших слов чисел
ADC		(R1)	учет переноса в старшие разряды
ADD	(R0)	(R1)	сложение старших слов чисел
HALT			стоп

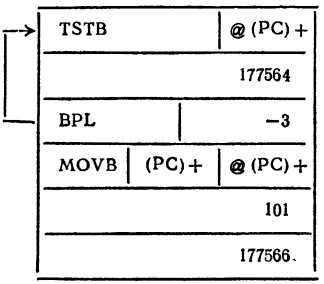
Внешние устройства, магистраль и еще раз понятие адреса. Как уже говорилось, компоненты ЭВМ — процессор, память и различные внешние устройства — взаимодействуют между собой через магистраль. «С точки зрения» магистрали каждая компонента ЭВМ — это просто какое-то количество 16-битных слов (слова процессора и внешних устройств принято называть *регистрами*). Каждое слово имеет свой уникальный адрес — число в диапазоне от 0 до 177776 (адреса здесь и далее — восьмеричные). Слова памяти имеют адреса от 0 до 157776, адреса регистров процессора и регистров внешних устройств не меньше 160000 (какие-то адреса могут быть и вовсе не задействованы). Например, регистр PS процессора имеет адрес 177776; клавиатура терминала «с точки зрения» магистрали — это два регистра с адресами 177560 и 177562; экран терминала — это также два регистра с адресами 177564 и 177566; магнитный диск типа RK (диски бывают разные) — это 7 регистров с адресами от 177400 до 177416. Магистраль как связующее устройство обеспечивает только операции чтения/записи слова или байта по некоторому адресу.

Регистры внешних устройств содержат информацию о состоянии этих устройств и могут изменяться по внешним (не зависящим от процессора) причинам. Например, при нажатии человеком на клавишу на клавиатуре терминала изменятся регистры клавиатуры:

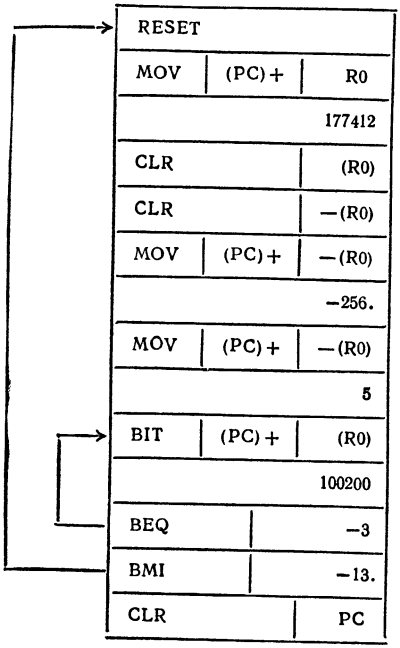
в регистре 177560 будет информация о том, что нажата клавиша (бит i_7 в состоянии 1), а в младшем байте регистра 177562 (т. е. в битах с i_7 по i_0) будет код нажатой клавиши. Таким образом, фрагмент программы, который ждет, пока человек не нажмет на какую-нибудь клавишу, а после этого помещает код нажатой клавиши в младший байт регистра R0, выглядит так:



Все команды управления внешними устройствами выглядят просто как запись определенной комбинации бит в тот или иной регистр устройства. Например, вывод символа А на экран терминала состоит в том, что процессор, проанализировав состояние регистра 177564, должен убедиться в том, что терминал готов изображать очередной символ (бит i_7 в состоянии 1), после чего записать 101 (восьмеричный код символа А) в младший байт регистра 177566:



Аналогичным образом, анализируя и меняя содержимое соответствующих регистров, процессор может работать и с диском. Пусть, например, в нулевом блоке на диске записана некоторая программа X в кодах ЭВМ. Для того чтобы выполнить программу X, ее надо поместить где-то в памяти ЭВМ (например, в самом начале, начиная с адреса 0) и записать в PC адрес первой команды программы X (т. е. 0). Эти действия, однако, может произвести и сама ЭВМ в процессе выполнения специальной программы, назовем ее *программой начальной загрузки*:



Здесь команда RESET инициализирует магистраль. Следующая группа команд устанавливает требуемые состояния регистров диска: в регистр с адресом 177412 записывается адрес блока на диске, в регистр 177410 — адрес в памяти, в регистр 177406 — число пересылаемых с диска или на диск слов. Запись в регистр 177404 числа 5 (т. е. установка битов i_2 и i_0) означает, что надо выполнить команду чтения. После этого диск сам, *без участия процессора* читает заданное в регистре 177406 число слов с диска, начиная с блока с адресом, заданным в регистре 177412, и записывает (через магистраль) эту информацию в последовательные слова памяти, начиная с адреса, заданного в регистре 177410. Таким образом, нулевой блок с диска (256 слов) будет прочитан в начало памяти ЭВМ. После окончания операции чтения, когда все требуемые слова будут записаны в память, диск установит в единицу бит i_7 (готово) в регистре R177404. Если при выполнении операции возникнут какие-нибудь ошибки, то в регистре 177404 диск установит в единицу бит i_{15} (ошибка). Таким образом, приведенная выше программа после запуска операции чтения с диска ждет окончания этой операции. В случае ошибки все повторяется с самого начала, а если нулевой блок нормально прочитан, то регистр PC очищается (устанавливается в 0). В соответствии с основным циклом работы

процессора следующей будет выполнена команда, расположенная в слове с адресом 0, т. е. первая команда прочитанной с диска программы. Затем будет выполнена вторая команда прочитанной программы и т. д.

Подпрограммы и их вызовы. Для вызова подпрограмм и продолжения выполнения программы после завершения подпрограммы служат команды процессора

JSR R,DD — jump to subroutine — переход к подпрограмме,
RTS R — return from subroutine — возврат из подпрограммы.

Хотя в этих командах R может быть любым регистром, мы рассмотрим и опишем только частный случай этих команд, когда R = PC. Для краткости будем обозначать эти команды CALL DD и RETURN соответственно. Таким образом, подпрограмма — это такая же последовательность команд процессора, как и любая программа в кодах, но в конце которой вместо команды HALT (стоп) стоит команда RETURN (возврат). Адрес первой команды подпрограммы принято называть *адресом подпрограммы*, а адрес команды, с которой надо продолжить выполнение программы, — *адресом возврата*.

Рассмотрим выполнение команды CALL DD на примере следующей программы:

CALL	@(PC)+	1000	переход к подпрограмме
	2400	1002	с адресом 2400

Пусть SP = 1000. К моменту, когда процессор прочел команду CALL и начинает ее выполнять, PC = 1002. Выполнение команды CALL состоит в следующем:

1) процессор получает адрес подпрограммы-аргумента (адрес d). Для команды CALL @(PC)+ (DD = 37) адрес подпрограммы берется из слова с адресом 1002 (содержимое PC), а PC увеличивается на 2 (до 1004);

2) текущее содержимое PC (а это адрес команды, следующей после CALL, т. е. адрес возврата) запоминается в памяти, в слове —(SP), т. е. SP уменьшается на 2 (до 776) и в слово с восьмеричным адресом 776 записывается состояние PC (1004);

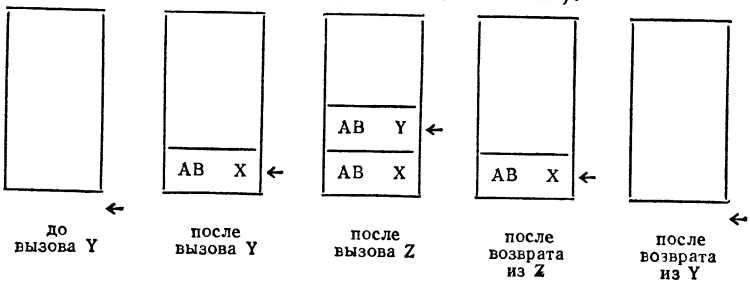
3) в PC помещается полученный ранее адрес подпрограммы (2400).

Таким образом, после выполнения этой команды PC = 2400 (адрес подпрограммы), SP = 776, в слове с адресом 776 записан адрес возврата (1004).

Команда RETURN эквивалентна команде MOV (SP)+, PC и работает так: слово памяти по адресу, записанному в SP (сейчас

SP = 776), пересылается в PC (PC станет равным 1004 — адресу возврата), а SP увеличивается на 2 (до 1000). В итоге SP станет таким же, как до команды CALL, а очередной будет выполняться команда, следующая за командой CALL.

Если программа X вызывает подпрограмму Y, которая в свою очередь вызывает Z, то адреса возвратов (сокращенно AB) будут сохраняться в памяти ЭВМ следующим образом (стрелкой указано слово памяти, адрес которого находится в SP):



Это и есть тот самый *стек приостановленных программ* (а точнее, *стек адресов возвратов*, т. е. точек приостановки), о котором шла речь в разд. 4. Регистр SP называется указателем стека (Stack Pointer) именно потому, что он в каждый момент времени содержит адрес вершины стека.

Прерывания. Есть еще один класс возникающих при работе ЭВМ ситуаций, которые обрабатываются процессором примерно так же, как вызовы подпрограмм. Это *прерывания*. Прерывания могут возникать как по внешним причинам (например, при нажатии человеком клавиш на клавиатуре терминала), так и вследствие различных внутренних исключительных ситуаций. Примерами исключительных ситуаций могут служить задание нечетного адреса в качестве адреса слова или попытка процессора обратиться по незадействованному адресу.

При возникновении прерывания процессор вызывает некоторую специальную программу, которая называется *программой обработки прерывания*, однако в отличие от вызова обычной программы сохраняет в стеке не только PC, но и PS. Тем самым при возврате из программы обработки прерывания (для этого существует специальная команда RTI — return from interrupt) восстанавливаются и PC, и PS (в частности, биты NZVC). Так как прерывания могут возникать *независимо от программы*, то адреса программ обработки прерываний и новые PS помещаются в *фиксированные места памяти* ЭВМ. Например, адрес программы обработки прерываний от клавиатуры терминала располагается в слове памяти с восьмеричным адресом 60, а состояние PS, с которым должна работать эта

программа, — в слове с адресом 62. Эту пару слов принято называть *вектором прерывания.*

Основной цикл работы процессора с учетом прерываний можно теперь записать более точно:

цикл выполнять
. **если** есть прерывание **и** процессор может его обработать
. . **то**
. . вызвать программу обработки прерывания
. . **иначе**
. . прочесть слово памяти с адресом ⟨**вх**: PC⟩ в ⟨**вых**: x⟩
. . PC.**увеличить на** 2
. . выполнить команду с кодом ⟨**вх**: x⟩
. **конец если**
конец цикла

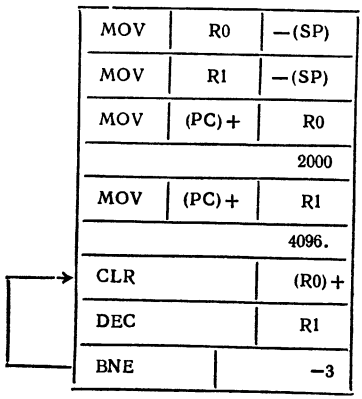
Для простоты ограничимся случаем, когда процессор может обработать прерывание, и рассмотрим следующую задачу. Пусть нужно реализовать очередь кодов нажатых человеком клавиш, т. е. чтобы коды попадали в очередь при нажатии человеком на клавиши, а брались из очереди по запросам из программы. Поскольку коды в очередь должны попадать независимо от состояния программы, то надо воспользоваться механизмом прерываний — в момент нажатия на клавишу должна вызываться программа обработки прерываний, которая добавит код нажатой клавиши в конец очереди. Предположим, что мы такую программу написали и поместили ее в память ЭВМ, начиная с адреса 2000. Для того чтобы эта программа обработки прерываний действительно вызывалась при каждом нажатии на клавишу, где-то в начале основной программы надо написать:

MOV	(PC)+	@ (PC)+	⟨слово с адресом 60⟩ := 2000 т. е. записать адрес программы обработки прерываний в первое слово вектора прерываний от клавиатуры
2000			
60			
MOV	(PC)+	@ (PC)+	⟨слово с адресом 62⟩ := 340 т. е. записать во второе слово вектора прерываний 340 — состояние PS, с которым должна работать программа обработки прерываний
340			
62			
MOV	(PC)+	@ (PC)+	⟨слово с адресом 177560⟩ := 100 т. е. установить бит i6 в регистре 177560 в состояние 1. После этого любое нажатие клавиш на клавиатуре будет вызывать прерывание
100			
177560			

Сама программа обработки прерываний должна являться реализацией предписания «добавить элемент ⟨вх: е⟩ в конец очереди» исполнителя «Очередь элементов типа код символа». Идеи такой реализации на базе части памяти ЭВМ («циклического» вектора) были изложены в разд. 12. Здесь входной параметр е — код нажатой клавиши — это просто младший байт регистра 177562 клавиатуры, глобальные объекты исполнителя — это какие-то фиксированные части памяти ЭВМ. Таким образом, программа обработки прерываний должна переслать младший байт регистра 177560 клавиатуры в определенное место в памяти ЭВМ, изменить содержимое еще каких-то слов памяти (глобальных объектов исполнителя) и закончиться командой RTI.

Последний пример. Использование особенностей устройства ЭВМ в ряде случаев позволяет создавать существенно более *эффективные* (быстрее работающие) и *компактные* (требующие меньше места в памяти) программы. Рассмотрим, например, задачу битовой реализации множества элементов типа 0..65535 на базе вектора элементов типа да/нет с индексом типа 0..65535 (идеи битовой реализации были изложены в разд. 14). Поскольку элемент типа да/нет можно представить одним битом, то для представления базового вектора понадобится 65536/16 = 4096 слов памяти.

Операции над отдельными битами можно выполнять с помощью команд BIT, BIS, BIC. Однако в массовых операциях «сделать множество пустым» и «множество пусто: да/нет» можно, используя возможности ЭВМ чистить или анализировать *не отдельные биты, а слова целиком.* Вот как, например, будет выглядеть подпрограмма «сделать множество пустым» при условии, что выделенные под вектор 4096 слов памяти начинаются со слова с восьмеричным адресом 2000:



добавить слово ⟨вх: R0⟩ в стек

добавить слово ⟨вх: R1⟩ в стек

R0 :=

2000 (восьмеричное)

R1 :=

4096. (десятичное)

цикл выполнять
- очистить слово (R0)+
- R1.уменьшить на 1
- пока R1≠0

конец цикла

MOV	(SP)+	R1	взять слово из стека в (вых : R1)
MOV	(SP)+	R0	взять слово из стека в (вых : R0)
RETURN			возврат из подпрограммы

Обратите внимание, что в результате выполнения этой подпрограммы регистры R0 и R1 не изменяются — в начале подпрограммы их состояния запоминаются в стеке (аналогично тому, как запоминаются адреса возвратов при вызовах подпрограмм), а в конце старые состояния восстанавливаются из стека.

Заключение. Хотя другие ЭВМ и программы в их кодах могут довольно существенно отличаться от рассмотренных выше (например, ЭВМ может вообще не иметь магистрали, регистров может быть меньше или больше, может быть совершенно иная система команд, основная единица информации — слово — может состоять не из 16, а, например, из 32 бит и т. д.), тем не менее все распространенные современные ЭВМ в целом похожи друг на друга. Общим является разбиение на основные компоненты (процессор, память, внешние устройства), хранение программ и данных в общей памяти, наличие счетчика команд, изменение состояния этого счетчика (команды переходов) как средство реализации управляющих конструкций, обработка вызовов программ и прерываний с помощью стека и пр. Общим и наиболее важным является также *программный принцип управления* ЭВМ, т. е. то, что любая ЭВМ — это Универсальный Выполнитель соответствующих программ. Таким образом, достаточно изменить программу — информацию в памяти ЭВМ, и та же ЭВМ начнет выполнять другие действия.

ЗАДАЧИ И УПРАЖНЕНИЯ

1. Напишите программу, которая помещает в регистр R0:

а) 0,

б) 2000,

в) число, адрес которого лежит в R1,

г) содержимое R1 с противоположным знаком,

д) содержимое R1, увеличенное на 200,

е) абсолютную величину числа, лежащего в R1,

ж) максимум из чисел, лежащих в R0, R1,

з) максимум из чисел, адреса которых лежат в R0, R1,

и) минимум из чисел, лежащих в R0, R1,

к) максимум из чисел, лежащих в R1, R2, R3,

л) сумму $2K+(2K-2)+\ldots+4+2$, где K — это содержимое R1,

м) сумму $2K+(2K-2)+\ldots+4+2$, где 2K — это содержимое R1,

н) n-й член последовательности Фибоначчи, где n — это содержимое R1.

В упражнениях 2, 3 считается, что в памяти, начиная с адреса 2000, расположены 100 целых чисел.

2. Напишите программу, после выполнения которой в регистре R0 будет содержаться:

а) сумма чисел,

б) сумма модулей чисел,

в) сумма положительных чисел,

г) минимальное число,

д) адрес первого минимального числа,

е) количество положительных чисел,

ж) количество чисел, больших 100.

3. Напишите программу, после выполнения которой:

а) все числа станут равны 0,

б) все числа увеличатся на 1,

в) все числа изменят знак на противоположный,

г) числа будут образовывать арифметическую прогрессию с начальным членом 1 и разностью —1,

д) в памяти будут записаны первые 100 чисел Фибоначчи.

В задачах 4, 5 речь идет о битовой реализации множества элементов типа 0..65535 на базе вектора элементов типа да/нет (типа бит) с индексом типа 0..65535. В качестве вектора используются 4096 слов памяти, начиная с адреса 2000.

4. Напишите подпрограмму «множество пусто : да/нет», которая вырабатывает в качестве значения типа да/нет бит C: 1, если множество пусто, и 0 — в противном случае. После завершения выполнения подпрограммы все регистры должны быть в тех же состояниях, в которых они были до ее вызова.

5*. Реализуйте подпрограммы:

а) элемент ⟨вх : e⟩ принадлежит множеству: да/нет,

б) добавить элемент ⟨вх : e⟩ к множеству.

в) удалить элемент ⟨вх : e⟩ из множества,

считая, что входной параметр e типа 0..65535 — это содержимое регистра R0, а программа «элемент принадлежит множеству» вырабатывает в качестве значения бит C. Состояния регистров в результате выполнения подпрограмм меняться не должны.

20. Работа программиста на ЭВМ

В предыдущем разделе были рассмотрены устройство и принципы работы ЭВМ. На практике, однако, ЭВМ всегда используется вместе с некоторым программным обеспечением и программист имеет дело не с ЭВМ «самой по себе», а с ЭПВМ — Универсальным Вы-

полнителем программ на том или ином языке программирования. В этом разделе мы рассмотрим, какие программы и как именно превращают ЭВМ в ЭПВМ.

Начальная загрузка. Прежде всего заметим, что в состав ЭВМ наряду с оперативной памятью (ее также называют ОЗУ — *оперативным запоминающим устройством*) обычно входит ПЗУ — *постоянное запоминающее устройство*. В отличие от оперативной памяти содержимое ПЗУ (какое-то количество слов с какими-то адресами) можно только читать. Соответствующая информация заносится в ПЗУ при его изготовлении (на заводе), сохраняется при выключении и включении ЭВМ и, как правило, не может быть изменена.

Обычно при включении ЭВМ сразу же начинает выполняться некоторая записанная в ПЗУ *программа начальной загрузки* (т. е. при включении ЭВМ в РС помещается фиксированный заранее адрес этой программы). Если ЭВМ имеет диски, то программа начальной загрузки аналогична приведенной на с. 317 — она считывает нулевой блок с диска в начало памяти и «передает на него управление» — помещает в РС адрес считанной программы, т. е. 0. После этого начинает выполняться та программа, которая была записана в нулевом блоке на диске. Заметим, что эта программа (А), в отличие от программ в ПЗУ, может быть легко изменена — достаточно записать в нулевой блок на диск новую информацию.

Загрузка операционной системы (ОС). Программа А (256 слов нулевого блока) в свою очередь обычно считывает другую программу В с фиксированного места на диске и передает на нее управление. Местоположение программы В на диске и число занимаемых ею блоков являются константами в программе А. При изменении расположения или размеров программы В надо изменить и программу А в нулевом блоке. Именно поэтому программа В считывается программой А, а не сразу программой начальной загрузки из ПЗУ.

Обычно программа В — это *монитор операционной системы*, т. е. программа, которая взаимодействует через терминал с человеком, анализирует последовательности нажимаемых человеком клавиш и выполняет соответствующие этим последовательностям действия. Кроме того, в состав программы В входят файловая система, подпрограммы обработки прерываний и другие необходимые для работы подпрограммы, вместе образующие так называемое *ядро операционной системы*. (*Операционная система* в целом — это весь комплекс служебных программ, обеспечивающих доступ к возможностям оборудования и создающих обстановку, в рамках которой программист может создавать, изменять и выполнять свои программы.)

В свою очередь, основная функция монитора — это загрузка (чтение с диска) и выполнение каких-то программ в кодах. Сами программы хранятся в именованных файлах на диске. Для выполнения, например, программы С человек, нажимая клавиши на клавиатуре терминала, должен в каком-то виде указать имя программы С. Действия монитора состоят в анализе последовательности нажатых человеком клавиш, определении имени файла с программой С, чтении содержимого этого файла (программы С) в память и выполнении С (т. е. передаче управления программе С).

Заметим, что исключительные ситуации и отказы при выполнении программы С обрабатываются программами обработки прерываний операционной системы. Эти программы вызывают выдачу соответствующего сообщения человеку, прекращение выполнения программы С и возобновление работы монитора В. Более того, монитор может объявить исключительной ситуацией и попытку выполнить команду HALT (стоп) (реально такое объявление означает установку или очистку некоторых не рассматривавшихся нами битов в регистре PS). Таким образом, выполнение программы С ни при каких условиях не приводит к останову ЭВМ — и при нормальном, и при ненормальном завершении программы С возобновляется работа монитора операционной системы В, который опять анализирует нажимаемые человеком клавиши, выполняет соответствующие программы и т. д.

Программирование в кодах. Если программисту по каким-то причинам требуется написать программу прямо в кодах ЭВМ, то в качестве вызываемой из монитора программы С обычно выступает специальная служебная программа *редактор программ в кодах* (на физическом уровне). После начала выполнения этот редактор изображает в символьном виде на экране терминала некоторую часть памяти ЭВМ, анализирует последовательности нажимаемых человеком клавиш, сопоставляет эти последовательности с некоторыми действиями и выполняет их. Обычно с помощью такого редактора человек может ввести, изменить программу в кодах (так же, как в экранном редакторе текстов вводится и изменяется текст), выполнить или записать на диск полученную программу. Записанную на диск программу в дальнейшем можно выполнять с помощью монитора ОС, не пользуясь редактором программ в кодах.

Компиляция. Обычно, однако, нет необходимости прибегать к программированию в кодах — программист чаще работает на каком-нибудь языке программирования более высокого уровня (например, на том, который был использован в первых трех главах этой книги), а перевод (компиляцию) программ с этого языка в коды автоматически осуществляет сама ЭВМ. Рассмотрим более подробно, как это происходит.

Прежде всего программист через монитор ОС вызывает специальную служебную программу *редактор текстов*, с помощью которой может ввести, изменить произвольный текст (например, текст своей программы) и записать его в файл на диск (частично такой редактор был описан и реализован в разд. 18). После этого управление возвращается в монитор и программист вызывает другую служебную программу — *компилятор*, указав последнему имя файла с текстом исходной программы и имя файла, в котором надо получить соответствующую программу в кодах. Хотя реальные компиляторы с языков программирования высокого уровня — это довольно сложные программы, в целом они работают примерно так, как было изложено в разд. 10.

После завершения компиляции программист получает сообщения об ошибках в тексте исходной программы или, если ошибок нет, программу в кодах, а управление снова возвращается в монитор. Полученную программу в кодах программист далее может вызвать средствами монитора ОС.

На практике, впрочем, компилятор компилирует исходную программу не прямо в коды, а в некоторую промежуточную форму, называемую *объектным кодом*, который хотя и близок к коду ЭВМ, но не совпадает с ним. Дело в том, что при изменениях в большом проекте, состоящем из десятков исполнителей и сотен отдельных программ, нет необходимости перекомпилировать весь проект — достаточно заново преобразовать одну измененную программу или исполнителя. Однако при компиляции отдельно взятой программы некоторые имена (например, имена подпрограмм) нельзя заменить адресами — ведь неизвестно, в каких местах памяти ЭВМ будут лежать те или иные программы. Поэтому после компиляции в такой промежуточный объектный код необходим еще один этап — его называют *сборкой* или *редактированием связей*, во время которого из объектных кодов отдельных программ и исполнителей собирается вся программа в кодах целиком. На этом же этапе происходит преобразование оставшихся имен в адреса и подключение стандартных подпрограмм (таких, как минимум, максимум, sin, cos и др.).

Таким образом, с помощью одной служебной программы — *редактора текстов* — программист должен записать в какие-то файлы на диске исходные тексты своей программы; с помощью другой — *компилятора* — преобразовать их в объектные коды; с помощью третьей — *редактора связей* (эту программу также называют *сборщиком*) — получить программу в кодах (так называемый *загрузочный файл*). После этого полученную программу в кодах (загрузочный файл) можно выполнить средствами монитора ОС.

Интерпретация. Компиляция исходной программы в коды ЭВМ — не единственный путь выполнения программы. Другим, широко распространенным подходом является *интерпретация*, при которой специальная служебная программа *интерпретатор* читает текст исходной программы, анализирует его и тут же выполняет (см. разд. 10, с. 161). Обычно интерпретатор может читать исходный текст и из файла на диске, и из памяти ЭВМ, и прямо с клавиатуры терминала по мере ввода его программистом.

Соотношение компиляции и интерпретации. И компиляция, и интерпретация имеют и достоинства, и недостатки. Основное достоинство компиляции состоит в том, что в конечном счете *выполняется программа в кодах*, т. е. на этапе выполнения никакой лишней работы, непосредственно с выполнением программы не связанной, не происходит. При интерпретации, наоборот, ЭВМ выполняет саму программу-интерпретатор, а уже программа-интерпретатор анализирует и выполняет программу человека. Таким образом, требуемые для выполнения свойства программы многократно вычисляются по ее тексту (см. также разд. 11, с. 187). В результате при интерпретации программа выполняется на порядок (а иногда и на два порядка) медленнее, чем будучи скомпилированной в коды.

Основной недостаток компиляции — наличие как минимум *двух представлений программы* (исходного и в кодах), с которыми должен работать программист. На этапе выполнения программы в кодах типичными, например, являются сообщения «нечетный адрес» или «деление на нуль при PC = 10546». Но для того чтобы понять, какие ошибки в *исходном тексте* программы привели к этим сообщениям, программист часто вынужден проделывать большую (иногда весьма нетривиальную и творческую) работу. Кроме того, после изменения программы он должен проводить ряд технических действий (компиляция, сборка, запуск), также отнимающих некоторую часть его интеллектуальных и временных ресурсов. При интерпретации, наоборот, измененную программу можно немедленно начать выполнять, и при этом сообщения об ошибках выдаются в терминах исходной программы.

Компиляция и интерпретация — это в каком-то смысле два крайних подхода. Существует и множество компромиссных решений. Довольно широко распространена компиляция исходной программы не в коды реальной ЭВМ, а в коды некоторой промежуточной *виртуальной* (воображаемой) *машины* с последующим использованием интерпретатора кодов виртуальной машины для выполнения программы. Этот подход применяется, например, при нехватке оперативной памяти, поскольку программа в кодах специальным образом подобранной виртуальной машины может быть существенно компактнее программы в кодах реальной ЭВМ.

К конечному результату труда программиста — разрабатываемым программам — предъявляются различные требования. Мы уже отмечали (разд. 4), что программа как минимум должна быть правильной, понятной и легко изменяемой. Другими требованиями часто являются эффективность (высокая скорость выполнения) и компактность (небольшой объем занимаемой памяти) программы. Методы работы человека на ЭВМ (так называемая операционная обстановка) в первую очередь должны быть подчинены достижению этих главных целей. Однако, коль скоро мы рассматриваем процесс создания программ, следует сформулировать еще одно требование — *удобство разработки* программ для программиста.

Требования правильности, понятности и легкости изменения означают, что программист должен использовать язык программирования высокого уровня с понятными конструкциями, соответствующими тем терминам, в которых мыслит человек. Требование эффективности почти с необходимостью означает, что программа должна быть скомпилирована, т. е. что конечным результатом должна быть программа в кодах. Заметим, хотя это и не вполне верно, что требование эффективности предъявляется к *конечному результату*, т. е. к уже разработанной программе. В процессе создания программы эффективность, как правило, не очень важна. В то же время отсутствие необходимости работать с двумя представлениями программы (исходным и в кодах), отсутствие связанных с этим технических манипуляций, а также возможность мгновенного запуска измененной программы на выполнение делают интерпретатор существенно более удобным инструментом разработки программ, чем компилятор.

Таким образом, идеальным представляется симбиоз из полностью совместимых интерпретатора (используемого при разработке программы) и компилятора (используемого после разработки для получения эффективной программы в кодах).

В настоящее время появился и ряд новых подходов, выходящих за рамки традиционных компиляции и интерпретации. Мы рассмотрим два из них — синтаксически ориентированное редактирование и работу в инструментальных средах.

Синтаксически ориентированные редакторы. Серьезным недостатком интерпретации является тот факт, что ни разу не выполнявшиеся части программы не будут проанализированы даже на формальную корректность. Это сильно снижает надежность (степень правильности) программы. Кроме того, даже при использовании интерпретатора программист вынужден делать много лишней работы, связанной, например, с посимвольным вводом основных конструкций языка программирования. Так, конструкция «**программа — дано — получить — конец программы**» традиционно вводится

как текст — отдельные символы, определенным образом расположенные по строкам, что требует примерно 60 нажатий на клавиши. В то же время с содержательной точки зрения вся эта конструкция представляет собой единое неделимое целое и как таковая должна вставляться за одно-два нажатия на клавиши.

Указанные недостатки отсутствуют при использовании для создания программ вместо редактора текстов специального *синтаксически ориентированного редактора*, который воспринимает программу не как произвольную последовательность строк и символов, а учитывает правила записи программ (*синтаксис* языка программирования). Кроме того, такой синтаксически ориентированный редактор обычно осуществляет и полный глобальный контроль за синтаксической корректностью программы в целом.

Инструментальные среды. Полученная тройка (синтаксически ориентированный редактор текстов программ, интерпретатор, компилятор), впрочем, тоже не свободна от недостатков. Во-первых, теперь много лишней работы делает интерпретатор, например разбирая по строкам и интерпретируя ту же конструкцию **«программа — дано — получить — конец программы»**. Эту лишнюю работу можно не делать, если заменить синтаксически ориентированный редактор *текстов программ* синтаксически ориентированным редактором самих программ, результатом работы которого будет не только формально правильный текст, но и некоторое *внутреннее представление*, отвечающее структуре программы. Кроме того, надо, чтобы интерпретатор интерпретировал внутреннее представление программы, а не ее текст. Реально это означает интеграцию редактора программ и интерпретатора в единую систему, работающую над общим внутренним представлением программы.

Во-вторых, обычно процесс разработки программы — длительный процесс. В момент создания одних компонент другие могут считаться законченными и к ним часто начинает предъявляться требование эффективности. Для того чтобы законченные компоненты можно было эффективно вызывать и выполнять, их следует скомпилировать. В то же время разрабатываемые новые компоненты должны интерпретироваться. Другими словами, компилятор также должен быть интегрирован в единую систему с редактором программ и интерпретатором. Эту единую систему — идеальный инструмент для разработки программ — принято называть *инструментальной средой*.

Итак, *инструментальная среда* — это единая интегрированная система, состоящая из: а) экранного редактора программ, обеспечивающего ввод и изменение программ в терминах используемого языка программирования и полный глобальный синтаксический контроль за правильностью программы; б) интерпретатора с разви-

той системой визуализации и управления процессом выполнения; в) встроенного компилятора, обеспечивающего эффективное выполнение законченных компонент программы. Обычно инструментальная среда обеспечивает также загрузку и выполнение программ в кодах из файлов на диске и используется вместо монитора ОС и самой ОС. Другими словами, после включения ЭВМ программист начинает работу сразу в инструментальной среде и работает только в ней.

ЭПВМ. Заметим, что, отвлекаясь от вопросов внутреннего устройства, ЭВМ с инструментальной средой можно рассматривать как Универсального Выполнителя программ. Другим Универсальным Выполнителем является комплекс ЭВМ + ОС + редактор текстов + компилятор + редактор связей. Хотя технически взаимодействие человека с этими Универсальными Выполнителями выглядит по-разному, оба они являются выполнителями программ, записанных на одном и том же языке программирования. Язык программирования, впрочем, не является единственным или предопределенным — создавая, например, разные компиляторы, можно превращать одну и ту же ЭВМ в Универсальных Выполнителей программ, написанных на разных языках программирования. В настоящее время в мире насчитывается несколько сотен реально используемых языков программирования (из них несколько десятков широко распространенных) и продолжают появляться все новые.

Работа на ЭВМ непрограммиста. До сих пор мы говорили о работе на ЭВМ только программиста. Другие люди (кассиры, конструкторы, писатели и пр. — их обычно называют *пользователями*) имеют дело не с Универсальным Выполнителем, а с конкретными исполнителями, созданными трудом программиста. Сам исполнитель А, конечно, может быть устроен как R(A, E) + УВ + E, но это детали его внутреннего устройства. Пользователь работает с А как с каким-то инструментом своей профессиональной деятельности и должен знать лишь внешнее описание А.

Всюду в этой книге мы реализовывали различных исполнителей, совершенно не предполагая их взаимодействия с человеком в процессе работы. Проектирование такого взаимодействия (его принято называть *внешним интерфейсом*) — это отдельная важная и сложная задача. Мы не будем подробно останавливаться на ней, однако подчеркнем несколько очевидных принципов, которым такое взаимодействие исполнителя с человеком должно удовлетворять.

Наиболее важным является осознание того факта, что любой исполнитель, любая система — не более чем *инструмент* в руках пользователя. Как инструмент система не должна пытаться имитировать разумное существо, «беседовать» с человеком, задавать вопросы, оценивать поведение человека и т. п. Она должна выступать в качестве простого, понятного и эффективного средства, повышаю-

щего производительность и/или качество труда пользователя, задействовать интуитивные представления пользователя о характере и предмете этого труда, профессиональный и жизненный опыт пользователя, в частности опыт ориентировки в пространстве, а также геометрическое, чувственное восприятие образов на экране. Общий стиль использования системы должен соответствовать стилю использования традиционных инструментов, таких как молоток, утюг и пр. Хорошим приемом выявления ошибок в проектировании интерфейса является подстановка названия какого-нибудь бытового инструмента, например слова «утюг», вместо слов «исполнитель», «диалоговая система», «ЭВМ» и др. в тексты, описывающие взаимодействие разрабатываемой системы с человеком.

Как и при использовании любого другого инструмента, человек должен обладать *полной свободой воли* — ни в какой момент система не должна навязывать предопределенный заранее порядок действий или темп работы, менять свои свойства (например, в зависимости от скорости работы человека). Все, что происходит, должно происходить только по явному указанию человека и под полным его контролем.

Следует иметь в виду, что пользователю вообще не нужна никакая, даже самая замечательная система — ему нужно делать свое дело. Чем меньше система будет его отвлекать, чем меньше усилий человек будет тратить на взаимодействие с системой, тем лучше. В частности, следует максимально разгрузить пользователя от зачастую не имеющих прямого отношения к делу многочисленных меню, окон, звуковых эффектов и цветовых пятен.

Наконец, заметим, что система должна быть ориентирована на *профессионала* в данной области деятельности. Овладение системой, как и любым другим инструментом, может требовать некоторого времени, приобретения определенных навыков работы, а иногда и специального обучения. Другими словами, не следует предполагать, что с системой будет работать новичок, впервые в жизни ее увидевший, или человек, не имеющий никакого представления о предметной области и целях собственной деятельности.

Разумеется, все эти принципы надо заменить на противоположные, если проектируемая система рассматривается не как инструмент профессиональной деятельности пользователя, направленный на достижение каких-то внешних целей, а как интересная и красивая игрушка, основная цель которой — доставить пользователю эстетическое наслаждение своими возможностями и внешним видом.

ГЛАВА 5
ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ ФОРТРАН

Язык программирования Фортран (от англ. FORmula TRANslator — переводчик формул) является одним из первых языков программирования высокого уровня, т. е. языков, не зависящих от особенностей конкретных ЭВМ. Его первая версия была разработана в 1955—1956 г. В дальнейшем язык неоднократно дорабатывался и улучшался. Наибольшее распространение получила версия Фортран IV, разработанная в начале шестидесятых годов. В настоящее время, однако, наряду с этой широко распространенной версией появились и используются более поздние: Фортран IV плюс, Фортран V, Фортран 77. Даже если говорить об одной версии языка, например о Фортране IV, нужно различать реализации этой версии на разных ЭВМ и в разных операционных системах. Такие отличающиеся друг от друга реализации принято называть *диалектами* языка. В этой главе мы будем иметь дело с диалектом языка Фортран IV плюс для операционной системы RSX11M ЭВМ СМ-4.

Впрочем, различия между диалектами и версиями языка обычно не очень велики. Программу, написанную на одном из диалектов одной из версий языка, как правило, легко перевести на другой диалект или другую версию. Поэтому всюду далее будем употреблять слово «Фортран», не указывая, о какой версии или диалекте идет речь.

В настоящее время в мире насчитывается несколько сотен реально используемых языков программирования. Разные языки используются в разных областях применения ЭВМ. В области научно-технических расчетов и различных систем автоматизации бесспорным лидером был и остается Фортран. Это одна из причин, по которой ему отдано предпочтение в этой книге. Кроме того, Фортран позволяет реализовать понятие исполнителя и аксиоматизирует важную особенность современных ЭВМ — линейную организацию памяти, поэтому на Фортране можно писать достаточно эффективные программы.

Целью главы не является подробное изложение всех деталей языка Фортран, в ней, например, совершенно не затронуты вопросы форматного ввода/вывода информации, вычисляемые переходы и др. Мы лишь дадим достаточно полное представление об этом языке и покажем, как перекодировать ранее написанные про-

граммы на Фортран. Другими словами, покажем *общий стиль* программирования на Фортране. Читатель, интересующийся не образцами программирования, а полным и подробным описанием Фортрана самого по себе, может найти такое описание, например, в книге К. Джермейн «Программирование на IBM-360» (Москва, Мир, 1983).

Практическая работа на Фортране зависит от конкретных ЭВМ, операционной системы, компилятора, других компонент базового программного обеспечения. Весь этот комплекс в целом мы будем называть Фортран-машиной и рассматривать как Универсального Выполнителя программ на Фортране.

21. Справочные сведения о языке Фортран. Простейшие примеры программ

Основу любого языка составляют: 1) основные структурные единицы, управляющие конструкции и операторы языка, 2) объекты, значения, типы и структуры данных, которыми можно оперировать с помощью этих структурных единиц, конструкций и операторов.

Пример программы. Для создания общего представления о языке Фортран приведем пример программы на Фортране, которая выводит на экран терминала первые 10 чисел Фибоначчи.

```
CXXXX ПРОГРАММА ПЕЧАТЬ ПЕРВЫХ 10 ЧИСЕЛ ФИБОНАЧЧИ
C
      INTEGER N(10)
      N(1)=1
      N(2)=1
      DO 1 I=3,10
        N(I) = N(I-1) + N(I-2)
1     CONTINUE
      TYPE *,'ПЕРВЫЕ 10 ЧИСЕЛ ФИБОНАЧЧИ: ', N
      END
```

Мы поясним эту программу чуть ниже, а сейчас опишем некоторые правила записи программ на Фортране.

Размещение информации внутри строки. Фортран-программа состоит из *операторов* и записывается на строках фиксированной длины (обычно 72 позиции). Расположение информации внутри строки существенно и описывается следующими правилами:

— символ C в первой (самой левой) позиции строки означает, что вся строка является *комментарием* и будет полностью проигнорирована Фортран-машиной (C от слова Comment — ком-

ментарий). Комментарием также считается часть любой строки от восклицательного знака (!) и до конца строки;

— в позициях 1—5 может быть записана так называемая *метка* — целое положительное число от 1 до 99999 (рекомендуется использовать позиции 2—5, чтобы метки отделялись от символов комментария С в первой позиции);

— если оператор не уместился на одной строке, его можно перенести на одну или несколько следующих строк. В строках продолжения в позициях 1—5 должны быть пробелы, в позиции 6 — не пробел (позиция 6 называется по этой причине *колонкой продолжения*). Рекомендуется в качестве символа продолжения использовать символ @ — если он случайно попадет не в 6-ю позицию, Фортран-машина заведомо предупредит об ошибке;

— в позициях 7—72 располагается текст оператора или его продолжения и комментарии. Пробелы в этих позициях, как правило, ничего не значат, и их можно располагать по своему вкусу, делая текст более приятным для глаза (рекомендуется начинать операторы с 9 позиции, оставляя 7 и 8 свободными — при вводе Фортран-программы с терминала в 9-ю позицию обычно можно попасть, всего один раз нажав на клавишу табуляции).

Разберем теперь приведенную выше программу. Первые две строки этой программы, начинающиеся с заглавной латинской буквы С, — это комментарий. Следующий оператор

```
INTEGER N(10)
```

начинается с 9-й позиции, занимает одну строку и описывает вектор N элементов типа «целое число с индексом от 1..10» (индексы в Фортране всегда начинаются с 1). За ним идут два оператора присваивания

```
N(1) = 1
N(2) = 1
```

которые устанавливают N(1) и N(2) равными 1 (операция присваивания в Фортране обозначается просто знаком =). Оператор

```
DO 1 I = 3,10
```

— это оператор цикла. Он заставляет Фортран-машину выполнить следующую за ним группу строк вплоть до строки с меткой 1 последовательно для I = 3, 4, 5, ... 10. Оператор присваивания

```
N(I) = N(I - 1) + N(I - 2)
```

вычисляет для каждого I от 3 до 10 очередное число Фибоначчи. Оператор

```
1 CONTINUE
```

— это оператор «ничего не делать». Он используется, чтобы отметить место между другими операторами Фортрана (в данном случае, чтобы отметить конец цикла DO, используется оператор CONTINUE с меткой 1). Оператор

```
TYPE *,'ПЕРВЫЕ 10 ЧИСЕЛ ФИБОНАЧЧИ:',N
```

выводит на экран терминала сначала текст ПЕРВЫЕ 10 ЧИСЕЛ ФИБОНАЧЧИ: , а потом и сами числа. Так как в операторе INTEGER было указано, что индекс в векторе N меняется от 1 до 10, то будет выведено 10 чисел. Символ * после слова TYPE означает, что выбор формы, в которой будет выводиться информация, представляется на усмотрение Фортран-машины. Наконец, оператор

```
END
```

означает конец текста программы на Фортране.

Секции. В целом программа на Фортране состоит из одной или нескольких структурных единиц, называемых *секциями*. Есть три вида секций: *основная секция*, *подпрограмма* и *подпрограмма-функция* (или просто *функция*). В программе должна быть одна и только одна основная секция — именно с нее начинается выполнение Фортран-программы. Основная секция начинается служебным словом PROGRAM (которое, впрочем, можно не писать), подпрограмма — служебным словом SUBROUTINE, а функция — служебным словом FUNCTION. Любая секция кончается служебным словом END. Каждая секция представляет собой текст, состоящий из строк. Секции записываются одна за другой — сначала основная, а затем все остальные в произвольном порядке.

Каждая подпрограмма или функция должна иметь *имя*, отличное от имен других подпрограмм и функций в данной программе. *Имя* — это последовательность не более чем из 6 заглавных латинских букв и цифр, начинающаяся с буквы. Функция в Фортране должна иметь хотя бы один формальный параметр, подпрограмма может параметров не иметь.

Чтобы создать представление об общем виде программы на Фортране, приведем без пояснений пример программы, которая состоит из двух секций: основной и подпрограммы-функции NOD. Эта программа запрашивает у человека два натуральных числа и выводит на экран терминала их общий делитель:

```
TYPE *,'ВВЕДИТЕ ДВА НАТУРАЛЬНЫХ ЧИСЛА'
ACCEPT *, N1, N2
K = NOD( N1,N2 )
TYPE *,'НАИБОЛЬШИЙ ОБЩИЙ ДЕЛИТЕЛЬ РАВЕН ', K
END
```

```
C-------------------------------------------------------
      FUNCTION NOD(I,J) !НАИБОЛЬШИЙ ОБЩИЙ ДЕЛИТЕЛЬ
      M=I
      N=J
 1    CONTINUE
      IF( M.EQ.Ø .OR. N.EQ.Ø ) GOTO 11
        MN = M - N
        IF( MN.GE.Ø ) M= MN
        IF( MN.LT.Ø ) N=-MN
      GOTO 1
 11   CONTINUE
      NOD = M + N
      END
```

Общая структура секции. Всякая секция логически делится на две части — *описания* и *выполняемые операторы*. Например, в программе, печатающей первые 10 чисел Фибоначчи, к описаниям относится оператор INTEGER, описывающий вектор N из 10 элементов, а остальные операторы являются выполняемыми. Правила Фортрана требуют, чтобы внутри секции все операторы описаний располагались *до* всех выполняемых операторов. Таким образом, хотя явной черты, отделяющей описания от тела программы, в Фортране нет, неявно, на логическом уровне эта черта существует: все описания должны быть расположены выше нее, а все выполняемые операторы — ниже.

Вызовы подпрограмм и функций. Рассмотрим подпрограмму без параметров

```
SUBROUTINE STAR
TYPE *,'**********'
END
```

которая выводит на экран терминала строку из 12 звездочек. Вызов подпрограммы в Фортране записывается с помощью оператора CALL. Например, фрагмент программы

```
N=1Ø
CALL STAR
TYPE *, '* N =', N
CALL STAR
```

выводит на экран следующие строчки:

```
**********
* N =      1Ø
**********
```

Вызов подпрограммы-функции не требует оператора CALL — для вызова (использования) функции (как и для использования программ, вырабатывающих значение) надо просто указать имя функции с фактическими параметрами в выражении или правой части оператора присваивания, например

```
K = NOD (N, 10) + 1
```

Управляющие конструкции. Операторы Фортрана

GOTO — перейти к строке с указанной меткой,
CONTINUE — ничего не делать,
IF — если

позволяют писать на Фортране фрагменты, эквивалентные любым управляющим конструкциям.

При записи оператора Фортрана в позициях 1—5 можно задать метку — десятичное число от 1 до 99999. Хотя метку можно задать для большинства операторов Фортрана, рекомендуется помечать только операторы с CONTINUE — это делает текст более понятным и к тому же удобно при изменениях программ.

Обычно Фортран-машина выполняет операторы программы один за другим, последовательно. Этот последовательный порядок можно изменить с помощью оператора GOTO. При выполнении, например, оператора

```
GOTO 10
```

Фортран-машина переходит к выполнению строки с меткой 10 той же секции. Условный оператор

```
IF (условие) оператор
```

работает так: сначала проверяется условие в скобках после IF. Если оно не выполнено, то оператор, записанный после скобок условия, не выполняется, а выполняется следующая строка программы. Если же условие выполнено, то выполняется оператор Фортрана, записанный после скобок условия. Например, оператор

```
IF (N.NE.0) K = K + 1
```

в случае, если $N \neq 0$ (операция сравнения $\neq$ записывается в Фортране в виде .NE.), увеличит значение K на 1, а если $N = 0$ — оставит неизменным. При выполнении фрагмента программы

```
      IF( N.NE.Ø ) GOTO 1Ø
      X=Ø
      Y=Ø
      Z=Ø
1Ø    CONTINUE
```

если N ≠ 0, то произойдет переход на строку с меткой 10 и значения X, Y, Z останутся без изменений. Если же N = 0, то оператор GOTO 10 выполняться не будет и X, Y, Z станут нулевыми.

С помощью операторов IF, GOTO, CONTINUE можно записывать и конструкции циклов. Например, цикл «пока» можно записать так:

```
1     CONTINUE                          !цикл пока условие
      IF(.NOT.(условие)) GOTO 11        !. выполнять
            . . .                       !.   . . .
      GOTO 1                            !конец
11    CONTINUE                          !цикла
```

В этом фрагменте оператор

GOTO 1

обеспечивает «зацикливание», а оператор

IF (.NOT.(условие)) GOTO 11

— выход из цикла при нарушении условия (операция логического отрицания в Фортране обозначается .NOT.).

Аналогично может быть записана на Фортране и конструкция **«если — то — иначе — конец если»**:

```
      IF(.NOT.(условие)) GOTO 20        !если условие то
         . . .                          !.    . . .
      GOTO 21                           !.
20    CONTINUE                          !. иначе
         . . .                          !.    . . .
21    CONTINUE                          !конец если
```

Приведенная выше общая форма записи конструкции **«если — то — иначе — конец если»**, однако, достаточно громоздка. Во многих случаях можно использовать какие-то более компактные частные схемы. Например, наличие решений уравнения $x^2 + px + q = 0$ можно проверить так:

```
D = P*P-4*Q
IF( D.GE.0 ) TYPE *,'РЕШЕНИЯ ЕСТЬ'
IF( D.LT.0 ) TYPE *,'РЕШЕНИЙ НЕТ'
```

(.GE. обозначает операцию $\geqslant$, а .LT. — операцию $<$).

Цикл DO. Для того чтобы выполнять некоторый фрагмент программы при последовательно меняющихся значениях целочисленного параметра, в Фортране есть специальный оператор DO. Например, фрагмент программы

```
N1=0
N2=0
N3=0
```

```
      DO 20 I=1,10          !цикл ∀ I ∈ 1..10
        N1=N1+I             !. выполнять
        N2=N2+I*I           !.      . .
        N3=N3+I*I*I         !.        .
 20   CONTINUE              !конец цикла
```

вычисляет сумму N1, сумму квадратов N2 и сумму кубов N3 натуральных чисел от 1 до 10. В этом примере I называется переменной цикла, а 1 и 10 — нижней и верхней границами цикла. Перебор значений из диапазона 1..10 можно осуществлять и в обратном порядке (реверс), написав

DO 20 I = 10,1,−1

Вообще, цикл

DO 20 I = I1,I2,I3

выполняется сначала для I = I1, затем для I = I1 + I3, затем для I = I1 + I3 + I3 и т. д. до тех пор, пока I ⩽ I2 (если I3 > 0) или I ⩾ I2 (если I3 < 0). I3 здесь называется шагом цикла. Если шаг цикла не указан, то он считается равным 1.

Неприятной особенностью оператора DO является то, что при любых I1, I2, I3 цикл всегда выполняется *хотя бы один раз* при I = I1. Поэтому, например, нельзя вычисление неотрицательной степени числа X записать в виде

```
      Y = 1
      DO 20 I=1,N
        Y = Y*X
 20   CONTINUE
```

При N = 0 выполнение этого фрагмента программы даст Y = X, а не Y = 1. Чтобы избежать этой ошибки в случаях, когда диапазон цикла может оказаться пустым, следует проверить непустоту диапазона перед выполнением цикла:

```
      Y = 1
      IF( 1.GT.N ) GOTO 21
      DO 20 I=1,N
        Y = Y*X
 20   CONTINUE
 21   CONTINUE
```

Предопределенные типы. В Фортране используются объекты и значения 4-х предопределенных типов:

LOGICAL (логический, т. е. тип да/нет)
INTEGER (целый),
REAL (действительный),
COMPLEX (комплексный).

Множество значений типа LOGICAL состоит из двух элементов .TRUE. (истина, т. е. да) и .FALSE. (ложь, т. е. нет). На ЭВМ СМ-4 множество значений целых чисел является диапазоном от —32768 до 32767 (т. е. целое число представляется с помощью одного слова — двух байт). Множество значений действительных чисел состоит из $\approx 2^{32}$ элементов, которые позволяют представить любое число в диапазоне $-10^{38} .. -10^{-38} \cup \{0\} \cup 10^{-38} .. 10^{38}$ приближенно, с точностью до 7 значащих цифр. Действительные числа хранятся в двух словах (четырех байтах) каждое.

Представляемые значения можно записывать либо в обычном виде — с десятичной точкой, — либо в экспоненциальном:

1.5E3, т. е. $1.5 \cdot 10^3 = 1500.0$
1.32E — 2, т. е. $1.32 \cdot 10^{-2} = 0.0132$

Комплексное число является просто парой (a, b) действительных значений, где a — действительная, а b — мнимая части комплексного числа, и записывается в виде CMPLX(a, b):

CMPLX (0.0,1.0), т. е. 0.0 + 1.0i
CMPLX (5.2,—7.45), т. е. 5.2 — 7.45i

Никаких идеальных значений $+\infty$, $-\infty$ и неопр типы LOGICAL, INTEGER, REAL, COMPLEX не содержат.

Над значениями предопределенных типов допустимы все стандартные операции: присваивание (=), сравнение на равенство (.EQ.) и неравенство (.NE.), для арифметических типов — арифметические операции, для LOGICAL — логические, в выражениях можно использовать скобки для явного указания порядка операций и пр. Обозначения операций таковы:

= — присваивание,
.EQ. — сравнение на равенство (EQual),
.NE. — сравнение на неравенство (Not Equal),
.LT. — меньше (Less Than),
.LE. — меньше или равно (Less or Equal),
.GE. — больше или равно (Greater or Equal),
.GT. — больше (Greater Than),
.AND. — и,
.OR. — или,
.NOT. — не,
+ — сложение,
— — вычитание,
* — умножение,
/ — деление и деление нацело,
** — возведение в степень.

Обратите внимание, что знак / используется в Фортране для обозначения *двух разных операций* — деления и деления нацело. Выбор конкретной операции осуществляется по типу делителя и делимого — если они оба целые, то деление производится нацело, в противном случае — как обычно. Таким образом,

3/5 + 2/5 = 0, 3/5 + 2.0/5 = 0.4,
3.0/5 + 2/5 = 0.6, 3.0/5 + 2.0/5 = 1.0.

Следует соблюдать осторожность и при преобразованиях формул

3/5 + 2/5 = 0, но (3 + 2)/5 = 1,
0.5/10 = 0.05, но 0.5 * (1/10) = 0.0.

Над значениями предопределенных типов определено какое-то количество стандартных функций (в Фортране они называются *встроенными*). Вот некоторые из них:

FLOAT — преобразование целого в действительное,
INT — целая часть действительного числа,
MOD — остаток от деления нацело,
ABS — модуль действительного числа,
IABS — модуль целого числа,
EXP — экспонента,
SQRT — квадратный корень,
SIN — синус,
COS — косинус,
ALOG — натуральный логарифм,
ALOG10 — десятичный логарифм,
ATAN — арктангенс,
CEXP — комплексная экспонента,
MIN0, AMIN0, MIN1, AMIN1 — минимум нескольких чисел,
MAX0, AMAX0, MAX1, AMAX1 — максимум нескольких чисел.

В именах функций, находящих минимум или максимум, 0 в конце означает, что аргументы функции являются целыми, а 1 — что действительными. Если имя начинается с буквы A, то значение функции является действительным, а если с M — то целым. Таким образом, AMIN0(3, 2)/5 = 0.4, но MIN0(3, 2)/5 = 0. Функции, находящие минимум и максимум, в Фортране могут иметь не 2 аргумента, а несколько. Например, можно написать MIN0(I, J, K) или AMIN1(X, X + Y, X + Y + Z, 1.0). Более полный список встроенных функций Фортрана можно найти в книге К. Джермейн «Программирование на IBM-360».

Вот несколько примеров записи арифметических и логических выражений на Фортране:

```
(X.EQ.Y).OR.(ALOG(X) ** 2.0.LT.Y)
(A.EQ.B).AND.((A.EQ.C).OR.(A.EQ.D))
```

```
1.473 * X + Y ** 3 — SIN(COS(X/EXP(X)))
CEXP(X + Y * CMPLX(0.0,1.0))
(MOD(N,4).EQ.1).OR.(MOD(N,4).EQ.2)
```

Способы конструирования. В Фортране есть только один способ конструирования объектов — *массив*. Массивы бывают одномерные, двумерные, трехмерные и т. д. Одномерный массив — это вектор, индекс которого меняется от 1 до некоторого максимального значения. Двумерный массив — это матрица, каждый индекс которой изменяется от 1. Элементами массивов могут быть только объекты одного из четырех предопределенных типов: LOGICAL, INTEGER, REAL, COMPLEX. Максимальные значения индексов массива должны указываться явно, в числовом виде, в момент написания программы. Например, запись

```
REAL A(5),B(5,10)
INTEGER N(100,2,3)
```

задает вектор A действительных чисел с индексом 1..5, матрицу B действительных чисел с индексами 1 .. 5, 1 .. 10 и трехмерный целочисленный массив N с индексами 1..100, 1..2, 1..3.

Для того чтобы использовать или изменить элемент массива, надо написать имя массива, а за ним в круглых скобках — значения индексов, например:

```
N(1,K,I + J) = 0
B(I,1) = A(I)
```

Другие объекты (стеки, списки и пр.) кодируются на Фортране путем создания соответствующих исполнителей (см. следующий раздел).

Задание объектов и типов объектов. Объекты в Фортране (они обычно называются *переменными*) локализованы внутри секции и могут быть заданы в ее начале. Для явного задания массивов и объектов предопределенных типов (будем их называть *простыми*) используются операторы

```
LOGICAL  ⟨имена переменных типа да/нет⟩
INTEGER  ⟨имена целых переменных⟩
REAL     ⟨имена действительных переменных⟩
COMPLEX  ⟨имена комплексных переменных⟩
```

В этих операторах имена переменных перечисляются через запятую, для массивов указываются максимальные значения каждого индекса. Например:

```
REAL  V(3),A(4,3)
```

Типы переменных можно и не описывать явно. В этом случае тип определяется по имени: если имя переменной начинается с

одной из 6 букв IJKLMN, то переменная считается целой, в противном случае — действительной. Это так называемое *IJKLMN-правило* можно изменить или дополнить с помощью оператора

IMPLICIT ⟨имя типа⟩ (⟨буква⟩)

Например, после оператора

```
IMPLICIT LOGICAL (Q)     !тип "да/нет"
```

все переменные, начинающиеся с буквы Q и не описанные явно, будут иметь тип да/нет (LOGICAL). Будем называть такое дополнение *IJKLMN/Q-правилом* и придерживаться его при написании программ на Фортране.

Параметры. В качестве фактического параметра при обращении к подпрограмме или к подпрограмме-функции может быть задано имя другой подпрограммы или функции. В этом случае его нужно обязательно описать оператором EXTERNAL.

Пусть, например, нам нужно вычислить корень непрерывной функции F от одной переменной на отрезке изоляции корня [T0, T1]. Для этого в Фортране можно написать функцию, аргументами которой будут F, T0, T1, а вырабатываемым значением — значение корня с некоторой точностью:

```
FUNCTION ROOT( F, T0, T1 )
  . . .
END
```

Мы реализуем эту функцию позже, а сейчас используем ее для аргументов F = SIN, T0 = 3.0, T1 = 4.0:

```
T = ROOT(SIN,3.0,4.0)
```

Однако если просто написать этот оператор, то Фортран-машина воспримет SIN как имя простой переменной, которая не описана явно. Коль скоро это имя начинается с буквы S, то Фортран-машина посчитает SIN переменной, имеющей тип REAL, заведет такой локальный объект и передаст его в качестве параметра. Чтобы этого не произошло, имя подпрограммы или функции, которое передается в качестве параметра другой подпрограмме или функции, надо обязательно описать с помощью оператора EXTERNAL:

```
EXTERNAL  SIN
  . . .
T = ROOT( SIN, 3.0, 4.0 )
```

В качестве формальных параметров в Фортране могут использоваться не только простые переменные и имена подпрограмм и функций, но и массивы. Размер формального параметра-массива

также может быть формальным параметром. В этом случае допускается описание массива в виде

```
REAL A(N)
```

Приведем в качестве примера подпрограмму, которая изменяет знак всех элементов вектора на противоположный:

```
      SUBROUTINE INV(A,N)   !инвертировать <вх:A,N>
      REAL A(N)             !.дано:A:вектор R(1..N),
      IF(N.LT.1)CALL OTKAZ  !.____ N ≥ 1
      DO 10 I=1,N           !. цикл ∀ I ∈ 1..N
        A(I) = -A(I)        !. .  A(I) := -A(I)
10    CONTINUE              !. конец цикла
      END                   !конец программы
```

Использованную здесь подпрограмму OTKAZ будем считать встроенной подпрограммой Фортрана, вызов которой заставляет Фортран-машину сообщить о возникновении ситуации **отказ** и прекратить выполнение Фортран-программы.

Функции. Функция является программой, вырабатывающей значение. В Фортране вырабатываемое функцией значение может иметь только предопределенный тип. Выработать в качестве значения массив или имя другой функции нельзя. Тип вырабатываемого функцией значения можно задать явно, в первой строке функции в виде

```
LOGICAL   FUNCTION ⟨имя⟩ (⟨формальные параметры⟩)
INTEGER   FUNCTION ⟨имя⟩ (⟨формальные параметры⟩)
REAL      FUNCTION ⟨имя⟩ (⟨формальные параметры⟩)
COMPLEX FUNCTION ⟨имя⟩ (⟨формальные параметры⟩)
```

Можно, однако, явно тип и не задавать, а пользоваться IJKLMN/Q-правилом и писать просто

```
FUNCTION ⟨имя⟩ (⟨формальные параметры⟩)
```

В отличие от подпрограмм, функции в Фортране обязаны иметь параметры, поэтому даже если у функции (например, у «стек пуст») нет ни одного аргумента, его приходится добавлять. Для такого фиктивного параметра рекомендуется использовать имя NLP (от слов NiL Parameter) *).

Для того чтобы при реализации функции указать ее значение, в Фортране вместо слова «ответ» пишется имя функции без скобок и аргументов, например:

*) Другая, легко запоминающаяся расшифровка NLP — неопознанный летающий параметр.

```
      FUNCTION NUMPOL(A,N) !число положительных
      REAL A(N)            !.дано:A:вектор R(1..N),
      IF(N.LT.1)CALL OTKAZ !.___ N ≥ 1
      NUMPOL=Ø             !. ответ := Ø
      DO 1Ø I=1,N          !. цикл ∀ I ∈ 1..N
        IF(A(I).GT.Ø)      !. . если A(I)>Ø то
     @    NUMPOL=NUMPOL+1  !. . . ответ := ответ+1
1Ø    CONTINUE             !. конец цикла
      END                  !конец программы
```

Константы. Задать константу в Фортране нельзя. Вместо этого можно задать с помощью специального оператора DATA начальные значения некоторых объектов и потом эти объекты не менять. Например, оператор

```
      DATA
     @        E/ 2.718281 /, PI/ 3.141593 /
     @,       A/ 1.1, 2.1, 3.1, 4.1
     @,          1.2, 2.2, 3.2, 4.2
     @,          1.3, 2.3, 3.3, 4.3 /
```

задает начальные значения простых переменных E, PI и массива A. Значение массива задается «по столбцам», т. е. сначала меняется первый индекс, затем второй и т. д.

Ввод/вывод. Подсистема ввода/вывода в Фортране является весьма обширной, обладает массой самых разнообразных возможностей. Мы, однако, ограничимся лишь шестью операторами, достаточными для так называемого *бесформатного* ввода/вывода на диск и на терминал:

CALL ASSIGN (⟨канал⟩,⟨имя файла⟩)
READ (⟨канал⟩,*, END = ⟨метка⟩, ERR = ⟨метка⟩) ⟨что⟩
WRITE (⟨канал⟩,*) ⟨что⟩
CALL CLOSE (⟨канал⟩)
TYPE *, ⟨что⟩
ACCEPT *, ⟨что⟩

Для ввода/вывода в Фортране есть специальные исполнители — *каналы*, заименованные числами от 1 до 99. Каждый канал имеет, среди прочих, предписания «начать работу»/«кончить работу» (ASSIGN/CLOSE) и предписания «прочесть»/«записать» (READ/WRITE). При начале работы с каналом указывается имя файла, с которым будет вестись работа. Между предписаниями «начать работу»/«кончить работу» разрешаются либо только предписания «прочесть», либо только предписания «записать».

В системе RSX11M на ЭВМ СМ-4 для ввода/вывода в файлы на диске обычно используются каналы 1—4, а для ввода/вывода

на терминал — операторы TYPE (вывести) и ACCEPT (ввести). Примеры использования этих операторов будут приведены ниже.

Примеры перекодировки программ на Фортран. Начнем с двух примеров законченных односекционных программ, каждая из которых читает информацию из файла, обрабатывает ее и записывает результаты работы в другой файл.

Значение в точке производной миогочлена, заданного по убывающим степеням, над комплексным полем.

```
Cпрограмма производная многочлена заданного по
C           убывающим степеням над комплексным полем
Cдано     :!В файле "MEXMAT.DAT" - комплексные числа
C          !Z, AN, ..., A1, A0 (одно число в строке)
Cполучить:!В файле "MEXMAT.RES" исходные данные и
C          !результат - значение производной P'(Z).
Cидеи реализации:
C     вычислим P'(Z) индуктивно, по схеме Горнера
C-------------------------------------------------------
      IMPLICIT COMPLEX (C)    !тип комплексное число
      CALL ASSIGN(1,'MEXMAT.DAT') !K1.начать работу
      CALL ASSIGN(2,'MEXMAT.RES') !K2.начать работу
      WRITE(2,*) 'Вычисление '  !K2.вывод заголовка
     @,'производной многочлена по схеме Горнера'
C
C
CXXXX Ввод/вывод точки Z :
C
      READ (1,*) CZ                !K1.прочесть<вых:Z>
      WRITE(2,*) 'Точка Z =',CZ !K2.вывести <вх: Z>
C
CXXXX Ввод/вывод коэффициентов многочлена P,
C                                   вычисление P'(Z):
C
      WRITE(2,*)'Коэффициенты многочлена в порядке'
     @,' убывания степеней:'
      CDP = CMPLX( 0.0, 0.0 )     ! DP := 0.0 + 0.0*i
      CP  = CMPLX( 0.0, 0.0 )     ! P  := 0.0 + 0.0*i
 1    CONTINUE                    ! цикл ∀ A ∈ K1
      READ (1,*,END=11) CA        ! . выполнять
      WRITE(2,*) CA               ! .  K2.вывести <A>
          CDP = CDP*CZ + CP       ! .  DP := DP*Z + P
          CP  = CP *CZ + CA       ! .  P  := P *Z + A
      GOTO 1                      ! конец
 11   CONTINUE                    ! цикла
```

```
C
C
CXXXX Вывод результата и конец работы с каналами:
C
      WRITE(2,*) 'DP(Z) =',CDP  ! К2.вывести <DP>
      CALL CLOSE( 2 )           ! К2.кончить работу
      CALL CLOSE( 1 )           ! К1.кончить работу
      END                       !конец программы
```

Поясним работу операторов READ и WRITE на этом примере. Каждый из этих операторов при каждом выполнении читает или записывает одну строку. Количество чисел в этой строке равно числу переменных в операторе. Например, оператор

```
READ (1,*,END=11) CA
```

пытается прочесть одно комплексное число в переменную CA. Если в файле с данными непрочитанных строк уже нет, то произойдет переход к оператору с меткой 11.

В операторе

```
WRITE (2,*) 'Точка Z =', CZ
```

перед именем переменной CZ указан текст. Этот текст будет помещен в начало очередной строки файла перед значением переменной CZ.

Число локальных максимумов числовой последовательности

```
Спрограмма число локальных максимумов числовой
C                                   последовательности
Сдано     :!В файле "MEXMAT.DAT"- последовательность
C          !чисел (одно число в строке)
Сполучить:!В файле "MEXMAT.RES" исходные данные и
C          !результат - число локальных максимумов
C          !(элемент называется локальным максиму-
C          !мом, если у него нет соседа, большего
C          !чем он сам)
C
Сидеи реализации: вычислим функцию индуктивно:
C   F = ( Ф, последний элемент, последний элемент
C                         является локальным максимумом)
C   F(x) = ( 1, x, да )  !F определено только на Ω1
C-------------------------------------------------------
      IMPLICIT LOGICAL (Q)          !тип да/нет
C
```

```
      CALL ASSIGN(1,'MEXMAT.DAT') !К1.начать работу
      CALL ASSIGN(2,'MEXMAT.RES') !К2.начать работу
C
      WRITE(2,*) 'Вычисление числа локальных ',
     @                                'максимумов'
C
      NMAXS = 0                   !Ф(А) := 0
C
      READ( 1, *, END=21 ) X      !если ω не пуста то
      NMAXS = 1                   !.
      XLAST = X                   !. F(x) := (1,x,да)
      QLAST = .TRUE.              !.
 1    CONTINUE                    !. цикл ∀ X ∈ К1
      READ  (1,*,END=11) X        !. . выполнять
      WRITE (2,*) X               !. .   К2.вывести <X>
      IF((     QLAST.AND.XLAST.EQ.X).OR.
     @   (.NOT.QLAST.AND.XLAST.LE.X)) NMAXS=NMAXS+1
       QLAST = (XLAST.LE.X)
       XLAST = X                  !. .
      GOTO 1                      !. конец
 11   CONTINUE                    !. цикла
 21   CONTINUE                    !конец если
C
      WRITE(2,*) 'число локальных максимумов = ',
     @                                 NMAXS
C
      CALL  CLOSE( 2 )            ! К2.кончить работу
      CALL  CLOSE( 1 )            ! К1.кончить работу
      END                         !конец программы
```

Фрагменты программ на Фортране. Ниже приведены фрагменты программ на Фортране—реализации отдельных подпрограмм и функций. Некоторые примеры использования этих подпрограмм и функций будут приведены ниже, при описании механизма передачи параметров.

Суммирование элементов массива.

```
      SUBROUTINE SUM(A,N,S)!сумма <вх:A,N,вых:S>
      REAL A(N)            !.дано:A:вектор R(1..N),
      IF(N.LT.0)CALL OTKAZ !.____ N ≥ 0
      S = 0.0              !. S.установить в 0
      IF(1.GT.N) GOTO 11   !. цикл ∀ I ∈ 1..N
      DO 10 I=1,N          !. . выполнять
         S = S + A(I)      !. .   S := S + A(I)
 10   CONTINUE             !. конец
 11   CONTINUE             !. цикла
      END                  !конец программы
```

Вычисление следа матрицы.

```
      FUNCTION TRACE(A,N,NMAX) !след <вх:A,N>:число
      REAL A(NMAX,NMAX)        !.дано: A : R(1..N,1..N)
      IF(N.LT.1)CALL OTKAZ     !.____  N ≥ 1
      TRACE = Ø.Ø              !. ответ.установить в Ø
      IF(1.GT.N) GOTO 11       !. цикл ∀ I ∈ 1..N
      DO 1Ø I=1,N              !. .выполнять
       TRACE=TRACE+A(I,I)      !. . ответ:=ответ+A(I,I)
1Ø    CONTINUE                 !. конец
11    CONTINUE                 !. цикла
      END                      !конец программы
```

Вычисление корня непрерывной функции F на отрезке изоляции корня методом деления пополам.

```
      FUNCTION ROOT(F,X,Y) !корень <F> на <X,Y>
      IF(F(X)*F(Y).GE.Ø.Ø) !.дано: F(X)*F(Y)<Ø.Ø
     @           CALL OTKAZ !.-----
      A = X                !. A := X
      B = Y                !. B := Y
1Ø    CONTINUE             !. цикл пока B-A>Ø.ØØ1
      IF(B-A.LE.Ø.ØØ1)GOTO 11 ! выполнять
      T = (A+B)/2.Ø           !  T := (A+B)/2.Ø
      IF(F(A)*F(T).LE.Ø.Ø)B=T ! ..............
      IF(F(B)*F(T).LE.Ø.Ø)A=T ! ..............
      GOTO 1Ø              !. конец
11    CONTINUE             !. цикла
      ROOT = (A+B)/2.Ø     !. ответ := (A+B)/2.Ø
      END                  !конец программы
```

Вычисление интеграла функции на отрезке методом трапеций.

```
      FUNCTION TRA(F,A,B,N)!интеграл функции <вх:F>
C                          !. на отрезке <вх:A,B>
C                          !. с числом трапеций <N>
      IF(N.LT.1)CALL OTKAZ !.дано: N ≥ 1
C                          !.-----
      DX = (B-A)/N         !. дельта_X := (B-A)/N
      S = F(A)/2.Ø         !. сумма    := F(A)/2.Ø
      X = A                !. X        := A
      DO 1Ø I=1,N          !. цикл ∀ I ∈ 1..N
        X = X+DX           !. . выполнять
        S = S+F(X)         !. .   ...
1Ø    CONTINUE             !. конец цикла
      TRA=( S-F(B)/2.Ø )*DX!. ответ := ...
      END                  !конец программы
```

Число минимальных элементов числового вектора.

```
      FUNCTION NMINS(A,N)   !число минимумов<вх:A,N>
      REAL A(N)             !.дано:A:вектор R(1..N),
      IF(N.LT.1)CALL OTKAZ  !.____ N ≥ 1
      NMINS = 1             !. число минимумов := 1
      AMIN  = A(1)          !. минимум := A(1)
      IF( 2.GT.N ) GOTO 11  !. цикл ∀ I ∈ 2..N
      DO 10 I=2,N           !. . выполнять
C
        IF( A(I).EQ.AMIN ) NMINS=NMINS+1
        IF( A(I).LT.AMIN ) NMINS=1
        IF( A(I).LT.AMIN ) AMIN =A(I)
C
 10   CONTINUE              !. конец
 11   CONTINUE              !. цикла
      END                   !конец программы
```

Вычисление SIN(X) с заданной точностью. Будем вычислять $\sin(x)$ с заданной точностью EPS как частичную сумму ряда

$$S = \sum_{i=1}^{n} a_i, \quad \text{где} \quad a_i = (-1)^{i-1} \frac{x^{2i-1}}{(2i-1)!}, \quad |a_n| < \text{EPS}.$$

Члены ряда a_i будем вычислять рекуррентно по формулам

$$a_1 = x, \quad a_i = -a_{i-1} \frac{x^2}{(2i-2)(2i-1)}.$$

```
      FUNCTION SINEPS(X,E) !синус от <вх:X>
C                          !    с точностью <вх:E>
      I = 1                !
      A = X                ! ...
      S = X                !
      CONTINUE             !цикл
      IF(ABS(A).LT.E)GOTO 11!. пока ABS(A) ≥ E
C                          !. выполнять
        I = I + 1
        A = -A*X**2 / ((2*I-2)*(2*I-1))
        S = S + A
      GOTO 1               !конец
 11   CONTINUE             !цикла
      SINEPS = S           ! ответ := S
      END                  !конец программы
```

Произведение двух матриц.

```
      SUBROUTINE MATMUL( A,B,C,N,K,M )
      REAL A(N,K), B(K,M), C(N,M)
      IF(1.GT.N .OR. 1.GT.K .OR. 1.GT.M) CALL OTKAZ
        DO 10 IN=1,N
        DO 20 IM=1,M
          C(IN,IM) = 0.0
          DO 30 IK=1,K
            C(IN,IM) = C(IN,IM) + A(IN,IK)*B(IK,IM)
30        CONTINUE
20      CONTINUE
10      CONTINUE
      END
```

Индекс максимального элемента вектора.

```
      FUNCTION INDMAX(A,N) !индекс максимума <A,N>
      REAL A(N)            !.дано:A:вектор R(1..N),
      IF(1.GT.N)CALL OTKAZ !._____ N ≥ 1
      INDMAX = 1           !. ответ:=1
      DO 10 I=1,N          !. цикл ∀ I ∈ 1..N
      IF(A(I).GT.A(INDMAX))!. . если A(I)>A(ответ)
     @              INDMAX=I!. . . то ответ:=I
C                          !. . конец если
 10   CONTINUE             !. конец цикла
      END                  !конец программы
```

Упорядочение элементов вектора по возрастанию.

```
      SUBROUTINE SORT(A,N) !упорядочение вектора
      REAL A(N)            !.дано:A:вектор R(1..N),
      IF(1.GT.N)CALL OTKAZ !.------ N ≥ 1
C получить: элементы в векторе A переставлены так,
C                                 что A(1)≤A(2)≤...≤A(N)
C
      DO 10 I=N,1,-1       !. цикл ∀I∈1..N реверс
C
C инв: A(K)≤A(I+1) ∀K≤I  и  A(I+1)≤A(I+2)≤...≤A(N)
C
        J = INDMAX(A,I)    !. .J:=индекс максимума
        X   = A(I)         !. .   в вырезке A(1..I)
        A(I) = A(J)        !. .обменять I-ый и J-ый
        A(J) = X           !. .  элементы вектора A
 10   CONTINUE             !. конец цикла
      END                  !конец программы
```

Некоторые вопросы устройства Фортран-машины. Фортран-машина — Универсальный Выполнитель программ на Фортране — реально представляет собой ЭВМ с соответствующим программным обеспечением (компилятором с языка Фортран, редактором связей, загрузчиком и др.). Работа Фортран-машины состоит из двух этапов: подготовки программы и ее выполнения. В процессе подготовки производится компиляция текста программы с языка Фортран в коды ЭВМ, редактирование связей и размещение в памяти ЭВМ полученной программы в кодах. На этапе выполнения полученная программа выполняется процессором ЭВМ так же, как и любая другая программа в кодах.

Здесь важно отметить две особенности работы Фортран-машины. Во-первых, каждая секция программы на Фортране компилируется в коды *независимо от остальных секций* (это свойство принято называть *раздельной компиляцией*). Другими словами, Фортран-машина не производит никаких проверок соответствия между описанием подпрограммы или функции и ее использованием. Ниже мы подробно остановимся на эффектах, к которым приводит такой подход.

Во-вторых, после подготовки остаются адреса памяти ЭВМ и содержимое памяти по этим адресам (комбинации бит). Полученная программа в кодах, как правило, не содержит никакой информации, например, о типах объектов, расположенных в тех или иных местах памяти ЭВМ, или даже о том, что расположено в данном месте памяти — объект или команды, соответствующие какому-то оператору Фортрана.

Расположение объектов в памяти. Каждый объект Фортран-программы занимает несколько байт с последовательными адресами. В разных Фортран-машинах объект одного и того же типа может занимать разное число байт. Обычно LOGICAL занимает 1, 2 или 4 байта, INTEGER — 2 или 4, REAL — 4 байта, а COMPLEX хранится как пара значений типа REAL — вещественная, а потом мнимая части. У нас и LOGICAL, и INTEGER занимают по 2 байта.

Одномерный массив из трех элементов располагается в порядке X(1), X(2), X(3). Для многомерных массивов выполнено правило: первый индекс меняется первым, второй — вторым и т. д. Так, матрица 3 * 2 располагается в порядке X(1, 1), X(2, 1), X(3, 1), X(1, 2), X(2, 2), X(3, 2).

Слова «*расположение объектов*» на самом деле означают просто, что каждому объекту соответствует некоторый адрес памяти, а все действия с объектами выражаются в терминах адресов и содержимого памяти по этим адресам. Расположением объектов в памяти (т. е. сопоставлением объектам адресов памяти) можно

явно управлять с помощью специальных операторов Фортрана EQUIVALENCE и COMMON.

Оператор COMMON. Имена переменных в Фортране локализованы внутри секции: одно и то же имя, например A, в одной секции означает одну переменную, а в другой — другую. Можно, однако, и отождествить переменные разных секций, явно указав, в каком месте памяти нужно расположить отождествляемые объекты. Это указание задается с помощью оператора COMMON (общий):

```
COMMON / ⟨имя COMMON-блока⟩ / !объекты:
@        ⟨имена объектов⟩     ! ...
@,       ⟨имена объектов⟩     ! ...
@,              ...           ! ...
```

Имена объектов перечисляются через запятую, для массивов указывается размерность по каждому индексу. Размерность массива должна быть указана только один раз — либо в операторе COMMON, либо при явном описании типа. Рекомендуется указывать размерность в операторе COMMON, а тип вообще не описывать (пользоваться IJKLMN/Q-правилом).

Отождествление объектов происходит за счет сопоставления им одного и того же адреса памяти:

1) каждому COMMON-блоку, например блоку XXCOM, Фортран-машина ставит в соответствие адрес в памяти, т. е. если в разных секциях использованы COMMON-блоки с одним и тем же именем, то им будет соответствовать *один и тот же адрес памяти*;

2) объекты COMMON-блока с данным именем, описанные в данной секции, располагаются в памяти, начиная с адреса COMMON-блока, последовательно, в порядке их описания в операторе COMMON.

Пусть, например, и в секции 1, и в секции 2 есть оператор

```
COMMON /XXCOM/ XM, X(3), DX(3), DDX(3)
```

Тогда переменная XM секции 1 и переменная XM секции 2 будут расположены в одном и том же месте памяти Фортран-машины — в первых 4 байтах COMMON-блока XXCOM. Массив X секции 1 и массив X секции 2 будут занимать следующие 12 байтов COMMON-блока XXCOM и т. д. Таким образом, переменные XM, X, DX и DDX окажутся общими для секции 1 и секции 2. Подчеркнем, что отождествление переменных вызвано не совпадением их имен, а *совпадением адресов*. Если, например, в секции 3 написать

```
COMMON /XXCOM/ X(3), DX(3), DDX(3), XM
```

то переменная XM секции 3 будет расположена с 37 по 40 байты COMMON-блока XXCOM и, таким образом, отождествится не с XM секций 1 и 2, а с элементом DDX(3) массива DDX секций 1 и 2.

Еще один пример. Пусть в секции 1 есть оператор

```
COMMON /XXCOM/ I(2), J,K
```

а в секции 2 — оператор

```
COMMON /XXCOM/ II(2), J,K, GAMMA
```

Тогда в памяти Фортран-машины эти объекты будут расположены следующим образом (* означает байт памяти):

	← начало COMMON-блока XXCOM					
память: *	* *	* *	* *	* *	* * * *	*
секция 1:	I (1)	I (2)	J	K		
секция 2:	II (1)	II (2)	J	K	GAMMA	

Оператор EQUIVALENCE. Оператор COMMON позволяет отождествить объекты разных секций. Для отождествления объектов внутри одной секции (т. е. сопоставления им одного и того же адреса памяти) используется оператор EQUIVALENCE:

```
EQUIVALENCE
@                (⟨имя объекта⟩, ⟨имя объекта⟩)
@,               (⟨имя объекта⟩, ⟨имя объекта⟩)
@,                          ...
```

В качестве имени объекта в этом операторе может быть указан и элемент массива. Чтобы понять, какие отождествления при этом произойдут, нужно помнить, как располагаются элементы массивов в памяти. Пусть, например, в некоторой секции заданы массивы

```
REAL COORD(3), A(3,3), A1(3), A2(3), A3(3), B(9)
```

Тогда оператор

```
EQUIVALENCE
1 ...(COORD(1),X), (COORD(2),Y), (COORD(3),Z)
```

отождествит объект X с первым элементом массива COORD, Y — со вторым, а Z — с третьим. После такого отождествления вместо COORD(1) можно написать X, вместо COORD(2) — Y и пр. Дру-

гими словами, оператор EQUIVALENCE можно использовать для кодирования обозначений. Оператор

```
      EQUIVALENCE (A(1,1),A1(1)),
     @            (A(1,2),A2(1)), (A(1,3),A3(1))
```

позволяет обращаться к столбцам матрицы A по именам A1, A2 и A3.

Довольно распространенный трюк — наложение одномерного массива на двумерный и работа с одномерным массивом вместо двумерного. Например, оператор

```
EQUIVALENCE (A(1,1), B(1))
```

отождествляет 9 элементов массива B с 9 элементами массива A: A(1,1) с B(1), A(2,1) с B(2), A(3,1) с B(3), A(1,2) с B(4), A(2,2) с B(5), A(3,2) с B(6), A(1,3) с B(7), A(2,3) с B(8), A(3,3) с B(9). После этого сделать все элементы матрицы A нулевыми можно, например, с помощью следующего фрагмента программы:

```
      DO 10 I=1,9
      B(I) = 0.0
10    CONTINUE
```

Механизм передачи параметров. При вызовах подпрограмм и функций формальные параметры обрабатываются так, как если бы они были расположены, начиная с того же адреса, что и соответствующие фактические параметры. Иначе говоря, при вызовах происходит передача *адреса* фактического параметра, а не его значения. Такой механизм передачи параметров называется передачей *по ссылке.*

Ни передаваемый адрес фактического параметра, ни содержимое памяти по этому адресу не содержат никакой информации об имени, типе или даже размерах занимаемой фактическим параметром памяти. При выполнении вызванной подпрограммы предполагается, что был передан адрес объекта нужного типа. Никаких проверок соответствия параметров ни на этапе подготовки, ни на этапе выполнения программы Фортран-машина, как правило, не проводит. Приведем несколько примеров вызовов реализованных выше подпрограмм SUM и TRACE (табл. 21.1), иллюстрирующих это явление, предполагая, что в начале секции, где стоят эти вызовы, есть следующие описания:

```
REAL    A(5), B(9), E(4,2)
COMPLEX C(2)
COMMON  /COM/  E, X, Y(3), Z, N(4)
```

Таблица 21.1

Вызов	Результат выполнения вызова
CALL SUM (A, 5, S)	S = сумма всех элементов A
CALL SUM (B, 9, S)	S = сумма всех элементов B
CALL SUM (B, 8, B (9))	B (9) = сумма первых 8 элементов B
CALL SUM (E, 8, S)	S = сумма всех элементов E
CALL SUM (B, 5, S)	S = сумма первых 5 элементов B
CALL SUM (B (7), 3, S)	S = сумма последних 3 элементов B
CALL SUM (C, 2, S)	S = Re (C (1)) + Im (C (1))
CALL SUM (X, 4, S)	S = X + Y (1) + Y (2) + Y (3)
CALL SUM (Y (0), 5, S)	S = X + Y (1) + Y (2) + Y (3) + Z
T = TRACE (E, 3, 3)	T = E (1,1) + E (1,2) + X
T = TRACE (A, 2, 2)	T = A (1) + A (4)
T = TRACE (B, 3, 3)	T = B (1) + B (5) + B (9)
T = TRACE (X, 2, 2)	T = X + Y (3)

Читатель может проконтролировать себя, попытавшись понять, что получится в результате выполнения оператора

```
CALL SUM(A, 5, A(3))
```

Следующий пример более сложен. При выполнении вызова

```
CALL SUM(N, 2, 1.7)
```

содержимое 8 байт массива N будет проинтерпретировано как два вещественных числа. Сумма этих чисел будет помещена в то место памяти, где раньше располагалось число 1.7.

Часто встречающиеся ошибки. Один из самых серьезных недостатков большинства существующих Фортран-машин — скупость на отказы. Нарушение требований языка при написании программы часто не приводит к немедленному отказу — признаки неправильной работы проявляются вдали от причины ошибки, либо ошибка может оказаться вообще не обнаруженной. Наиболее часто встречающиеся и опасные ошибки — это несоответствие параметров и выход индекса за границы массива. Обычно эти ошибки приводят к непредсказуемым последствиям и искать их рекомендуется, читая тексты программ подальше от ЭВМ.

При несоответствии формальных и фактических параметров ошибки могут возникать самые неожиданные и парадоксальные. Например, если подпрограмма S(X) имеет вещественный параметр X, то вызов

```
CALL S(3)  вместо  CALL S(3.0)
```

приведет к ошибке, так как в подпрограмму S будет передан адрес A целого числа 3 (занимающего одно слово памяти), а при

выполнении подпрограммы S будет взято содержимое двух слов по адресу A и проинтерпретировано как какое-то действительное число. Заметим, что отказа в момент вызова S не произойдет — подпрограмма S *будет выполняться с каким-то, неизвестно каким, X.* Проявиться эта ошибка может существенно позже, когда установить причины ее возникновения будет очень трудно. Более того, если программа выдаст правдоподобные результаты или если при конкретных исходных данных оператор CALL S(3) выполняться не будет, то ошибка вообще может остаться необнаруженной. Такая необнаруженная ошибка — это своего рода «бомба замедленного действия» — программа, долгое время считавшаяся правильной, вдруг в каких-то ситуациях начинает вести себя совершенно непредсказуемо. А в результате взрывается управляемый этой программой космический корабль, ставится неверный диагноз больному и т. п.

Классическим примером ошибок, связанных с несоответствием параметров, является порча числа: пусть есть подпрограмма

```
SUBROUTINE S(X)   !подпрограмма записывает число
X = 0.0           !0.0 по адресу, переданному из
END               !вызывающей программы
```

Последовательность операторов

```
CALL S(1.7)
R = 1.7
```

обычно приводит к тому, что объект R становится равным 0.0: вначале в подпрограмму S передается адрес числа 1.7, подпрограмма S заменяет содержимое памяти по этому адресу на 0.0, при выполнении строки R = 1.7 объекту R присваивается содержимое того места памяти, в котором хранилось число 1.7. Но теперь в этом месте уже находится 0.0 *)!

Другая распространенная ошибка — выход индекса за границы массива. Если, например, массив L описан оператором

```
COMMON /XXCOM/ K(2), L(2), W
```

то Фортран-машина при выходе индекса за границы массива L экстраполирует его следующим образом:

		←начало COMMON-блока XXCOM									
память:	*	*	*	*	*	*	*	*	*	*	*	*
содержимое:			K(1)	K(2)	L(1)	L(2)	W					
экстраполяция:	L(−2)		L(−1)	L(0)	L(1)	L(2)	L(3)		L(4)			

*) Как сказал Козьма Прутков, «если на клетке слона прочтешь надпись «ВУЙВОЛ», — не верь глазам своим»!

Таким образом, выход индекса за границы может привести к разрушению в памяти других объектов и даже команд программы.

ЗАДАЧИ И УПРАЖНЕНИЯ

В упражнениях 1—10 с помощью односекционных программ по образцу программы вычисления производной многочлена над комплексным полем (исходные данные и результат — в файлах ".DAT" и ".RES") определите

1. Максимум из чисел A, B, C.
2. Сколько максимальных среди чисел A, B, C.
3. Сколько разных среди чисел A, B, C.
4. Сколько корней у уравнения $ax^2 + bx + c = 0$.
5. Сколько целых точек удовлетворяет условию $ax^2 + bx + c > 0$.
6. В скольких точках окружность $x^2 + y^2 = r$ пересекает отрезок $(x = a, y_{min} \leqslant y \leqslant y_{max})$.
7. Интервал отрицательности функции $\Phi(x) = \max(a - x, b - 2x, x + c)$.
8. Пересекаются ли две окружности на плоскости.

В задачах 9—11 стандартным прямоугольником на плоскости называется множество точек (x, y) таких, что $x_{min} \leqslant x \leqslant x_{max}$, $y_{min} \leqslant y \leqslant y_{max}$.

9. Площадь/периметр пересечения/объединения двух стандартных прямоугольников на плоскости.
10. Разность двух равных стандартных квадратов в виде объединения стандартных прямоугольников.
11*. Модуль разности площадей фигур, на которые прямая $y = ax + b$ делит стандартный прямоугольник.

В упражнениях 12—23 требуется вычислить значение некоторой функции на пространстве числовых последовательностей. Элементы последовательности лежат в файле ".DAT" по одному числу в строке.

12. Число элементов последовательности.
13. Последний элемент.
14. Есть положительные элементы.
15. Все элементы положительны.
16. Число положительных элементов.
17. Максимальный и минимальный элемент.
18. Номер первого/последнего максимального элемента.
19. Число максимальных элементов.
20. Сумма/произведение элементов.
21. Среднее арифметическое.

22. Среднее квадратичное уклонение.

23. Число локальных максимумов.

24. Выпишите общую схему реализации конструкции выбор на Фортране.

22. Реализация исполнителей на Фортране. Примеры реализации структур данных

В Фортране отсутствуют специальные структурные единицы, соответствующие понятию исполнителя. Тем не менее, подобно тому как с помощью операторов IF и GOTO можно реализовать управляющие конструкции, с помощью оператора COMMON можно реализовать работу с исполнителями. Основное отличие исполнителя от группы программ состоит в том, что у исполнителя есть глобальные объекты, общие для всех программ данного исполнителя. Используя эти глобальные объекты, различные программы исполнителя могут передавать информацию друг другу. Таким образом, для реализации исполнителей нужен оператор, который позволит ввести объекты, общие для некоторой группы программ. Таким оператором и является оператор COMMON. Напомним, что оператор COMMON в общем виде записывается так:

```
  COMMON / ⟨имя блока⟩ /          !объекты:
@          ⟨имена переменных⟩     !  ...
@,         ⟨имена переменных⟩     !  ...
@,                ...             !  ...
```

Имена переменных перечисляются через запятую. Для массивов указывается размерность по каждому индексу (если она не указана в явном описании типа).

По правилам языка Фортран описание общих для некоторых программ объектов с помощью оператора COMMON должно быть включено в каждую программу. Чтобы не писать много раз одно и то же, воспользуемся оператором INCLUDE, который включает содержимое указанного в этом операторе файла в текст программы. Общий вид этого оператора таков:

```
INCLUDE "⟨имя файла⟩"
```

Использование этого оператора не является обязательным — в Фортране IV, где этого оператора нет, можно продублировать требуемый текст в каждой программе, например, средствами экранного редактора. Использование оператора INCLUDE, однако, делает тексты более компактными, легче понимаемыми и позволяет избежать ошибок, связанных с расхождением описаний в разных программах, например, при их модификации.

При реализации исполнителя на Фортране с использованием оператора INCLUDE следует сначала перекодировать описатель исполнителя и поместить полученный текст в отдельный файл. После этого при перекодировке отдельных программ исполнителя, в каждую из них надо с помощью оператора INCLUDE включить перекодированный описатель исполнителя. Перекодировку программ исполнителя разумно поместить во второй файл, соответствующий данному исполнителю. В результате закодированный на Фортране исполнитель будет размещаться в двух файлах: в одном описатель, а в другом — кодировка программ. Мы будем придумывать исполнителю двухбуквенное латинское имя и помещать описатель в файл с этим именем и расширением ".CLS", а кодировку программ — в файл с расширением ".FTN". Например, придумав для исполнителя стек сокращение ST, поместим описатель в файл "ST.CLS", а перекодировку программ — в файл "ST.FTN".

Ограниченный стек 4-мерных комплексных векторов. Перекодируем на Фортран непрерывную реализацию стека на базе вектора, считая, что элементами стека являются 4-мерные комплексные векторы.

```
Cфайл "ST.CLS"
Cисполнитель Ограниченный стек C4-векторов (ST)
C СП:
C  1. начать работу                          STINIT
C  2. сделать стек пустым                    STNUL
C  3. стек пуст ? да/нет                    QSTNUL (NLP)
C  4. есть свободное место ? да/нет         QSTADD (NLP)
C  5. добавить элемент <вх:E> в стек         STADD ( E )
C  6. взять элемент из стека в <вых:E>       STGET ( E )
C  7. вершина стека <вых:E>                  STQTOP( E )
C  8. установить вершину стека в <вх:E>      STSTOP( E )
C  9. удалить вершину стека                  STDEL
C 10. кончить работу                         STTERM
C
```

Это первая часть перекодировки описателя исполнителя на Фортран. Хотя результат этой перекодировки целиком состоит из строк комментария, т. е. Фортраном игнорируется, ее проведение требует содержательной работы: надо придумать сокращение для имени исполнителя и имена программ исполнителя (эти имена должны содержать не более 6 символов, состоять только из заглавных латинских букв и цифр и начинаться с буквы). Будем в качестве первых двух букв имен подпрограмм использовать двухбуквенное латинское сокращение имени исполнителя, а для имен функций добавлять перед этими двумя буквами еще букву, опре-

деляющую тип вырабатываемого значения: Q — для логических функций, I — для целых и R — для действительных. Некоторые понятия в терминах Фортрана могут оказаться невыразимыми, например на Фортране нельзя реализовать программу «вершина стека», вырабатывающую и принимающую значение, — в Фортране вообще отсутствуют такие возможности. Поэтому мы разбили эту программу на две: программу «вершина стека», вырабатывающую значение, и программу «установить вершину стека», которая должна эту вершину менять. Но даже после этого программа «вершина стека» оказалась невыразимой — ведь она должна вырабатывать значение типа «4-мерный комплексный вектор», а в Фортране функции могут вырабатывать значение только одного из четырех предопределенных типов LOGICAL, INTEGER, REAL, COMPLEX. Поэтому на Фортране эту программу придется записать в виде подпрограммы с одним выходным параметром, а не в виде функции.

Далее надо закодировать константы, типы, объекты и обозначения исполнителя:

```
C
Стипы, объекты и обозначения:
C
      IMPLICIT LOGICAL (Q)          !тип "да/нет"
      DATA                          !константы:
     @        MAXIND  / 100 /       !  максинд = 100
      COMMON  /STCOM/               !объекты:
     @                NELEMS       !  мощность: 0..максинд
```

Обратите внимание, что мы включаем во все программы исполнителя не только один и тот же оператор COMMON, обеспечивающий работу с глобальными объектами исполнителя, но и строки

```
      IMPLICIT LOGICAL (Q)          !тип "да/нет"
      DATA                          !константы:
     @        MAXIND  / 100 /       !  максинд = 100
```

Таким образом, во всех программах объекты и имена функций, которые начинаются с буквы Q, будут иметь тип да/нет и во всех программах можно будет использовать вместо константы 100 имя MAXIND.

Глобальные объекты каждого исполнителя будем размещать в отдельном COMMON-блоке, имя которого состоит из двухбуквенного латинского сокращения имени исполнителя и трех латинских букв COM. В данном случае глобальные объекты исполнителя ST размещены в COMMON-блоке STCOM.

```
C используемые исполнители:
C  VE : вектор С4-векторов с индексом 1..максинд
C мнемоника некоторых Фортранных имен:
C  Q      - inQuire       INIT    - INITialize
C  NUL    - NULl          TERM    - TERMinate
C  DEL    - DELete        NELEMS  - Number of ELEMentS
Cконец описаний !-----------------------------------
```

Раздел «мнемоника некоторых фортранных имен» призван облегчить чтение программы человеком. Дело в том, что по шестисимвольному сокращению бывает очень трудно понять, исходя из чего это сокращение придумано, что затрудняет чтение и понимание программы.

Далее следует перекодировать программы исполнителя так, как было показано в предыдущем разделе:

```
C файл "ST.FTN"
C-----------------------------------------------------
      SUBROUTINE  STINIT    !Начать работу
      INCLUDE  "ST.CLS"     !
      CALL  VEINIT          !  вектор.начать работу
      NELEMS = Ø            !  мощность := Ø
      END                   !конец программы
C-----------------------------------------------------
      SUBROUTINE  STNUL     !сделать стек пустым ==
      INCLUDE "ST.CLS"      !
      NELEMS = Ø            !  мощность := Ø
      END                   !
C-----------------------------------------------------
      FUNCTION QSTNUL(NLP)  !стек пуст : да/нет  ==
      INCLUDE "ST.CLS"      !
      QSTNUL=(NELEMS.EQ.Ø)  !  мощность = Ø
      END                   !
C-----------------------------------------------------
      FUNCTION QSTADD(NLP)  !есть св.место:да/нет==
      INCLUDE "ST.CLS"      !
      QSTADD=(NELEMS.LT.MAXIND)   ! мощность<максинд
      END                   !
C-----------------------------------------------------
      SUBROUTINE STADD(E)   !добавить <вх:E> в стек
      INCLUDE "ST.CLS"      !.дано: есть св. место
      IF(.NOT.QSTADD(NLP))CALL OTKAZ
      NELEMS = NELEMS+1     !. мощность:=мощность+1
      CALL VEWRIT(NELEMS,E)!. вектор(мощность):=E
      END                   !конец программы
```

```
C-------------------------------------------------------------
      SUBROUTINE STGET(E)   !взять из стека в <E>
      INCLUDE "ST.CLS"      !.
      IF(QSTNUL(NLP))       !.дано: стек не пуст
     @          CALL OTKAZ!.-----
      CALL STQTOP( E )      !. E := вершина стека
      CALL STDEL            !. удалить вершину стека
      END                   !конец программы
C-------------------------------------------------------------
      SUBROUTINE STQTOP(E) !вершина стека : E
      INCLUDE "ST.CLS"      !.
      IF(QSTNUL(NLP))       !.дано: стек не пуст
     @          CALL OTKAZ!.-----
      CALL VEREAD(NELEMS,E)!. ответ :=
C                           !.       вектор(мощность)
      END                   !конец программы
C-------------------------------------------------------------
      SUBROUTINE STSTOP(E) !установить вершину <E>
      INCLUDE "ST.CLS"      !.
      IF(QSTNUL(NLP))       !.дано: стек не пуст
     @          CALL OTKAZ!.-----
      CALL VEWRIT(NELEMS,E)!. вектор(мощность):=E
      END                   !конец программы
C-------------------------------------------------------------
      SUBROUTINE  STDEL     !удалить вершину стека
      INCLUDE "ST.CLS"      !.
      IF(QSTNUL(NLP))       !.дано: стек не пуст
     @          CALL OTKAZ!.-----
      NELEMS = NELEMS - 1   !. мощность:=мощность-1
      END                   !конец программы
C-------------------------------------------------------------
      SUBROUTINE  STTERM    !кончить работу  ==
      INCLUDE  "ST.CLS"     !
      CALL  VETERM          !  вектор.кончить работу
      END                   !
C-------------------------------------------------------------
Cконец исполнителя !=========================
```

Вектор 4-мерных комплексных векторов. Выше мы реализовали стек ST 4-мерных комплексных векторов на базе вектора VE 4-мерных комплексных векторов.

Язык Фортран не содержит встроенных способов конструирования для создания таких структур, поэтому реализуем исполнителя VE:

```
C файл "VE.CLS"
C исполнитель вектор С4-векторов (VE)
C  1. начать работу                          VEINIT
C  2. элемент с индексом <вх:I> : E          VEREAD(I,E)
C  3. установить элемент <вх:I> в <E>        VEWRIT(I,E)
C  4. кончить работу                         VETERM
C
C типы, объекты и обозначения:
C
      IMPLICIT LOGICAL (Q)        !тип "да/нет"
      COMPLEX  E(4)               !тип "С4-вектор"
      DATA                        !константы:
     @        MAXIND  / 100 /     ! максинд     = 100
     @,       NDIM    /   4 /     ! размерность =   4
      COMPLEX    VECTOR           !
      COMMON  /VECOM/             !объекты:
     @           VECTOR(4,100)    ! вектор С4-векторов
C                                 !  с индексом 1..100
Cконец описаний !-------------------------------
```

Обратите внимание на строку

```
COMPLEX E(4)                    !тип "С4-вектор"
```

Поскольку эта строка включена в описатель исполнителя, то далее с помощью оператора INCLUDE она будет вставлена во все программы исполнителя. Таким образом, если в какой-то программе встретится имя E, то Фортран, в соответствии с явным описанием имени, будет считать объект E одномерным массивом (вектором) из 4-х элементов типа COMPLEX (комплексное число).

```
C файл "VE.FTN"
C-----------------------------------------------
      SUBROUTINE VEINIT         !начать работу ==
      INCLUDE "VE.CLS"          !      ничего не делать
      END                       !
C-----------------------------------------------
      SUBROUTINE VEREAD(I,E)    !элемент <вх:I> : E
      INCLUDE "VE.CLS"          !.
      IF(1.GT.I.OR.I.GT.MAXIND) !.дано: 1≤I≤максинд
     @                CALL OTKAZ!.-----
      DO 1 J=1,NDIM             !.
         E(J) = VECTOR(J,I)     !. ответ := вектор(I)
 1    CONTINUE                  !.
      END                       !конец программы
```

Оставшиеся два предписания этого исполнителя реализуйте самостоятельно.

Вектор 3-мерных вещественных векторов. Обратите внимание, что, хотя речь все время шла о 4-мерных комплексных векторах, реализованный выше исполнитель ST на самом деле является *универсальным стеком элементов любого типа* E — сами элементы хранятся в исполнителе VE и только VE знает, каков тип этих элементов. По сути дела, стек просто оперирует с адресами объектов. Поэтому, чтобы реализовать, например, стек 3-мерных вещественных векторов, достаточно заменить исполнителя VE.

Для получения реализации вектора 3-мерных вещественных векторов достаточно в реализации исполнителя VE, не меняя текстов программ, заменить раздел «типы, объекты и обозначения» на следующий:

```
C
C типы, объекты и обозначения:
C
      IMPLICIT LOGICAL (Q)          !тип "да/нет"
      REAL     E(3)                 !тип "точка R3"
C                                   !
      DATA                          !константы:
     @          MAXIND  / 100 /     ! максинд     = 100
     @,         NDIM    /   3 /     ! размерность =   3
C                                   !
      COMMON /VECOM/                !объекты:
     @              VECTOR(3,100)   ! вектор точек R3
C                                   !  с индексом 1..100
C
```

Ограниченный динамический вектор точек плоскости.

```
C файл "DV.CLS"
C исполнитель Ограниченный динамический вектор
C                                  точек плоскости (DV)
C  1. начать работу                    DVINIT
C  2. сделать вектор пустым             DVNUL
C  3. вектор пуст ? да/нет              QDVNUL (NLP)
C  4. есть свободное место ? да/нет     QDVADD (NLP)
C  5. число элементов вектора ? Z+      IDVNEL (NLP)
C  6. добавить элемент <вх:E> в конец   DVADD ( E )
C  7. удалить элемент из конца вектора  DVDEL
C  8. элемент с индексом <вх:I> ? E     DVREAD(I,E)
```

```
C  9. установить элемент <вх:I> в <E>   DVWRIT(I,E)
C 10. кончить работу                    DVTERM
C
C типы, объекты и обозначения:
C
      IMPLICIT LOGICAL (Q)      !тип "да/нет"
      REAL E( 2 )               !тип "точка R2"
      DATA                      !константы:
     @        MAXIND /100/      !  максинд = 100
      COMMON /DVCOM/            !объекты:
     @          NELEMS          ! мощность:0..максинд
     @,         V(2,100)        ! вектор точек R2
C                                !         (1..максинд)
C
C мнемоника некоторых Фортранных имен:
C  NELEMS,NEL - Number of ELEMentS.-число элементов
C
Cконец описаний !-------------------------------

C файл "DV.FTN"
C-----------------------------------------------
      SUBROUTINE DVINIT         !начать работу
      INCLUDE "DV.CLS"          !
      NELEMS = 0                !      ==  мощность := 0
      END                       !
C-----------------------------------------------
      SUBROUTINE DVNUL          !сделать вектор пустым
      INCLUDE "DV.CLS"          !
      NELEMS = 0                !      ==  мощность := 0
      END                       !
C-----------------------------------------------
      FUNCTION QDVNUL(NLP)      !вектор пуст : да/нет
      INCLUDE "DV.CLS"          !
      QDVNUL=(NELEMS.EQ.0)      !      ==  мощность = 0
      END                       !
C-----------------------------------------------
      FUNCTION QDVADD(NLP)      !есть св.место:да/нет==
      INCLUDE "DV.CLS"          !
      QDVADD=(NELEMS.LT.MAXIND)  ! мощность<максинд
      END                       !
```

```
C-------------------------------------------------------------
      FUNCTION IDVNEL(NLP)   !число элементов: Z+
      INCLUDE "DV.CLS"       !
      IDVNEL = NELEMS        !      == мощность
      END                    !
C-------------------------------------------------------------
      SUBROUTINE DVADD(E)    !добавить<вх:E> в конец
      INCLUDE "DV.CLS"       !.
      IF(.NOT.QDVADD(NLP))   !.дано: есть св.место
     @            CALL OTKAZ!.------
      NELEMS = NELEMS + 1    !. мощность:=мощность+1
      V(1,NELEMS)=E(1)       !. вектор(мощность):=E
      V(2,NELEMS)=E(2)       !.
      END                    !конец программы
C-------------------------------------------------------------
      SUBROUTINE DVDEL       !удалить эл-т из конца
      INCLUDE "DV.CLS"       !.
      IF(QDVNUL(NLP))        !.дано: вектор не пуст
     @            CALL OTKAZ!.------
      NELEMS = NELEMS - 1    !. мощность:=мощность-1
      END                    !конец программы
C-------------------------------------------------------------
      SUBROUTINE DVREAD(I,E)!элемент <вх:I> : E
      INCLUDE "DV.CLS"       !
      IF(1.GT.I.OR.I.GT.NELEMS) !дано: 1≤I≤мощность
     @              CALL OTKAZ!
      E(1) = V(1,I)          !. ответ := вектор(I)
      E(2) =.V(2,I)          !.
      END                    !конец программы
C-------------------------------------------------------------
      SUBROUTINE DVWRIT(I,E)!установить <вх:I> в<E>
      INCLUDE "DV.CLS"       !
      IF(1.GT.I.OR.I.GT.NELEMS) !дано: 1≤I≤мощность
     @                CALL OTKAZ!
      V(1,I) = E(1)          !. вектор(I) := E
      V(2,I) = E(2)          !.
      END                    !конец программы
C-------------------------------------------------------------
      SUBROUTINE DVTERM      !кончить работу ==
      INCLUDE "DV.CLS"       !       ничего не делать
      END                    !
C-------------------------------------------------------------
C конец исполнителя !=========================================
```

Ссылочная реализация Л1-списка.

```
C файл "L1.CLS"
C исполнитель Ограниченный Л1-список точек R3  (L1)
C СП:
C  1. начать работу                          L1INIT
C  2. сделать список пустым                  L1NUL
C  3. список пуст ? да/нет                   QL1NUL (NLP)
C  4. есть свободное место ? да/нет          QL1ADD (NLP)
C
C  5. установить указатель в начало          L1BEG
C  6. указатель в конце ? да/нет             QL1END (NLP)
C  7. передвинуть указатель вперед           L1MOVF
C  8. добавить <вх:E> за указателем          L1AEAP( E )
C  9. взять за указателем в <вых:E>          L1GEAP( E )
C  10. элемент за указателем ? E             L1QEAP( E )
C  11. установить за указателем в <E>        L1SEAP( E )
C  12. удалить элемент за указателем         L1DEAP
C
C  - связать <вх:I1,I2:индекс>              L1LINK(I1,I2)
C  - захватить место <вых:I:индекс>         L1GETM(  I  )
C  - освободить место <вх:I:индекс>         L1FREM(  I  )
C
C  13. кончить работу                          L1TERM
C
C типы, объекты и обозначения:
C
      IMPLICIT LOGICAL (Q)          !тип "да/нет"
      DATA                          !константы:
     @        MAXIND /100/          !  максинд = 100
      COMMON /L1COM/                !объекты:
     @          IBP,  IAP           ! указ:запись(@до,@за)
     @,         INEXT( 100 )        ! @след( 1..максинд )
      DATA                          !обозначения:
     @        NILL1  / 1 /          !  нил списка    == 1
     @,       NILFRE / 2 /          !  нил св.места == 2
C
C используемые исполнители:
C  VE:вектор элементов типа E с индексом 1..максинд
C
C мнемоника некоторых Фортранных имен:
C  AEAP        - Add     Element After  Pointer
C  GEAP        - Get     Element After  Pointer
C  QEAP        - inQuire Element After  Pointer
```

```
C  SEAP          - Set     Element After  Pointer
C  DEAP          - Delete  Element After  Pointer
C  Q             - inQuire
C  BEG  / END    - BEGin  /  END'
C  MOVF          - MOVe Forward
C  GETM / FREM - GET Memory / FREe Memory
C  IBP  / IAP   - Index Before/After Pointer
C
Cконец описаний !-----------------------------------------


C файл "L1.FTN"
C-----------------------------------------------------------
      SUBROUTINE  L1INIT     !начать работу
      INCLUDE "L1.CLS"       !.
C
C дано    : максинд > 2
C получить: траектория "список" = нил списка
C           траектория "св.место" =
C                  нил св.места, 3, 4, ..., максинд
C
      CALL VEINIT            !. вектор.начать работу
      CALL L1LINK            !. связать
     @          (NILL1,NILL1)!.       (нил сп,нил сп)
      IBP = NILL1            !. указатель :=
      IAP = NILL1            !.       (нил сп,нил сп)
      DO 10 I=NILFRE,MAXIND-1! цикл ∀ I ∈ ...
        CALL L1LINK(I,I+1)   !. . связать( I, I+1 )
 10   CONTINUE               !. конец цикла
      CALL L1LINK            !. связать
     @        (MAXIND,NILFRE)!.      (максинд,нил св)
      END                    !конец программы
C-----------------------------------------------------------
      SUBROUTINE  L1NUL      !сделать список пустым
      INCLUDE "L1.CLS"       !.
      CALL  L1BEG            !. встать в начало
 1    CONTINUE               !. цикл
      IF(QL1NUL(NLP))GOTO 11!. .пока список не пуст
          CALL  L1DEAP       !. . удалить элемент
C                            !. .       за указателем
      GOTO  1                !. конец
 11   CONTINUE               !. цикла
      END                    !конец программы
```

```
C-----------------------------------------------------------
      FUNCTION QL1NUL(NLP) !список пуст ? да/нет ==
      INCLUDE "L1.CLS"     !
      QL1NUL = INEXT(NILL1)!   @след(нил сп)
     @             .EQ.NILL1!                  =нил сп
      END                  !
C-----------------------------------------------------------
      FUNCTION QL1ADD(NLP) !есть свободное место ==
      INCLUDE "L1.CLS"     !
      QL1ADD= INEXT(NILFRE)!   @след(нил св)
     @             .NE.NILFRE!                  ‡нил св
      END                  !
C-----------------------------------------------------------
      SUBROUTINE  L1BEG    !установить указатель
      INCLUDE "L1.CLS"     !.          в начало списка
      IBP = NILL1          !. ук.@до:=нил сп
      IAP = INEXT(NILL1)   !. ук.@за:=@след(нил сп)
      END                  !конец программы
C-----------------------------------------------------------
      FUNCTION QL1END(NLP) !указатель в конце ==
      INCLUDE "L1.CLS"     !
      QL1END=(IAP.EQ.NILL1)!   указатель.@за=нил сп
      END                  !
C-----------------------------------------------------------
      SUBROUTINE  L1MOVF   !передвинуть указатель
      INCLUDE "L1.CLS"     !.                  вперед
      IF(QL1END(NLP))      !.дано: не в конце
     @           CALL OTKAZ!.-----
      IBP = IAP            !. ук.@до:=ук.@за
      IAP = INEXT(IAP)     !. ук.@за:=@след(ук.@за)
      END                  !конец программы
C-----------------------------------------------------------
      SUBROUTINE L1AEAP(E) !добавить <вх:E> за ук.
      INCLUDE "L1.CLS"     !.
      IF(.NOT.QL1ADD(NLP)) !.дано: есть св.место
     @           CALL OTKAZ!.-----
      CALL L1GETM(I)       !. захватить место <I>
      CALL L1LINK(IBP,I)   !. связать <ук.@до, I>
      CALL L1LINK(I,IAP)   !. связать <I, ук.@за>
      IAP = I              !. ук.@за := I
      CALL VEWRIT(I,E)     !. вектор(I) := E
      END                  !конец программы
C-----------------------------------------------------------
```

```
      SUBROUTINE L1GEAP(E) !взять элемент за
      INCLUDE "L1.CLS"     !.  указателем в <вых:E>
      IF(QL1END(NLP))CALL OTKAZ   !дано: не в конце
      CALL L1DEAP( E )     !. E:=эл-т за указателем
      CALL L1DEAP          !. удалить за указателем
      END                  !конец программы
C-----------------------------------------------------
      SUBROUTINE L1DEAP(E) !элемент списка за
      INCLUDE "L1.CLS"     !.          указателем : E
      IF(QL1END(NLP))      !.дано: не в конце
     @          CALL OTKAZ!.----
      CALL VEREAD(IAP,E)   !. ответ:=вектор(ук.@за)
      END                  !конец программы
C-----------------------------------------------------
      SUBROUTINE L1SEAP(E) !установить элемент за
      INCLUDE "L1.CLS"     !.   указателем в <вх:E>
      IF(QL1END(NLP))      !.дано: не в конце
     @          CALL OTKAZ!.----
      CALL VEWRIT(IAP,E)   !. вектор(ук.@за) := E
      END                  !конец программы
C-----------------------------------------------------
      SUBROUTINE  L1DEAP   !удалить элемент списка
      INCLUDE "L1.CLS"     !.          за указателем
      IF(QL1END(NLP))      !.дано: не в конце
     @          CALL OTKAZ!.----
      I = IAP              !. I := ук.@за
      IAP = INEXT(I)       !. ук.@за := @след(I)
      CALL L1LINK(IBP,IAP) !.связать(ук.@до,ук.@за)
      CALL L1FREM( I )     !. освободить место(I)
      END                  !конец программы
C-----------------------------------------------------
      SUBROUTINE L1LINK(I1,I2) !связать <I1,I2> ==>
      INCLUDE "L1.CLS"     !
      INEXT( I1 ) = I2     !          @след(I1):=I2
      END.                 !
C-----------------------------------------------------
      SUBROUTINE L1GETM(I) !захватить место <вых:I>
      INCLUDE "L1.CLS"     !.
      IF(.NOT.QL1ADD(NLP)) !.дано: есть св.место
     @          CALL OTKAZ!.----
      I = INEXT(NILFRE)    !. I:=@след(нил св)
      CALL L1LINK          !. связать
     @     (NILFRE,INEXT(I))!.      (нил св,@след(I))
      END                  !конец программы
```

```
      SUBROUTINE L1FREM(I) !освободить место <вх:I>
      INCLUDE "L1.CLS"     !.
      CALL L1LINK          !. связать
     @     (I,INEXT(NILFRE))!.      (I,@след(нил св))
      CALL L1LINK(NILFRE,I)!. связать(нил св,I)
      END                  !конец программы
C-----------------------------------------------
      SUBROUTINE L1TERM    !кончить работу ==
      INCLUDE "DV.CLS"     !
      CALL VETERM          ! вектор.кончить работу
      END                  !
C-----------------------------------------------
Сконец исполнителя !==============================
```

ЗАДАЧИ И УПРАЖНЕНИЯ

1. Перекодируйте на Фортран описатели исполнителей «Выпуклая оболочка», «Многоугольник», «Планиметр» гл. 2.

2. Перекодируйте на Фортран реализацию исполнителя «Выпуклая оболочка».

3. Выпишите на Фортране систему предписаний исполнителя «Дек точек плоскости» и перекодируйте на Фортран реализацию исполнителя «Многоугольник».

4. Перекодируйте на Фортран исполнителя «Стереометр».

5. Считая, что рациональное число x — это вектор из двух целых элементов m и n (x = m/n), реализуйте на Фортране предписания следующего исполнителя:

```
Сисполнитель  Операции над рациональными числами
C  1. скопировать <вх:N1> в <вых:N2>  RNMOVE(N1,N2)
C  2. <вх:N1,N2> равны       :да/нет  QRNEQ  (N1,N2)
C  3. <вх:N1> меньше <вх:N2>:да/нет  QRNLT  (N1,N2)
C  4. прибавить <N1> к <вх/вых:N2>    RNADD (N1,N2)
C  5. домножить на<N1> <вх/вых:N2>    RNMULT(N1,N2)
C  6. сократить <вх/вых:N2>           RNREDU(   N2)
C  7. целая часть <вх:N1>   :целое    IRNVAL (N1)
C  8. приближенное значение<N1> :R    RRNVAL (N1)
C
C типы, объекты и обозначения:
C
        IMPLICIT LOGICAL (Q) !тип "да/нет"
        INTEGER N1(2), N2(2) !объекты N1, N2 типа
C                            !   рациональное число
Сконец описаний !-------------------------------
```

6. Придумайте систему предписаний и реализуйте на Фортране исполнителя «Операции над N-мерными векторами».

7. Перекодируйте на Фортран реализации множества на базе вектора (последовательный поиск, бинарный поиск, битовая реализация, хеширование) гл. 3.

8. Придумайте систему предписаний и реализуйте на Фортране исполнителя для работы с вещественными векторами размером 32000 элементов, из которых не более 1000 элементов отличны от нуля. Для задания каждого вектора используйте не более одного вещественного и одного целочисленного массивов длины 1000. Исполнитель должен позволять складывать векторы, умножать вектор на число, находить скалярное и векторное произведения векторов и т. п.

9. Придумайте систему предписаний и реализуйте на Фортране исполнителя для работы с вещественными матрицами размером 32000*32000 элементов, из которых не более 1000 элементов отличны от нуля. Для задания каждой матрицы используйте не более одного вещественного и двух целочисленных массивов длины 1000. Исполнитель должен позволять складывать и перемножать матрицы, умножать матрицу на число и т. п.

10. Придумайте систему предписаний и реализуйте на Фортране исполнителя для работы с кривой на плоскости, считая, что:

а) кривая класса C^0 и состоит из отрезков;

б) кривая класса C^1 и состоит из дуг окружностей;

в) кривая класса C^1 и состоит из дуг парабол;

г) кривая класса C^1 и состоит из дуг парабол, дуг окружностей и отрезков.

11. Используя исполнителя задачи 10 в качестве базового, составьте на Фортране программу, которая уменьшает ломаную, оставляя начало кривой:

а) до первого пересечения с заданной прямой;

б) до первого пересечения с заданным кругом;

в) заданной длины.

12. Используя исполнителя задачи 10 в качестве базового, составьте на Фортране программу, которая находит:

а) длину кривой;

б) интеграл от заданной функции вдоль кривой;

в) интеграл от заданной 1-формы вдоль кривой;

г) число точек пересечения кривой с заданной прямой;

д) число точек самопересечения кривой;

е) площадь замкнутой кривой без самопересечений;

ж) длину части кривой внутри заданного круга;

з) длину части кривой внутри заданного прямоугольника;

и) расстояние от заданной точки до кривой;

к) угол, под которым кривая видна из заданной точки.

ПРИЛОЖЕНИЕ 1

КРАТКОЕ НЕФОРМАЛЬНОЕ ОПИСАНИЕ ИСПОЛЬЗОВАННОГО В КНИГЕ ЯЗЫКА ПРОГРАММИРОВАНИЯ

Операционная обстановка. Предполагается, что программист работает в некоторой *среде* (обстановке) — множестве ранее созданных исполнителей и отдельных программ. Использование этих исполнителей и программ не требует никаких специальных усилий со стороны программиста — достаточно просто написать вызов соответствующей программы и указать имя исполнителя в разделе используемые исполнители. Все вновь создаваемые исполнители и программы добавляются к этому множеству автоматически, также без каких-либо специальных усилий со стороны программиста. Никакой явной выделенной «основной» программы нет — выполнять можно любые программы, в том числе и программы, реализующие предписания исполнителей.

Основные структурные единицы. Основными единицами языка являются конструкции **исполнитель** и **программа.** Конструкция **исполнитель** записывается в виде

исполнитель ⟨имя⟩:⟨тип⟩

или

исполнитель ⟨имя⟩
. СП:
. ⟨система предписаний⟩
. ⟨описания⟩
конец описаний | ----------
⟨последовательно расположенные программы⟩
конец исполнителя | ==========

Здесь и ниже в угловые скобки берутся имена понятий (метасимволы), используемые для описания языка. Раздел ⟨система предписаний⟩ в описателе исполнителя — это последовательность строк вида

⟨номер⟩.⟨заголовок⟩

или

—.⟨заголовок⟩

Конструкция программа имеет вид

программа ⟨заголовок⟩ == ⟨ответ или действия⟩

или

программа ⟨заголовок⟩
. **дано** : ⟨утверждение⟩
. **получить:** ⟨утверждение⟩
. ⟨описания⟩
. --------
. ⟨действия⟩
конец программы

Если программа не вырабатывает значения, то ⟨заголовок⟩ — это просто имя программы с указанием формальных параметров. Параметры могут располагаться между любыми словами имени программы, а также до или после имени и заключаются в скобки — круглые или угловые, по желанию программиста. Для каждого параметра можно указать его тип (если тип не указан, он определяется из контекста), а также является ли параметр входным, выходным или входно-выходным.

Для программ, вырабатывающих значение, в заголовке после имени и формальных параметров через двоеточие (:) указывается тип вырабатываемого значения. Внутри таких программ в числе ⟨действий⟩ должен встречаться оператор

ответ := ⟨вырабатываемое значение⟩

Для программ, вырабатывающих и принимающих значение, тип значения указывается после двойного двоеточия (::), а внутри должна встречаться строка вида

ответ == ⟨объект⟩

где метасимвол ⟨объект⟩ обозначает имя глобального объекта, его компоненту или вырезку, или имя программы, вырабатывающей и принимающей значение.

При краткой форме записи программы служебное слово **ответ** не пишется, а вырабатываемое значение, имя глобального объекта и др. записывается сразу после знака ==.

Управляющие конструкции. Конструкции выбора (ветвления):

если ⟨условие⟩	**выбор**
. **то** ⟨действия⟩	. {**при** ⟨условие⟩ ⇒ ⟨действия⟩}
. [**иначе** ⟨действия⟩]	. [**иначе** ⇒ ⟨действия⟩]
конец если	**конец выбора**

Здесь и далее в квадратные скобки заключаются части конструкции, которые могут отсутствовать, а фигурные скобки означают повторение содержимого нуль или более раз.

Конструкции циклов (повторения) имеют вид:

цикл
. [**инв:** ⟨утверждение⟩]

. [⟨итератор⟩]
. [**пока** ⟨условие⟩]
. **выполнять**
. ⟨действия⟩
конец цикла

где ⟨итератор⟩ — это либо «N **раз**», либо «**для каждого** x **из** M [**реверс**]» (вместо служебных слов **для каждого** и **из** можно использовать знаки ∀ и ∈). В качестве N может быть указано выражение. Если значение этого выражения меньше либо равно нулю, то тело цикла (действия между словами **выполнять** и **конец цикла**) не выполняется ни разу. В качестве M могут быть указаны имя структуры (объекта или исполнителя типа **стек**, **дек**, **список** и др.) или тип **отрезок** (точнее тип, сконструированный с помощью способа конструирования **отрезок**, или его имя):

∀x ∈ ***типа*** ⟨отрезок⟩
∀x ∈ ***стека*** ⟨имя стека⟩
∀x ∈ ***очереди*** ⟨имя очереди⟩
∀x ∈ **начала** ***дека*** ⟨имя дека⟩
∀x ∈ **конца** ***дека*** ⟨имя дека⟩
∀x ∈ ***непрочитанной части последовательности*** ⟨имя⟩
∀x ∈ ***части списка*** ⟨имя списка⟩ **за** ***указателем***
∀x ∈ ***части списка*** ⟨имя списка⟩ **до** ***указателя***
∀x ∈ ***множества*** ⟨имя множества⟩

(слова, выделенные полужирным курсивом, можно не писать).

Всюду выше ⟨действия⟩ — это последовательность операторов языка и управляющих конструкций, ⟨действия⟩ которых в свою очередь могут содержать управляющие конструкции и т. д. К операторам языка относятся операторы

утв: ⟨утверждение⟩
ничего не делать
отказ
выход из цикла
выход из программы

а также присваивание

⟨объект⟩ := ⟨выражение⟩

и вызов программы, не вырабатывающей значение

[⟨имя исполнителя⟩.] ⟨имя программы с фактическими параметрами⟩

Если в вызове программы ⟨имя исполнителя⟩ не указано, то вызывается программа того же исполнителя, к которому относится

и программа, выполнявшаяся до вызова. Фактические параметры располагаются в тексте имени программы также, как и формальные. Для каждого параметра можно указать его тип, а также является ли параметр входным, выходным или входно-выходным.

Записываемые в **дано/получить**, инвариантах и утверждениях (**инв** и **утв**) ⟨утверждения⟩ — это произвольные выражения, принимающие значения типа **да/нет**, которые, однако, в отличие от ⟨условий⟩ могут в качестве логических элементов включать фрагменты вида

⟨объект⟩ : ⟨тип⟩

Такие фрагменты служат как контекстом, исходя из которого определяется тип объекта, так и средством контроля. Важно заметить, что «проверка» этих фрагментов происходит на этапе создания программы, до начала ее выполнения.

Описания. ⟨Описания⟩ в программе и в описателе исполнителя состоят из следующих разделов:

типы:
{⟨имя типа⟩ = ⟨тип⟩}
константы:
{⟨имя константы⟩ [: ⟨тип⟩] = ⟨значение константы⟩}
обозначения:
{⟨имя с формальными параметрами⟩ == ⟨что обозначает⟩}
объекты:
{⟨имя объекта⟩ : ⟨тип⟩}
идеи реализации:
⟨произвольный текст-комментарий⟩
используемые исполнители:
{⟨имя исполнителя⟩ [(⟨имя-синоним⟩)] [: ⟨тип⟩]}

Любого из разделов **типы**, **константы**, **обозначения** и др., равно как и всех их вместе, может не быть. Разделы могут повторяться (т. е. может быть несколько одноименных разделов) и идти в произвольном порядке.

Во всех разделах вместо одного имени (типа, константы, объекта и пр.) может быть указано несколько имен через запятую. Если значение константы состоит из нескольких элементарных значений, то они перечисляются в круглых скобках через запятую. Для матриц элементы перечисляются по строкам, т. е. сначала меняется второй индекс, а потом первый:

константы:
пи = 3.14
пробел : символ = " "
V : вектор Z(1:3) = (4,5,6)

M: **матрица** R(1..2,1..3) =
(1.1, 1.2, 1.3, | элементы располагаются
2.1, 2.2, 2.3) | по строкам

В правой части обозначений (после знака ==) можно указать конкретное значение, имя объекта или его компоненты, или вызов программы, например:

обозначения:

нил == 1
x == r(1)
sp(a,b) == Стереометр.скалярное произведение ⟨**вх**: a,b⟩
se(a) == Стереометр.скалярное произведение ⟨**вх**: a,e⟩

Типы. Во всех описаниях выше метасимвол ⟨тип⟩ — это либо имя типа, либо явное описание типа с помощью одного из способов конструирования, либо многоточие (...). В последнем случае тип определяется из контекста.

С содержательной точки зрения тип может быть либо предопределенным, либо сконструированным.

Предопределенные типы. Существует пять предопределенных типов: **да/нет, символ, неотрицательное целое, целое** и **действительное число** (или просто **число**). Для предопределенных типов можно использовать также обозначения **Q, S, Z+, Z** и **R** соответственно. Константы записываются обычным образом, константы-символы заключаются в двойные (") или одинарные (') кавычки по желанию программиста. Объект предопределенного типа кроме обычных значений может принимать значение **неопр**. Объекты упорядоченных типов (**S, Z+, Z, R**) могут принимать еще два идеальных значения $-\infty$ и $+\infty$.

Смысл и форма записи большинства операций над объектами и значениями предопределенных типов являются общепринятыми (:=, =, ≠, <, ≤, ≥, >, **и, или, не**, +, —, *, /). Операция возведения в степень обозначается двумя звездочками (**). Обозначение унарной операции логического отрицания (**не**) может стоять не только перед именем, но и между словами имени объекта типа **да/нет** или предписания, вырабатывающего значение типа **да/нет**. Для обозначения операции и можно использовать запятую (,). Определены также операции

.увеличить на
.уменьшить на
div (деление нацело)
mod (остаток от деления)

для арифметических типов (**Z+, Z, R**) и операции

.следующий
.предыдущий

для дискретных упорядоченных типов (**S, Z+, Z**).

В число стандартных функций языка входят общепринятые функции sqrt (квадратный корень), sin, cos, tg, ctg, arcsin, arccos, arctg, arcctg, exp (экспонента), ln (натуральный логарифм), log 10 (десятичный логарифм), log 2 (логарифм по основанию 2), min или минимум (минимум произвольного числа аргументов), max или максимум (максимум произвольного числа аргументов), sign или знак, abs или модуль (абсолютная величина), int или целое (целая часть), частное и остаток, дублирующие операции **div** и **mod** соответственно, а также функции код (получение по символу его кода в диапазоне 0..255), и символ (получение символа по его коду).

Способы конструирования типов включают **перечисление, отрезок, запись** и структуры **стек, очередь, дек, множество, нагруженное множество, последовательность, Л2-список, Л1-список, вектор, матрица, динамический вектор.** Сами способы и формы их записи изложены в разд. 7. Кроме них существуют способы коструирования **имя программы и имя исполнителя:**

объекты:
X: **имя программы** [⟨заголовок без имени⟩]
Y: **имя исполнителя**
[.⟨система предписаний⟩
конец описаний]

Задание типов объектов. Тип объекта можно описать: а) в описаниях в разделе **объекты**, б) в дано/**получить**, инвариантах и утверждениях, в) в вызовах и заголовках программ (фактические и формальные параметры соответственно). Если тип объекта вообще не описан или описан через многоточие, то он будет определен из контекста. Допустимы и многократные описания типа, если они согласуются между собой.

Разное. Всюду выше ⟨имя⟩ — это последовательность идентификаторов, разделенных пробелами или группами пробелов. Идентификатор — это последовательность букв и цифр, начинающаяся с буквы. К буквам, кроме строчных и прописных русских и латинских букв относятся знаки диез (#), подчеркивание (_) и at коммерческое (@).

При записи нескольких операторов на одной строке они разделяются точкой с запятой (;). Наличие точки с запятой в конце оператора или конструкции не является ошибкой, даже если более на этой строке ничего нет.

Символом начала комментария является символ |, концом комментария является конец строки.

Адрес 302, 303, 315
— возврата 318
— подпрограммы 318
Алгоритм однопроходный 131
Алфавит 114, 150

Байт 250, 266, 301, 303
Бит 250, 301, 303
Биты NZVC 304
Блок 250, 266
Буфер 272
— кольцевой 197

Вектор 103
— динамический 104
— циклический 197
— прерывания 320
Вершина стека 16
Вид адресации 311
Выбор 23
Вызов предписания 9
— программы 29
Вырезка 276
Выход из цикла 24

Глубина стека 16
Грамматика 151

Действие 9, 26
— алфавита 115
Дек 98—99
Декомпозиция 31
Дерево вывода (дерево разбора) 153
Диск 265

Заголовок программы 375
Запоминающее устройство — см. память
Запроцедуривание циклов 244—245
Значение объекта 78
— стационарное 116

Имя 379, 335
— объекта 78
— файла 282
Инвариант 56—57
— исполнителя 289
Индекс 103
Индуктивное расширение 117
— — минимальное 118
Интерпретация, интерпретатор 161, 327
Интерфейс внешний 330
Исполнитель 6, 9, 29, 36, 374
Итерация 53

Код объектный 326
— операции 306
— символа 266, 302
Команда перехода 309
Комментарий 13, 333
Компиляция, компилятор 154, 166, 325—326
— и интерпретация 161, 187, 327
— раздельная 352
Композиция реализаций 31
Константа 36, 40, 377
Конструкция управляющая 21, 375
Критерий индуктивности 115
Курсор 283
Кэш-память, кэш 250, 252

Локализация 79, 84

Магистраль 301, 315
Массив 342
Матрица 104
Машина виртуальная 327
Метаалфавит, метасимвол 151—152
Метка 334
Множество 99, 100
Монитор ОС 324

неопр 86
Нетерминал 153

НФБН (нормальная форма Бэкуса—Наура) 153

Обозначение 40, 378
Объект 9, 78
— глобальный 9, 79, 81
— локальный 79
— простой, простого типа 194
ОЗУ см. память оперативная
Окно 283
Оператор 376, 333
— выполняемый 336
Операция 86, 87, 378
— массовая 191
— присваивания 80
Описание, 377, 336
— исполнителя внешнее 30
— типа 87, 88
ОС (операционная система) 324
Отказ 10
Откатка 299
Очередь 98
Очистка бита 301

Память 300—301
— виртуальная 250, 254, 264
— кэш см. кэш-память
— оперативная 250, 300, 324
— постоянная 324
Параметр 9, 82, 83
— входно-выходной 9, 83
— входной 9, 82, 83
— выходной 9, 83
— фактический 20, 83
— формальный 20, 83
Переменная 342
ПЗУ см. память постоянная
Подпрограмма 29, 318, 336
Поиск 211
— бинарный 214, 215
— последовательный 211, 214
Поле 96
Пользователь 330
Последовательность 100—101
Постусловие 45
Предикат 53
Предписание 9
— вырабатывающее значение 11
— принимающее и вырабатывающее значение 97, 98
Предусловие 45
Прерывание 319
Присваивание 80
Программа 10, 21, 374
— в кодах 300
— вырабатывающая значение 84
Программа локальная 38, 76
— начальной загрузки 316, 324
— обработки прерывания 319
— основная 335
— принимающая и вырабатывающая значение 97, 98, 193
Программирование 10, 30
Процессор 300, 304
Пул 251
— буферный 282

Реализация битовая 220—221
— исполнителя 29—30
— непрерывная 194
— программы 29—30
— ссылочная 202
— структуры 189
Регистр внешнего устройства 315
— процессора 304
Редактирование, редактор 283
— программ в кодах 325
— связей 326
— синтаксически ориентированное 328—329
— текстов 326
Рекурсия 50
Роллирование 291

Секция 336
Семантика 12
Символ 150
Система операционная см. ОС
— предписаний 9, 36
— файловая 267
Слово 303
Состояние объекта 9, 78
Список 101—103, 285
Способ конструирования 85, 93, 379
— — запись 95
— — исполнителей 104
— — объектов 105
— — отрезок 94
— — перечисление 93
Среда инструментальная 329
— операционная 374
Стек 15—16, 97
— адресов возврата 319
Структура 96
— динамическая 96
— ограниченная 194
Структуры, ограниченные в совокупности 201, 210
Схема вычисления инвариантной функции 90

Схема индуктивного вычисления функции 113
— обработки информации 53
— проектирования цикла с помощью инварианта 56
Счетчик команд 304

Тело цикла 23
— — для каждого 94, 107—111
Терминал 153
Технология сверху-вниз 31
Тип 78, 85, 378
— предопределенный 86, 378
Трансляция см. компиляция

УВ (универсальный выполнитель) 19
Указатель списка 101—102
— стека 304
Условие 21, 26, 89
— корректности вызова 47
— окончания 57
Установка бита 301
Устройство внешнее 301
Утверждение 24, 46, 377

Файл 266
— загрузочный 326
Функция 335
— индуктивная 114
— стандартная 379
— хеш см. хеш-функция

Хеширование 223, 226, 235, 237—238
Хеш-функция 223, 229

Цикл 23

Экземпляр программы 52
ЭПВМ 19, 330

Ядро ОС 324
Язык 150
— порожденный грамматикой 153
— программирования 19

Кушниренко Анатолий Георгиевич
Лебедев Геннадий Викторович

ПРОГРАММИРОВАНИЕ ДЛЯ МАТЕМАТИКОВ

Редактор *Л. Г. Полякова*
Художественный редактор *Г. М. Коровина*
Технический редактор *И. Ш. Аксельрод*
Корректоры *Е. Ю. Рычагова, И. Я. Кришталь*

ИБ № 32308

Сдано в набор 03.03.87. Подписано к печати 15.04.88. Т-09564. Формат 84 × 108¹/₃₂. Бумага тип. № 2. Гарнитура литературная. Печать высокая. Усл. печ. л. 20,16. Усл. кр.-отт. 20,16. Уч.-изд. л. 24,77. Тираж 70 000 экз. Заказ № 496. Цена 1 р. 10 к.

Ордена Трудового Красного Знамени
Издательство «Наука»
Главная редакция физико-математической литературы
117071 Москва В-71, Ленинский проспект, 15

Ленинградская типография № 2 головное предприятие ордена Трудового Красного Знамени Ленинградского объединения «Техническая книга» им. Евгении Соколовой Союзполиграфпрома при Государственном комитете СССР по делам издательств, полиграфии и книжной торговли. 198052, г. Ленинград, Л-52, Измайловский проспект, 29

ИЗДАТЕЛЬСТВО «НАУКА»
ГЛАВНАЯ РЕДАКЦИЯ
ФИЗИКО-МАТЕМАТИЧЕСКОЙ ЛИТЕРАТУРЫ

117071 Москва В-71,
Ленинский проспект, 15

ГОТОВИТСЯ К ПЕЧАТИ В 1989 ГОДУ:

Смирнов А. Д. **Архитектура вычислительных систем** — 20 л. (Аннотированный темплан 1989 г., поз. 158).

Рассматривается архитектура вычислительных машин и систем от супер-ЭВМ до мини- и персональных ЭВМ. Описываются поколения машин от первого до пятого. Материал изложен в структурированной форме: от основных понятий, обзора и эволюции архитектуры вычислительных систем к логической организации и эволюции основных устройств, примерам ЭВМ и системам различных классов.

Для студентов вузов, а также специалистов в области информатики.

Предварительные заказы на данную книгу принимаются без ограничения всеми магазинами книготорга и Академкниги, распространяющими физико-математическую литературу.

1 p 10 x