Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Кафедра программного обеспечения информационных технологий

Л.А. Глухова, В.В. Бахтизин

ОСНОВЫ АЛГОРИТМИЗАЦИИ И СТРУКТУРНОГО ПРОЕКТИРОВАНИЯ ПРОГРАММ

УЧЕБНОЕ ПОСОБИЕ
по курсам
«Основы алгоритмизации и программирования»
и «Технология разработки программного обеспечения»
для студентов специальности 40 01 01
«Программное обеспечение информационных технологий»
дневной формы обучения

Минск 2003

УДК 681.3.06 (075.8)
ББК 32.973-018.2 я 73
Г 55

Р е ц е н з е н т :
зав. кафедрой информатики Минского государственного высшего радиотехнического колледжа, канд. техн. наук, доц. Ю.А. Скудняков

Глухова Л.А.

Г 55 Основы алгоритмизации и структурного проектирования программ: Учеб. пособие по курсам «Основы алгоритмизации и программирования» и «Технология разработки программного обеспечения» для студ. спец. 40 01 01 «Программное обеспечение информационных технологий» дневной формы обучения / Л.А. Глухова, В.В. Бахтизин. – Мн.: БГУИР, 2003. – 72 с.: ил.

ISBN 985-444-528-3.

В учебном пособии даны общие сведения о программном обеспечении. Приведено описание основ алгоритмизации, рассмотрены способы описания и разновидности структур алгоритмов. Даны основы теории и реализации методологии структурного программирования. Приведена методология нисходящего проектирования программ, не использующая предварительную разработку схемы алгоритма.

УДК 681.3.06 (075.8)
ББК 32.973-018.2 я 73

ISBN 985-444-528-3

СОДЕРЖАНИЕ

1. Общие сведения о программном обеспечении 4
1.1. Принцип программного управления 4
1.2. Автоматическое выполнение команд программы 5
1.3. Этапы постановки и решения задачи на компьютере 6
1.4. Назначение и классификация универсальных языков программирования 7
1.4.1. Машинно-ориентированные языки 7
1.4.2. Машинно-независимые языки 7
1.5. Системы программирования 9
2. Основы алгоритмизации 12
2.1. Алгоритм и его свойства 12
2.2. Способы описания алгоритмов 12
2.2.1. Словесное описание 12
2.2.2. Графическое описание 13
2.2.3. Запись на алгоритмическом языке 25
2.3. Разновидности структур алгоритмов 25
2.3.1. Линейный вычислительный процесс 25
2.3.2. Разветвляющийся вычислительный процесс 26
2.3.3. Циклический вычислительный процесс 28
3. Структурное программирование 35
3.1. Теория структурного программирования 35
3.2. Реализация структурного проектирования в современных языках программирования 38
3.3. Преобразование неструктурированных программ в структурированные 39
3.3.1. Метод дублирования кодов программы 39
3.3.2. Метод введения переменной состояния 44
3.3.3. Метод булева признака 49
3.4. Способы графического представления структурированных схем алгоритмов 52
3.4.1. Метод Дамке 53
3.4.2. Схемы Насси–Шнейдермана 56
4. Структурирование и проектирование программ на языке Паскаль 64
4.1. Структурирование и оформление программ на языке Паскаль 64
4.2. Методология нисходящего проектирования программы, не использующая предварительную разработку схемы алгоритма 65
Контрольные вопросы для самостоятельной подготовки 69
Литература 71

1. ОБЩИЕ СВЕДЕНИЯ О ПРОГРАММНОМ ОБЕСПЕЧЕНИИ

1.1. Принцип программного управления

Одним из основных свойств современных компьютеров является программное управление.

Сущность программного управления заключается в том, что сигналы управления работой отдельных частей компьютера вырабатываются внутри компьютера в процессе вычислений. Источником информации о требуемых типах сигналов на каждом шаге вычислений является код команды, считываемый из памяти компьютера.

Команда указывает, какую операцию и над какими данными необходимо выполнить. Последовательность команд, реализующих заданный алгоритм (т.е. заданную последовательность действий), представляет собой *программу* решения задачи на компьютере.

В современных компьютерах реализован принцип хранимой программы. Его сущность заключается в том, что программа, задающая последовательность операций, записывается в память компьютера до начала вычислений. Этот принцип сформулирован американским ученым фон Нейманом в 1945 г.

Совокупность всех команд, которые может выполнить компьютер, представляет собой *систему команд* данного компьютера. В системе команд обычно содержится более 200 команд. Команда изображается в виде двоичного кода и имеет *структуру*, приведенную на рис. 1.1.

Рис. 1.1. Структура двухадресной команды

На данном рисунке приняты следующие обозначения:

КОП – код операции; для его представления, как правило, используется 1 байт;

A1, A2 – соответственно адрес первого и второго операнда.

Операндами называются данные, которые участвуют в операции.

Таким образом, в команде, кроме кода операции, указывается, по какому адресу в памяти находится каждый из операндов. Кроме того, в команде нужно указать, по какому адресу необходимо поместить результат операции. В большинстве операций при двухадресной структуре команд (структура команд, в которой имеется два поля адреса) результат записывается на место первого операнда.

Как правило, адресная часть (А1 + А2) команды занимает от 1 до 5 байт, а вся команда – от 2 до 6 байт.

В системе команд могут быть выделены следующие основные *типы команд*:

1) арифметические (сложение, вычитание, умножение, деление);

2) логические (логические сложение и умножение, отрицание, сравнение и др.);

3) передачи управления;

4) обработки адресов (специальные);

5) ввода/вывода.

1.2. Автоматическое выполнение команд программы

Под *автоматическим выполнением команд программы* подразумевается обработка входных (исходных) данных, поступивших на вход компьютера, без участия человека. Сущность ее заключается в следующем.

Программа работы компьютера и исходные данные заносятся в память машины. Процесс вычислений начинается после запуска программы на выполнение. При этом в устройство управления (УУ) компьютера передается адрес памяти, где записана первая команда программы. УУ вырабатывает сигналы, по которым в памяти находится требуемая команда и из нее извлекается код операции (КОП). Он определяет, что должен сделать компьютер над кодами чисел, адреса которых в памяти определены командой; адрес ячейки памяти, куда надо отправить результат операции, и адрес памяти, где необходимо взять код следующей команды программы. Если анализ результата операции требует изменения хода вычислений, то УУ автоматически изменяет адрес памяти, по которому извлекается код следующей команды.

Выполнив считывание из памяти одного или двух операндов, УУ направляет их в арифметико-логическое устройство (АЛУ), где в соответствии с КОП команды производится необходимое преобразование информации. По мере переработки информации промежуточные и окончательные результаты хранятся в памяти компьютера. При необходимости последние команды программы могут определять операции передачи информации из памяти в устройство вывода информации, где информация помещается на некотором носителе информации в форме, удобной для восприятия человеком или компьютером.

1.3. Этапы постановки и решения задачи на компьютере

Можно выделить следующие *этапы* постановки и решения задачи на компьютере:

1) четкая формулировка задачи, выделение исходных данных и формы представления результатов;

2) формальная (математическая) постановка задачи – представление ее в виде уравнений, соотношений, ограничений;

3) выбор метода решения (метод решения определяется решаемой задачей);

4) разработка алгоритма решения задачи;

5) выбор структуры данных (от выбора способа представления данных зависит способ их обработки, поэтому этапы 4 и 5 взаимосвязаны);

6) собственно программирование (запись разработанного алгоритма на языке программирования);

7) тестирование и отладка программы (проверка правильности работы программы и исправление обнаруженных ошибок);

8) выполнение программы на компьютере.

Для того чтобы программа была понятна компьютеру, она должна быть составленной из последовательности элементарных операций, представленных на *машинном языке* – в виде совокупности нулей и единиц, т.е. в так называемых машинных кодах.

Программирование в машинных кодах было характерным для начального этапа развития вычислительной техники. Существенными недостатками такого программирования являются большая трудоемкость и высокая стоимость программирования, необходимость высокой квалификации программиста. В связи с этим возникла потребность в максимальном облегчении процесса общения человека с компьютером, освобождении его от необходимости описания алгоритма на машинном языке. Стала очевидной возможность и необходимость автоматизации процесса получения программ. Возникли методы автоматизации программирования, позволяющие облегчить процесс написания программ.

К основным *методам автоматизации программирования* можно отнести следующие:

1) использование языков высокого уровня, близких к естественному человеческому языку, позволяющих автоматически однозначно преобразовывать написанную на них программу в программу на языке машины, т.е. в машинные коды;

2) создание и использование библиотек стандартных программ и подпрограмм, предназначенных для реализации часто используемых функций и задач;

3) использование современных технологий программирования;

4) использование Case-средств, предназначенных для автоматизации процесса проектирования программ.

1.4. Назначение и классификация универсальных языков программирования

Основная идея автоматизации программирования заключается в отказе от написания программ непосредственно в машинных кодах. Программа пишется на некотором входном языке. Универсальные входные языки можно разделить на следующие группы:

1) машинно-ориентированные языки;
2) процедурно-ориентированные языки;
3) объектно-ориентированные языки;
4) языки четвертого поколения (4GL).

1.4.1. Машинно-ориентированные языки

Программы, написанные на этих языках, могут выполняться только на тех компьютерах, для которых разработаны соответствующие языки. Данные языки относятся к языкам низкого уровня.

Уровень языка определяется соотношением количества машинных команд, необходимых для реализации некоторой средней программы, и количества операндов языка, необходимых для написания данной программы. Чем ближе данное соотношение к единице, тем ниже уровень языка.

К языкам низкого уровня относятся языки символического кодирования, автокоды, ассемблеры.

Недостатки данной группы языков: трудоемкое программирование, машинная ориентированность, высокие требования к уровню подготовки программистов.

1.4.2. Машинно-независимые языки

Вторая–четвертая группы языков относятся к *машинно-независимым языкам высокого уровня.* Для языков высокого уровня характерно то, что их понятия и структура удобны для восприятия человеком. Написанные на них программы могут быть выполнены на любых компьютерах, имеющих транслятор с данного языка.

Процедурно-ориентированные языки

Это языки программирования, в которых действия над данными выражаются в терминах последовательностей команд [2]. Первоначально эти языки предназначались для решения конкретного класса задач – инженерных, научно-технических, обработки экономической информации, обработки списков, моделирования и т.д. Со временем большинство из языков данной

группы развилось и превратилось в достаточно мощные универсальные языки программирования. Ниже дана краткая характеристика первых версий языков данной группы.

Фортран. Разработан в 1956 г. Предназначался для решения инженерных и научно-технических задач. Предоставлял пользователю большие возможности для обработки числовых данных, но располагал бедными средствами для работы с символьными строками.

Алгол (1958 г.). Первый язык с блочной структурой. Располагал бедными средствами ввода-вывода. Предназначался для решения широкого круга математических задач.

Кобол (1959 г.). Язык, ориентированный на решение коммерческих (планово-экономических) задач. В Коболе особое место было отведено понятиям записи, файла, описанию поля, предусматривались широкие возможности для манипулирования данными.

Бейсик (1965 г.). Первоначально был разработан как язык для обучения программированию. Являлся упрощенной версией Фортрана. Предоставлял широкие средства для диалога, но имел ограниченные возможности по сравнению с другими языками высокого уровня.

ПЛ/1 (1965 г.). Сочетал в себе многие черты Алгола, Кобола и Фортрана. Являлся одним из самых универсальных языков программирования. Располагал большим числом средств, что делало его весьма сложным для изучения и использования.

Паскаль (1971 г.). Основан на Алголе. Являлся языком с блочной структурой, способствовал структурному подходу к программированию. Располагал большим набором управляющих операторов. Широко используется в настоящее время в персональных компьютерах.

Си (1972 г.). Первоначально был разработан для мини-ЭВМ, использующих операционную систему UNIX. Включал средства для проектирования на уровне ассемблера, средства для эффективного использования аппаратуры. Поэтому считался наиболее эффективным в плане скорости выполнения программы и необходимой памяти. Нашел широкое применение при написании программ вычислительного характера и программ операционных систем.

Ада (1979 г.). Представлял собой расширение языка Паскаль для больших машин. Являлся существенно структурированным языком. Был особенно удобен при использовании в системах реального времени.

Объектно-ориентированные языки

Объектно-ориентированные языки являются развитием процедурно-ориентированных языков. Поэтому они часто не выделяются в отдельную группу языков.

Объектно-ориентированные языки, при сохранении свойств процедурно-ориентированных языков, соответствуют концепциям объектно-ориентированного программирования.

К данной группе языков относятся версии языка Турбо Паскаль 5.5 и выше, Си++ и ряд других.

Языки четвертого поколения

Реализуют современные технологии визуального программирования. Автоматически генерируют исходный текст программ целиком или в виде отдельных фрагментов. К данной группе языков можно отнести, например, визуальные среды программирования Delphi, Builder C++.

1.5. Системы программирования

Система программирования – совокупность программ, описаний и инструкций, предназначенных для автоматизации процесса разработки программ.

Основное *назначение* систем программирования – максимально облегчить процесс написания программ, освободить программиста от необходимости описания алгоритма на машинном языке, предоставить возможность использования языка высокого уровня.

Состав системы программирования:

1) входной язык системы;
2) транслятор с входного языка на машинный язык;
3) редактор связей;
4) библиотеки программ;
5) средства отладки;
6) обслуживающие (сервисные) программы;
7) документация.

1. Входной язык системы

Входным языком системы программирования является один из языков программирования низкого или высокого уровня.

2. Транслятор

Транслятор – это программа, предназначенная для преобразования исходной программы, написанной на входном языке программирования, в программу на машинном языке. Существует *два вида* трансляторов:

а) *компиляторы* – трансляторы, в которых трансляция отделена от выполнения программы (транслятор компилирует рабочую программу, которая в дальнейшем может быть выполнена). К компиляторам относятся, например, трансляторы со входных языков Паскаль, Си;

б) *интерпретаторы* – трансляторы, в которых трансляция совмещена с выполнением программы. Каждый оператор языка читается, расшифровывается и выполняется. Интерпретаторы отличаются малой скоростью работы, но являются более простыми по сравнению с компиляторами. К интерпретаторам относится, например, транслятор со входного языка Бейсик.

3. Редактор связей

Современные системы программирования основаны на модульном принципе: программы оформляются в виде совокупности взаимосвязанных программ. Каждая такая программа называется *модулем*.

В соответствии с ГОСТ 19781-90 *программным модулем* (Program module) называется программа или функционально завершенный фрагмент программы, предназначенный для хранения, трансляции, объединения с другими программными модулями и загрузки в оперативную память.

Различают следующие виды программных модулей.

Исходный модуль (Source module) – это программный модуль, записанный на входном языке программирования, обрабатываемый транслятором и представляемый для него как целое, достаточное для проведения трансляции. Каждый исходный модуль транслируется независимо от других модулей.

Объектный модуль (Object module) – это программный модуль, полученный в результате трансляции. Он содержит текст программы на машинном языке и дополнительную информацию, обеспечивающую объединение этого модуля с другими независимо транслированными модулями. Объектный модуль полностью готов к редактированию связей.

Редактор связей – это программа, предназначенная для сборки и установления связей между модулями. Редактор связей создает загрузочные модули на основании одного или нескольких объектных или загрузочных модулей путем разрешения перекрестных ссылок между модулями и, при необходимости, настройки адресов.

Таким образом, в результате работы редактора связей создается еще один вид программного модуля – загрузочный модуль.

Загрузочный модуль (Load module) – это модуль, представленный в форме, пригодной для загрузки в основную память для выполнения.

4. Библиотека программ

Библиотека программ представляет собой готовые программы, предназначенные для решения распространенных задач. Программы, включенные в библиотеку, оформляются специальным образом, облегчающим их вызов, использование, передачу входных данных и результатов. Программы, включенные в библиотеку программ, вызываются для выполнения специальными командами вызова.

5. Средства отладки

Основная *цель этапа отладки* – выявление и исправление ошибок в программе. Процесс отладки состоит из многократных попыток выполнения программы на компьютере и анализа получившихся результатов.

Собственно процессу выполнения на машине предшествует трансляция программы и ее редактирование.

Большинство синтаксических ошибок обнаруживается машиной автоматически на этапе трансляции. Современные трансляторы с языков программирования выдают информацию о синтаксических ошибках (ошибках, допущенных при записи текста программы на алгоритмическом языке), указывают места ошибок и их характер.

На этапе редактирования обнаруживаются ошибки, связанные с неправильным оформлением подпрограмм, ошибки в командах вызова подпрограмм и программ из библиотеки программ.

На этапе выполнения обнаруживаются логические ошибки программы (например, деление на ноль, бесконечный цикл и т.п.).

Современные средства отладки позволяют в пошаговом режиме обнаружить и локализовать ошибку.

После того как программа становится работоспособной, проводится ее *тестирование* – проверка правильности ее функционирования на различных наборах исходных данных из диапазона их допустимых значений.

6. Системные сервисные программы

Системные сервисные программы носят вспомогательный характер и упрощают работу с системой программирования.

7. Документация

Документация представляет собой описания, связанные с компонентами 1–6 системы программирования.

2. ОСНОВЫ АЛГОРИТМИЗАЦИИ

2.1. Алгоритм и его свойства

Под *алгоритмизацией* понимается сведение задачи к последовательности этапов, выполняемых друг за другом так, что результаты предыдущих этапов используются при выполнении следующих.

Алгоритмом называется система правил, четко описывающая последовательность действий, которые необходимо выполнить для решения задачи.

Правильно разработанный алгоритм обладает следующими *свойствами*:

1. *Дискретность* – значения величин в каждый следующий момент времени получаются по определенным правилам из значений величин, имевшихся в предшествующий момент времени.
2. *Определенность (детерминированность)* – каждое правило алгоритма должно быть однозначным. Значения величин, получаемых в какой-то момент времени, однозначно связаны со значениями величин, вычисленных ранее.
3. *Результативность (конечность)* – алгоритм должен приводить к решению задачи за конечное число шагов.
4. *Массовость* – алгоритм разрабатывается в общем виде так, чтобы его можно было применить для класса задач, различающихся лишь исходными данными.

2.2. Способы описания алгоритмов

Существуют следующие способы описания алгоритмов:

1) запись на естественном языке (словесное описание);
2) изображение в виде схемы (графическое описание);
3) запись на алгоритмическом языке (составление программы).

2.2.1. Словесное описание

Существуют различные способы словесного описания алгоритмов. Они отличаются применяемыми метаязыками. В программировании *метаязыком* называется язык, предназначенный для описания языка программирования. Один из способов словесного описания приведен ниже.

При использовании данного способа для описания алгоритмов используются следующие *типовые этапы*.

1. Этап обработки (вычисления)

Этап обработки описывается в виде

V = выражение.

Здесь V – переменная.

Любые вычисления можно выполнять только на этом этапе.

Для описания этапа обработки часто используется *символ присваивания*

:=

Слева от символа (**:=**) записывается переменная, которой присваивается значение, записанное справа от этого символа. Например, запись X **:=** Y означает, что переменной X присваивается значение переменной Y.

2. Проверка условия

Этап проверки условия описывается в виде

Если условие, идти к N

Если условие выполняется, то осуществляется переход к этапу с номером (меткой) N. Если условие не выполняется, то осуществляется переход к следующему по порядку этапу.

3. Переход к этапу с номером N

Данный этап описывается в виде

Идти к N

4. Конец вычислений

Данный этап описывается в виде

Останов.

Пример 2.1. Словесное описание алгоритма решения квадратного уравнения $a*x^2 + b*x + c = 0$.

1. D **:=** $b^2 - 4*a*c$
2. Если D < 0, идти к 4
3. x1 **:=** $(-b + \sqrt{D})/(2*a)$

   x2 **:=** $(-b - \sqrt{D})/(2*a)$
4. Останов.

Недостаток словесного описания – малая наглядность.

2.2.2. Графическое описание

Графическое описание позволяет представить алгоритм в наглядной форме в виде схемы.

Схемой алгоритма называется графическое представление алгоритма, в котором этапы процесса обработки информации и носители информации

представлены в виде геометрических символов из заданного ограниченного набора, а последовательность процесса отражена направлением линий.

Для стандартизации и унификации языка схем алгоритмов в 1985 г. был принят международный стандарт **ISO 5807-85**. В 1992 г. после переработки он был принят в СССР под обозначением **ГОСТ 19.701-90** «Единая система программной документации. Схемы алгоритмов, программ, данных и систем. Условные обозначения и правила выполнения» [1]. В настоящее время данный стандарт продолжает действовать в Республике Беларусь и Российской Федерации.

В указанном стандарте регламентированы условные обозначения символов, применяемых в схемах алгоритмов, программ, данных и систем, и правила выполнения схем с использованием данных символов. Стандарт не ограничивает форму записей и обозначений, помещаемых внутри символов или рядом с ними и служащих для уточнения выполняемых функций.

Ниже кратко рассмотрены основные требования ГОСТ 19.701-90 и даны необходимые пояснения к этим требованиям.

В соответствии с данным стандартом схемы состоят из символов, краткого пояснительного текста и соединительных линий.

Схемы могут использоваться с различным уровнем детализации. Уровень детализации зависит от размеров и сложности задачи и должен быть таким, чтобы различные части схемы и взаимосвязь между ними были понятны.

В стандарте различаются следующие ***виды схем***, предназначенные для использования в программной документации.

1. **Схема данных**

Схемы данных отображают путь данных при решении задач и определяют этапы обработки, а также различные применяемые носители данных.

2. **Схема программы**

Схемы программ отображают последовательность операций в программе. Понятие схемы программы аналогично понятию схемы алгоритма. Различие может заключаться лишь в уровне детализации схемы.

3. **Схема работы системы**

Схемы работы системы отображают управление операциями и потоки данных в системе. В схеме работы системы каждая программа может изображаться более чем в одном потоке управления.

4. **Схема взаимодействия программ**

Схемы взаимодействия программ отображают путь активации программ и взаимодействий с соответствующими данными. Каждая программа в схеме взаимодействия программ показывается только один раз.

5. Схема ресурсов системы

Схемы ресурсов системы отображают конфигурацию блоков данных и обрабатывающих блоков, которая требуется для решения задач.

Символы, регламентируемые стандартом, подразделяются на следующие *группы*:

1) символы данных;
2) символы процесса;
3) символы линий;
4) специальные символы.

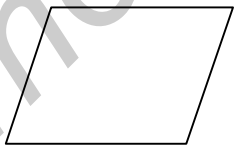
Каждая из трех первых групп в свою очередь подразделяется на *две подгруппы*:

- основные символы;
- специфические символы.

1. Символы данных

К *основным символам данных* относятся символы, не конкретизирующие носитель данных. (Носитель данных – это источник или приемник данных, например, это может быть клавиатура или дискета при вводе информации, экран дисплея или дискета при выводе информации.) В данную подгруппу входят следующие символы.

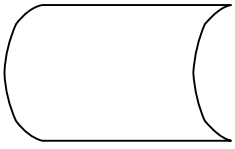
1.1. Данные



Символ отображает данные, носитель данных не определен. Это наиболее универсальный символ, с помощью которого определяется вводимая/выводимая информация в том случае, если для задачи не имеет значения, откуда информация вводится или куда она выводится.

Например, в некотором алгоритме необходимо отразить ввод значений X, У и вывод значения Z, причем конкретный источник и приемник информации неважны. В этом случае для отображения ввода/вывода данных в схеме алгоритма необходимо использовать символ «Данные».

1.2. Запоминаемые данные

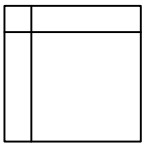


Символ отображает хранимые данные. Конкретный носитель данных не

определяется. Данный символ используется в схемах алгоритма, например, для изображения результирующей информации, которую нужно запомнить, причем тип приемника информации значения не имеет.

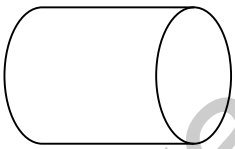
К *специфическим символам данных* относятся символы, конкретизирующие носитель входных/выходных данных. Наиболее часто из символов этой подгруппы используются следующие.

1.3. Оперативное запоминающее устройство



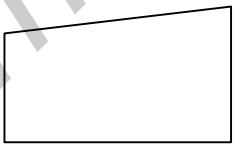
Символ отображает данные, хранящиеся в оперативном запоминающем устройстве.

1.4. Запоминающее устройство с прямым доступом



Символ отображает данные, хранящиеся в запоминающем устройстве с прямым доступом (например магнитный диск).

1.5. Ручной ввод



Символ отображает данные, вводимые вручную во время обработки с устройств любого типа (например с клавиатуры).

1.6. Дисплей

Символ отображает данные, представленные в удобной для человека форме на отображающем устройстве (например на экране дисплея).

2. Символы процесса

К подгруппе *основных символов процесса* относится единственный

символ, в наиболее общем виде отображающий функцию обработки данных любого вида (выполнение определенной операции или группы операций).

2.1. Процесс

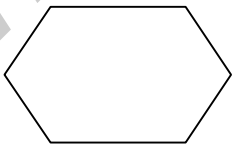
Вычислительные операции любого вида на схемах алгоритма можно изображать только с помощью данного символа.

К *специфическим символам процесса* относятся символы процесса более узкого, конкретного назначения. В данную подгруппу входят следующие символы.

2.2. Предопределенный процесс

Символ отображает процесс, состоящий из одной или нескольких операций или шагов программы, которые определены в другом месте. С помощью данного символа изображаются, например, подпрограммы или модули программ.

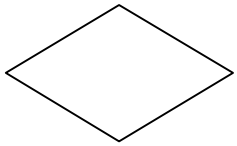
2.3. Подготовка



Символ отображает модификацию команды или группы команд с целью воздействия на некоторую последующую функцию (установка переключателя, модификация индексного регистра или инициализация программы).

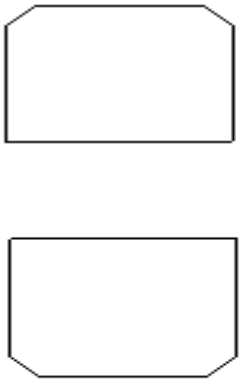
Данный символ используется, как правило, в схемах алгоритмов, изображающих работу программ, которые реализуются на языках низкого уровня. Например, программа на ассемблере изменяет значения индексного регистра или флагов микропроцессора. Данные действия на схеме алгоритма будут изображаться с помощью символа «Подготовка».

2.4. Решение



Символ отображает функцию переключательного типа, имеющую один вход и ряд альтернативных выходов, один из которых активизируется после вычисления условий, записанных внутри этого символа. Соответствующие результаты вычисления записываются рядом с линиями, отображающими эти выходы.

2.5. Граница цикла



Символ состоит из двух частей, отображающих начало и конец цикла. Обе части символа должны иметь один и тот же идентификатор. Это значит, что схеме каждого цикла должно быть присвоено имя. Это имя записывается в обеих частях символа. Условия, управляющие выполнением цикла (инициализация, завершение, приращение параметра и т.д.), помещаются внутри первой или второй части символа в зависимости от расположения операции, проверяющей условие.

Например, при изображении цикла с предусловием (цикл While) условие выполнения цикла должно быть записано в верхней части символа. При изображении цикла с постусловием (цикл Repeat) условие выхода из цикла должно быть записано в нижней части символа. При изображении цикла с параметром (цикл For) условие выполнения цикла должно быть записано в верхней части символа, поскольку данный цикл, по существу, является циклом с предусловием, а приращение параметра – в нижней части символа.

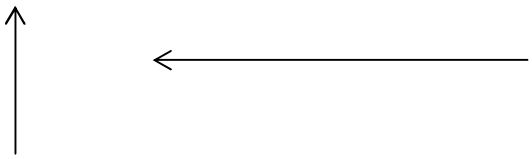
3. Символы линий

Подгруппа *основных символов линий* содержит единственный символ.

3.1. Линия

Символ отображает поток данных или управления.

При необходимости или для повышения удобочитаемости к линии могут быть добавлены стрелки.



Из подгруппы *специфических символов линий* в схемах алгоритмов наиболее широко используется следующий символ.

3.2. Пунктирная линия

В схемах алгоритмов данный символ используется для обведения выделяемого участка, а также как часть символа комментария.

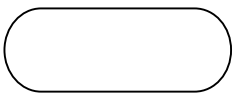
4. Специальные символы

В данную группу входят следующие символы.

4.1. Соединитель

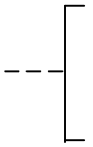
Символ отображает выход в другую часть схемы и вход из другой части этой схемы и используется для обрыва линии и продолжения её в другом месте. Соответствующие символы-соединители должны содержать одно и то же уникальное обозначение. Чаще всего в качестве таких обозначений применяются буквы или арабские цифры.

4.2. Терминатор



Символ отображает выход во внешнюю среду и вход из внешней среды (в схемах алгоритмов – это начало или конец алгоритма).

4.3. Комментарий



Символ используют для добавления комментариев (пояснительных записей). Пунктирная линия в символе комментария связана с соответствующим символом и может обводить группу символов, если комментарий относится ко всей группе. Текст комментариев помещается

справа от ограничивающей его квадратной скобки.

4.4. Пропуск

…

Символ применяется в схемах для отображения пропуска символа или группы символов, если не определены ни тип, ни число символов. Символ используется только в символах линии или между ними. Он применяется главным образом в схемах, изображающих общие решения с неизвестным числом повторений. Например, на рис. 2.1 приведено общее решение схемы циклического процесса с неизвестным числом повторений.

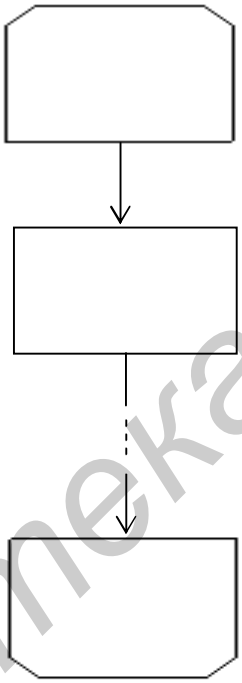


Рис. 2.1. Общее решение схемы циклического процесса с неизвестным числом повторений

Стандарт определяет ***правила применения символов в схемах***. Ниже рассмотрены основные правила, знание которых необходимо при разработке схем алгоритмов.

Символы в схеме должны быть расположены равномерно. Следует придерживаться минимального числа длинных линий.

Стандарт регламентирует форму символов, но не их конкретные размеры. Размеры символов должны позволять включать текст внутрь символа. Не должны изменяться углы и другие параметры, влияющие на соответствующую форму символов. Символы должны быть по возможности одного размера.

Символы могут быть вычерчены в вертикальной ориентации или в зеркальном изображении. Предпочтительным является изображение, приведенное в стандарте.

Внутри символа следует помещать минимальное количество текста, необходимое для понимания функции данного символа. Текст должен записываться слева направо и сверху вниз независимо от направления потока.

Например, в обоих фрагментах схем алгоритма, приведенных на рис. 2.2, вначале вычисляется значение B, а затем – значение C.

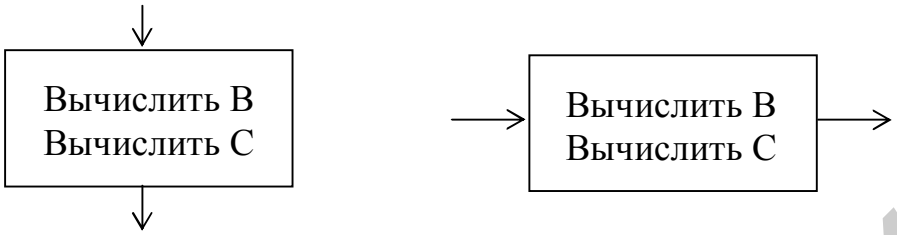


Рис. 2.2. Расположение текста в символах схемы алгоритма

Если объём текста не помещается внутри символа, следует использовать комментарий.

Если использование символов комментария загромождает схему, текст комментария следует помещать на отдельном листе и давать перекрестную ссылку на комментарий, используя символ-соединитель.

В схемах может использоваться *идентификатор символов*. Это идентификатор, который определяет символ для использования в справочных целях в других элементах документации (например, в листинге программы или в тексте пояснений к схеме алгоритма). Идентификатор символа должен располагаться слева над символом (рис. 2.3).

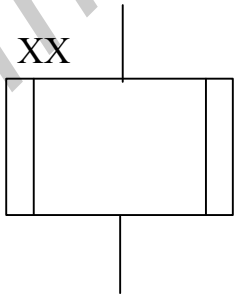


Рис. 2.3. Изображение идентификатора символа

Часто для удобства описания схем алгоритмов символы схемы нумеруются. В этом случае в качестве идентификатора символа используется номер символа в схеме.

В схемах может использоваться *описание символов*. Это любая другая информация, например, для отображения специального применения символа с перекрестной ссылкой или для улучшения понимания функции как части схемы. Описание символа должно быть расположено справа над символом (рис. 2.4).

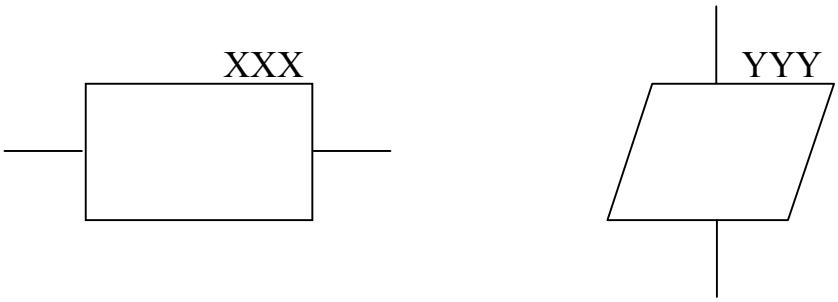


Рис. 2.4. Изображение описания символа

В схемах может использоваться *подробное представление*, которое обозначается с помощью символа с полосой для процесса или данных. Символ с полосой указывает, что в этом комплекте документации в другом месте имеется более подробное представление данного символа.

Символ с полосой представляет собой любой символ из групп символов процесса или символов данных, внутри которого в верхней части проведена горизонтальная линия. Между этой линией и верхней линией символа помещен идентификатор, указывающий имя схемы с подробным представлением данного символа (рис. 2.5).

В качестве первого и последнего символа подробного представления должны быть использованы символы «Терминатор», в которых указывается идентификатор, имеющийся в символе с полосой (см. рис. 2.5).

Символ с полосой Подробное представление

Рис. 2.5. Изображение символа с полосой и его подробного представления

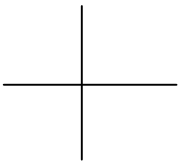
Стандарт определяет ***правила выполнения соединений в схемах***. Ниже рассмотрены основные правила, знание которых необходимо при разработке схем алгоритмов.

Потоки данных и потоки управления в схемах показывают линиями. Направление потока слева направо и сверху вниз считается *стандартным*.

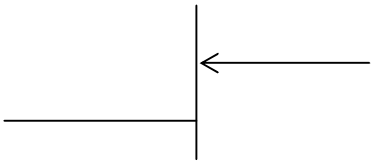
В случаях, когда необходимо внести большую ясность в схему (например, при соединениях), на линиях используются стрелки. Если поток

имеет направление, отличное от стандартного, оно должно указываться стрелками.

В схемах следует избегать пересечения линий. Пересекающиеся линии не связаны между собой, поэтому изменения направления в точках пересечения не допускается. Пересечение линий изображается следующим образом:



Две или более входящие линии могут объединяться в одну исходящую линию. В этом случае места объединения линий должны быть смещены. Например:



Линии в схемах должны *подходить к символу слева либо сверху*, а *исходить справа либо снизу*. Линии должны быть направлены к центру символа.

Для избежания излишних пересечений и слишком длинных линий или если схема состоит из нескольких страниц, линии в схемах следует разрывать.

Соединитель в начале разрыва называется *внешним соединителем*, а соединитель в конце разрыва — *внутренним соединителем*.

Для *межстраничных соединителей* совместно с символами соединителя используются символы комментария. В комментариях указываются номера страниц, на которых расположены внутренний и внешний соединители (рис. 2.6).

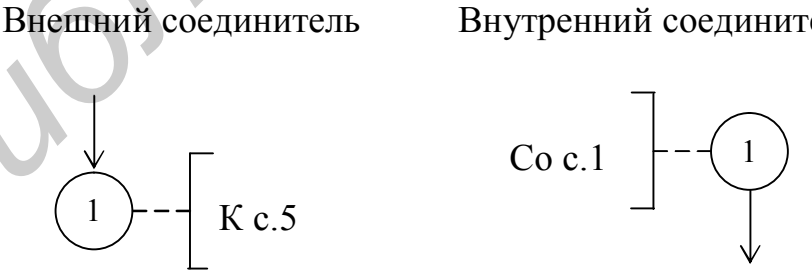


Рис. 2.6. Изображение межстраничных соединителей

В стандарте поясняются некоторые ***специальные условные обозначения***.

Для схем алгоритмов представляет интерес изображение нескольких выходов из символа (несколько выходов в схеме алгоритма может иметь лишь символ «Решение»).

Несколько выходов из символа можно показывать следующими способами (рис. 2.7):

1) несколькими линиями от данного символа к другим символам;

2) одной линией от данного символа, которая затем разветвляется на соответствующее число линий.

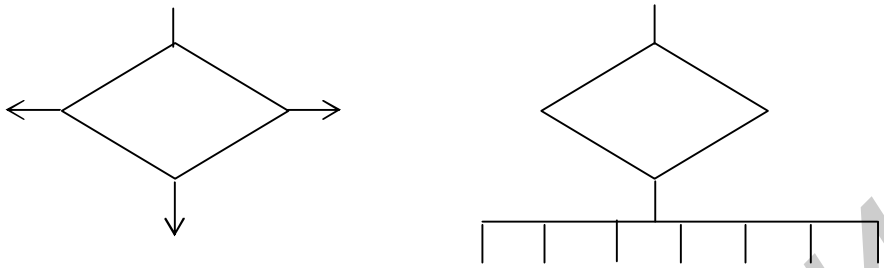


Рис. 2.7. Изображение нескольких выходов из символа

Каждый выход из символа должен сопровождаться соответствующими значениями условий, определяющими переход по данному логическому пути (рис. 2.8).

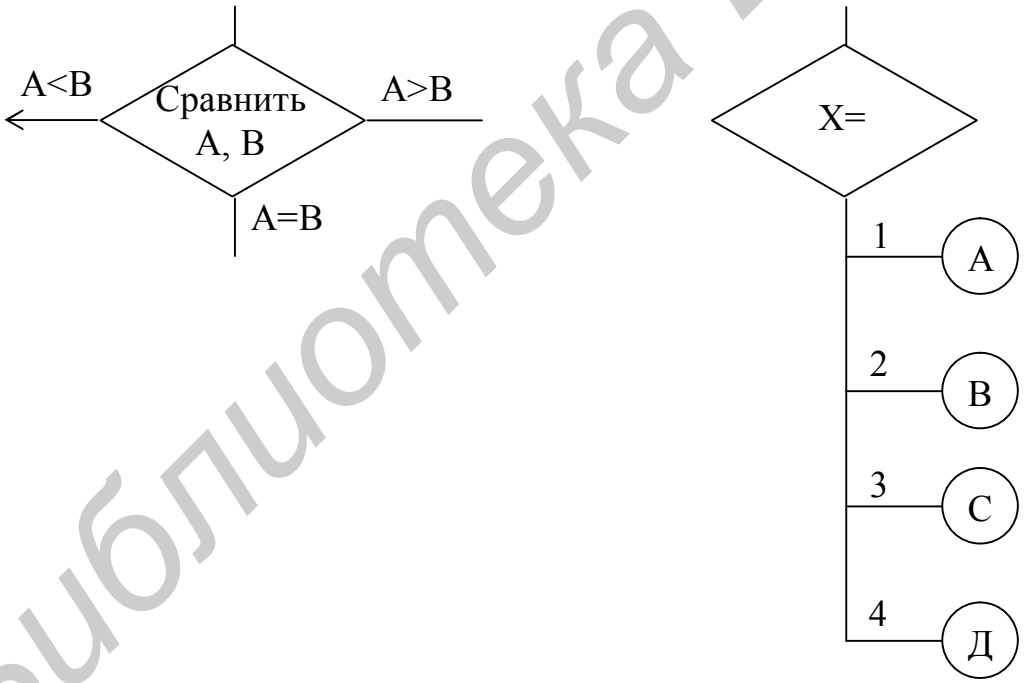


Рис. 2.8. Изображение значений условий, определяющих переход по логическому пути

2.2.3. Запись на алгоритмическом языке

Чтобы разработать алгоритм решения задачи, нужно представить ее в виде последовательности четких правил. Этому требованию полностью соответствует и исходный текст программы.

Понятия алгоритма и программы не очень четко разграничены. Обычно программа, записанная на алгоритмическом языке, – это окончательный вариант алгоритма решения задачи, ориентированный на конкретного исполнителя (компьютер или язык программирования).

В настоящее время существуют технологии разработки исходного текста программ без предварительного создания схем алгоритмов. Одна из них описана в разд. 4.

2.3. Разновидности структур алгоритмов

Различают следующие *структуры алгоритмов*:

1) линейные;
2) разветвляющиеся;
3) циклические.

2.3.1. Линейный вычислительный процесс

Линейный вычислительный процесс – это процесс, в котором направление вычислений является единственным.

Пример 2.2. Вычислить значение функции

$$Y = \sqrt{(\sin X + 2 * \cos(X/Z) + 3 * X/Z)} \ .$$

Символом $*$ в языках программирования принято обозначать операцию умножения.

Алгоритм вычисления данной функции является линейным, поскольку ход вычислительного процесса не зависит от каких-либо условий.

Алгоритм может быть разработан с различной степенью детализации. На рис. 2.9 приведена укрупненная схема алгоритма вычисления функции Y. На рис. 2 10 приведена подробная схема того же алгоритма.

Для повышения эффективности алгоритма желательно, чтобы выражения, участвующие в вычислениях несколько раз, вычислялись один раз, а затем использовались уже вычисленные их значения (на рис. 2.9, 2.10 один раз вычислено значение X/Z, полученное значение присвоено переменной A, которая затем используется в вычислениях).

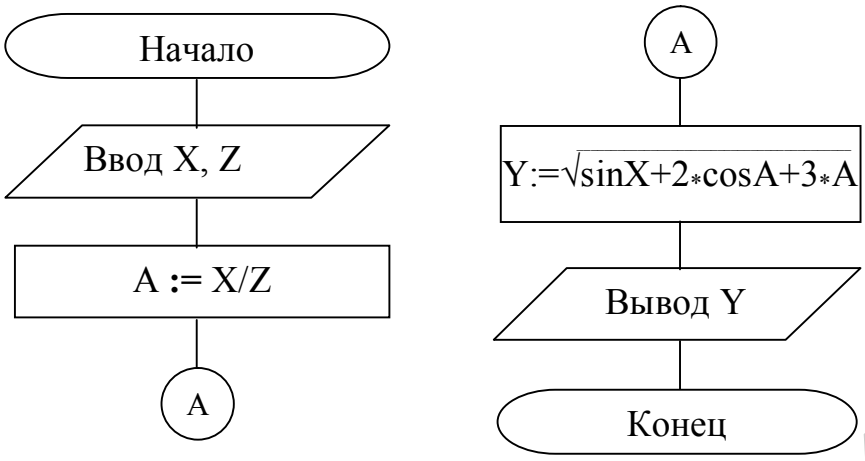


Рис. 2.9. Укрупненная схема алгоритма вычисления линейной функции Y

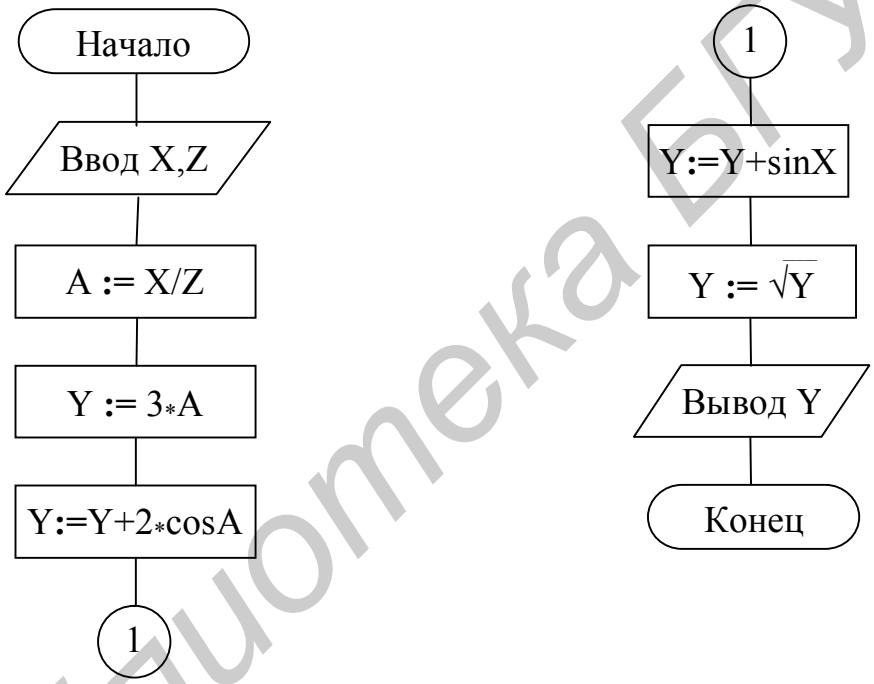


Рис. 2.10. Подробная схема алгоритма вычисления линейной функции Y

2.3.2. Разветвляющийся вычислительный процесс

Разветвляющийся вычислительный процесс – это процесс, в котором направление вычислений определяется некоторыми условиями.

Пример 2.3. Вычислить значение функции

$$Y = \begin{cases} 0, \text{ если } X < 0; \\ 1, \text{ если } X = 0; \\ 2, \text{ если } 0 < X < 0{,}5; \\ 3, \text{ если } 0{,}5 \le X < 1; \\ 4, \text{ если } X \ge 1. \end{cases}$$

Схема алгоритма вычисления функции Y для данного примера имеет вид, приведенный на рис. 2.11.

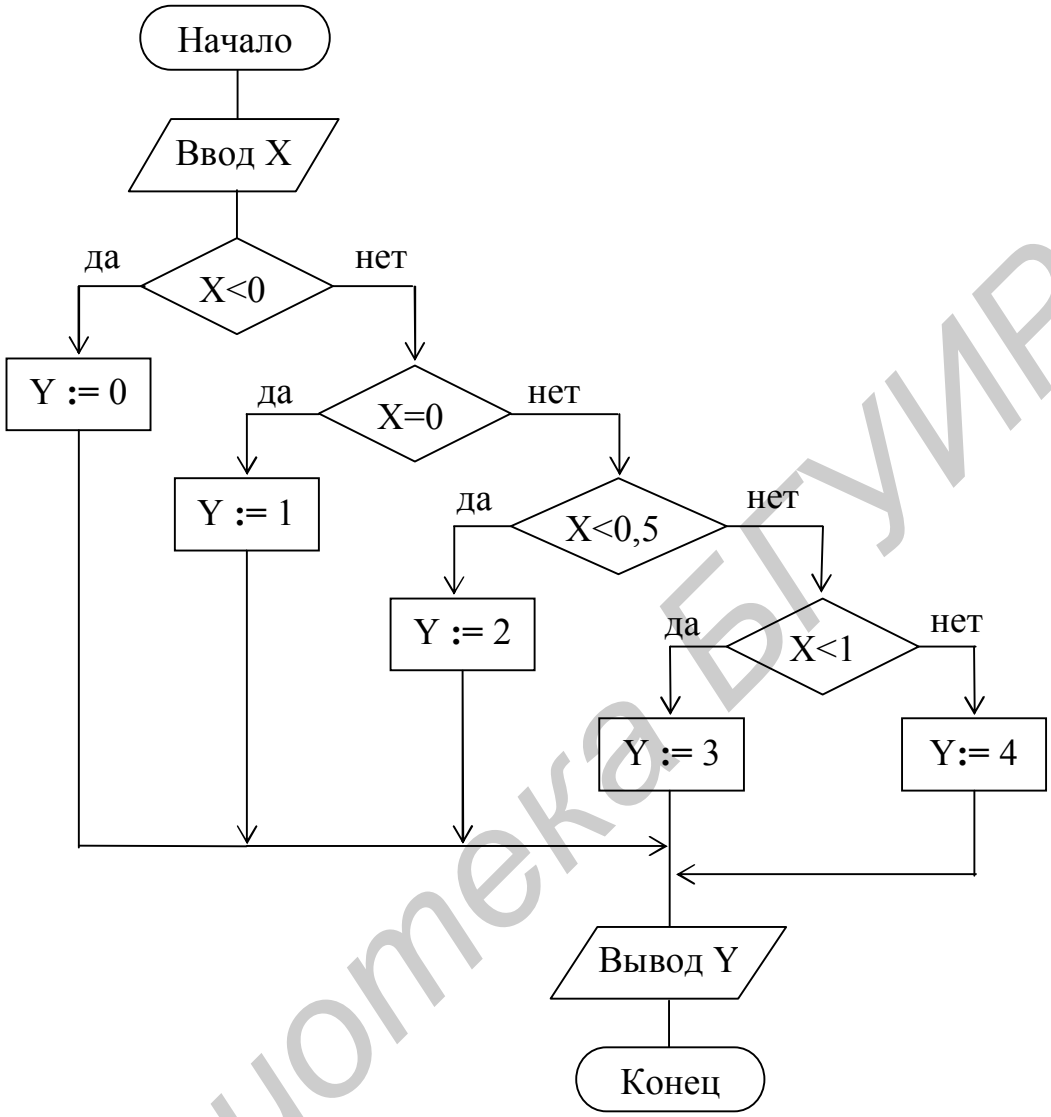


Рис. 2.11. Схема алгоритма вычисления разветвляющейся функции Y

Этот же алгоритм в более компактной форме может быть представлен, например, так, как показано на рис. 2.12.

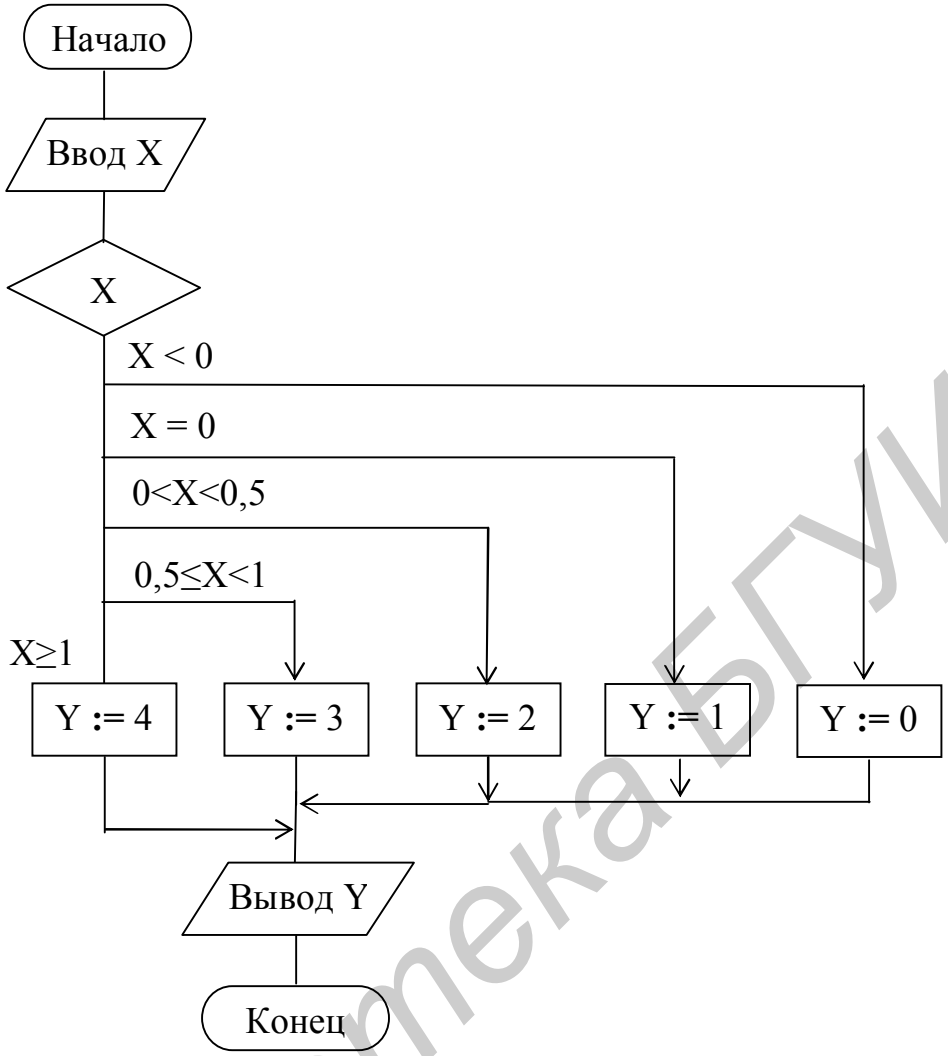


Рис. 2.12. Компактная схема представления алгоритма вычисления разветвляющейся функции Y

2.3.3. Циклический вычислительный процесс

Циклический вычислительный процесс – это процесс, в котором отдельные участки вычислений выполняются многократно.

Участок схемы, многократно повторяемый в ходе вычислений, называется *циклом*. При повторениях обычно используются новые значения исходных данных.

В соответствии с *взаимным расположением циклов* в теле программы или алгоритма различают следующие циклы:

1) *простые* – циклы, не содержащие внутри себя других циклов;

2) *сложные* – циклы, содержащие внутри себя другие циклы;

3) *вложенные (внутренние)* – циклы, входящие в состав других циклов (цикл в цикле);

4) *внешние* – циклы, не являющиеся составной частью других циклов, но содержащие в своем составе внутренние циклы.

В зависимости от *местоположения условия* выполнения цикла различают:

1) циклы с предусловием;
2) циклы с постусловием.

В соответствии с *видом условия выполнения* циклы делятся на:

1) циклы с параметром;
2) итерационные циклы.

Для решения вопроса о том, сколько раз нужно выполнять цикл, используется анализ переменной (или нескольких переменных), называемой *параметром цикла*.

В общем случае схема алгоритма циклического процесса с параметром имеет вид, приведенный на рис. 2.13.

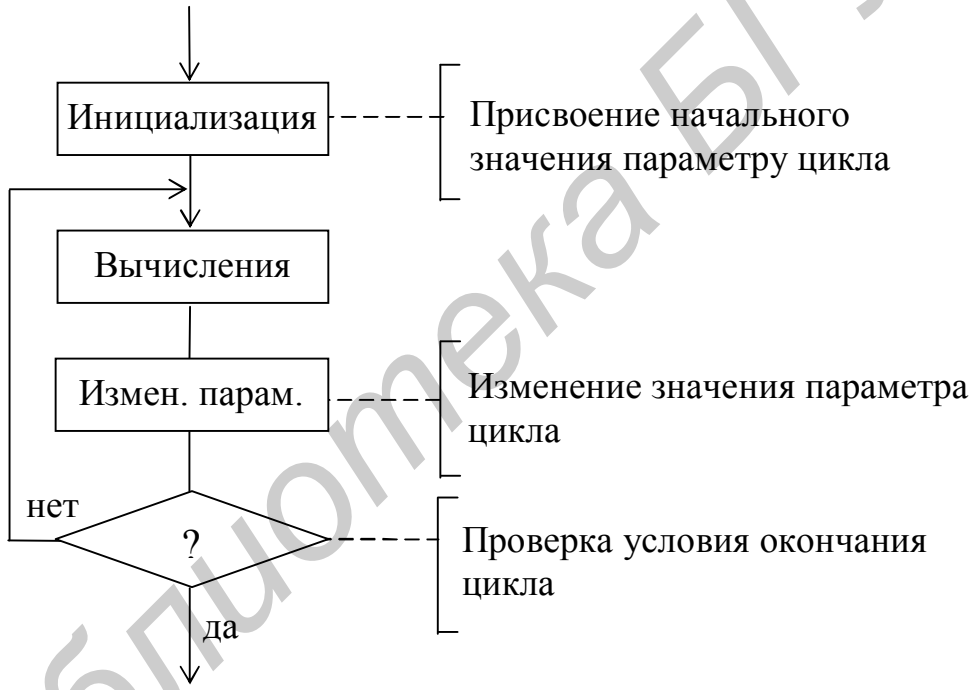


Рис. 2.13. Схема алгоритма циклического процесса с параметром (с постусловием)

Блоки, выполняющие вычисления и изменение значения параметра цикла, составляют так называемое *тело цикла* – последовательность действий, которая выполняется многократно.

В терминах метода структурного программирования вышеприведенный цикл называется *циклом «До» (циклом с постусловием)* – в нем тело цикла выполняется **до** проверки условия выхода из цикла.

В *цикле «Пока» (цикле с предусловием)* проверка условия выполнения цикла производится до тела цикла (рис. 2.14).

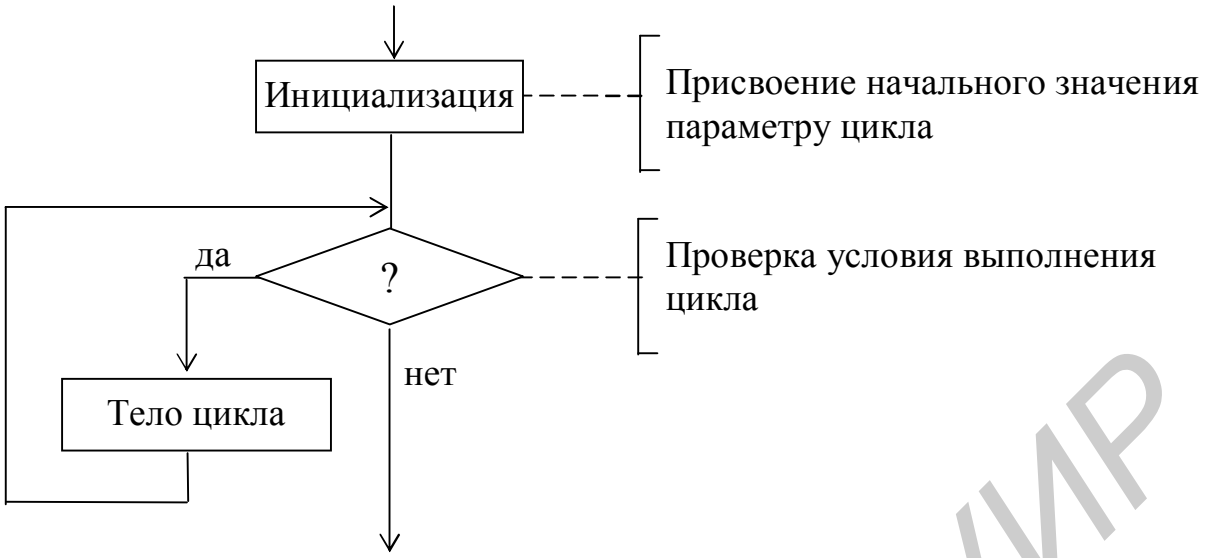


Рис. 2.14. Схема алгоритма циклического процесса с предусловием

Циклический процесс, в котором число выполнений тела цикла заранее определено, называется ***циклическим процессом с известным количеством повторений (циклы со счетчиками, циклы с параметром цикла)***. Например, при вычислении суммы двадцати элементов массива заранее известно, что количество повторений тела цикла равно 20.

Пример 2.4. Алгоритм, содержащий сложный цикл с заданным числом повторений. Вычислить значение функции

$$Y = \prod_{i=1}^{I} \sum_{j=1}^{J} Xij.$$

Пусть I = 10, J = 20. Тогда выражение для вычисления функции Y будет иметь вид

$$Y = (X_{11} + X_{12} + ... + X_{1(20)})*(X_{21} + X_{22} + ... + X_{2(20)})*...*(X_{(10)1} + X_{(10)2} + ... + X_{(10)(20)}).$$

Схема алгоритма вычисления функции Y имеет вид, приведенный на рис. 2.15.

Данный алгоритм содержит внешний цикл, в состав которого входит внутренний цикл. Параметром внешнего цикла является переменная i, параметром внутреннего цикла – переменная j. Блоки 5, 6 составляют тело внутреннего цикла, блоки 4–9 – тело внешнего цикла. Алгоритм реализован с использованием циклов «До». *Переменные, служащие для накопления суммы, в исходном состоянии всегда устанавливаются в ноль* (в данном примере это S). *Переменные, служащие для формирования произведения, в исходном состоянии всегда устанавливаются в единицу* (в данном примере это Y).

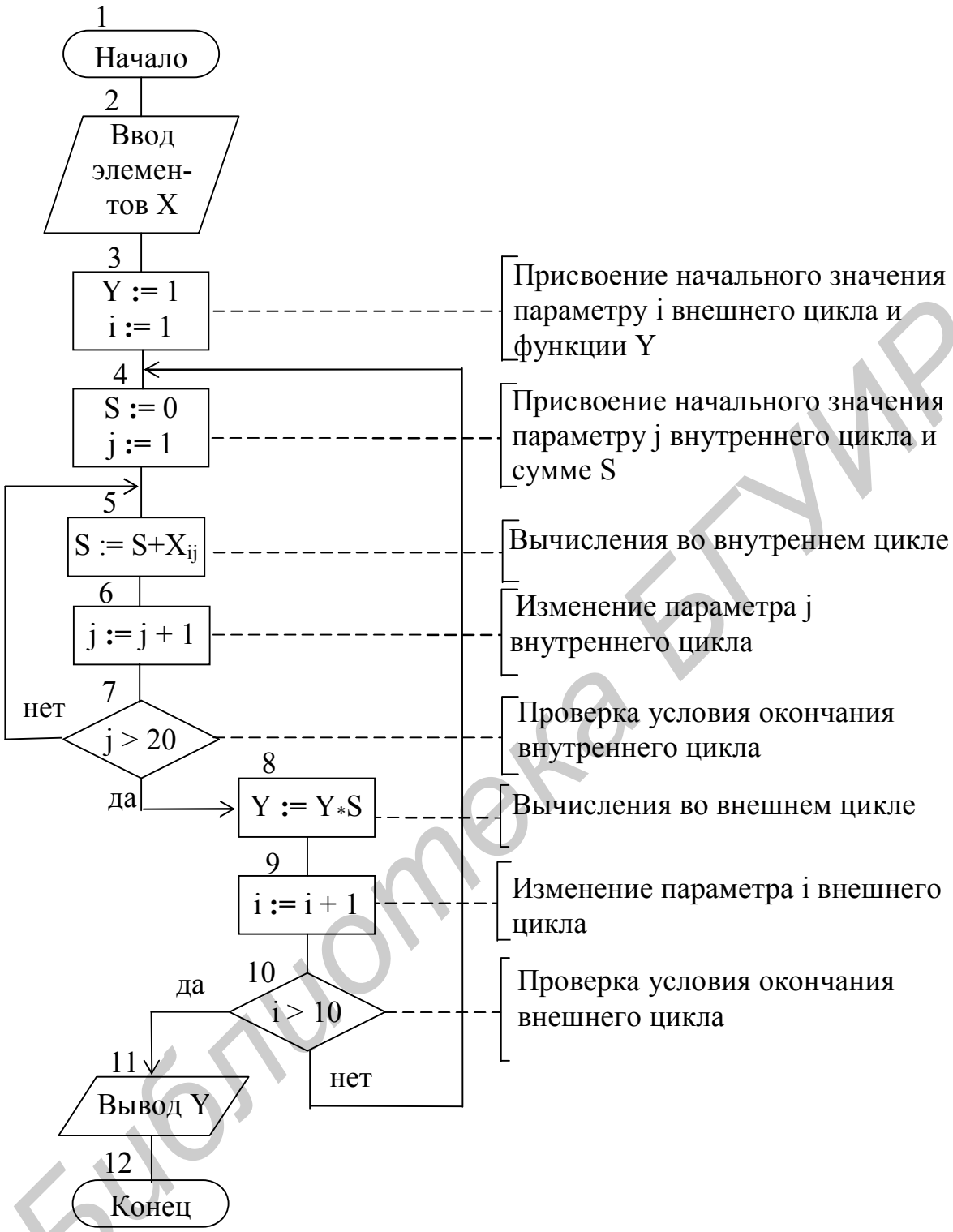


Рис. 2.15. Схема алгоритма вычисления функции Y (к примеру 2.4)

Циклический процесс, в котором количество повторений заранее неизвестно и зависит от получающихся в ходе вычислений результатов, называется ***итерационным***.

Типовой пример – вычисление суммы ряда с погрешностью вычислений, не превышающей заранее заданной. Вычисления продолжаются до тех пор,

пока, например, разность между значениями величин, получаемых на соседних шагах цикла, не станет меньше или равной некоторой заранее заданной величине – точности.

Для итерационных процессов характерно то, что значения, получаемые на текущем шаге итерации, используются, как правило, в качестве исходных данных для следующего шага итерации. Такое использование в большинстве случаев позволяет существенно повысить эффективность разрабатываемого алгоритма.

В общем виде итерационный процесс может быть представлен следующим образом (рис. 2.16).

Рис. 2.16. Схема алгоритма итерационного процесса

Пример 2.5. Алгоритм, содержащий сложный цикл с неизвестным числом повторений. Вычислить значение функции

$$Y = \sin X$$

через разложение функции в бесконечный ряд

$$Y = \sin X = X - X^3/3! + X^5/5! - X^7/7! + \mathbf{...}$$

Вычисления прекратить, когда разность между модулями двух соседних слагаемых станет меньше величины Eps = 0,0001.

Схема алгоритма решения данной задачи имеет вид, представленный на рис. 2.17.

В переменной Y накапливается значение суммы ряда. В переменной A1 хранится слагаемое, сформированное на предыдущей итерации и используемое в качестве исходной величины для вычисления слагаемого на данной итерации.

В переменной A2 формируется слагаемое на данной итерации. Переменная i – служебная переменная, используемая для формирования знаменателя слагаемых и для определения выполненного числа итераций.

Рис. 2.17. Схема алгоритма итерационного процесса вычисления значения функции Y = sinX

В переменной E формируется разность между соседними слагаемыми (с учетом их разнозначности).

После окончания текущей итерации необходимо запомнить значение слагаемого, сформированное в переменной A2, иначе на следующей итерации оно потеряется. Поэтому значение переменной A2 присваиваем переменной A1.

Многие из реально существующих алгоритмов имеют смешанный характер, т.е. могут содержать линейные участки, разветвления, циклы с известным количеством повторений и итерационные циклы. В связи с этим составление алгоритмов сразу в законченной форме затруднено. Поэтому для составления сложных алгоритмов рекомендуется использовать *нисходящее проектирование программ* (метод пошаговой детализации, метод последовательных уточнений). *Его суть*: первоначально продумывается общая структура алгоритма, без детальной проработки его отдельных частей. Далее прорабатываются отдельные блоки, не детализированные на предыдущем шаге. Таким образом, на каждом шаге разработки уточняется реализация фрагмента алгоритма, т.е. решается более простая задача.

3. СТРУКТУРНОЕ ПРОГРАММИРОВАНИЕ

В 70-х годах прошлого века, с появлением ЭВМ третьего поколения, возникает новый подход к разработке алгоритмов и программ, который получает название *структурного проектирования программ*. Одним из первых инициаторов структурного программирования был профессор Э. Дейкстра. В 1965 г. он высказал предположение, что оператор GoTo (оператор безусловного перехода) вообще может быть исключён из языков программирования. По мнению Дейкстры, «квалификация программиста обратно пропорциональна числу операторов GoTo в его программах».

Достоинства структурного программирования по сравнению с интуитивным неструктурным программированием:

1) уменьшение трудностей тестирования программ;
2) более высокая производительность программистов;
3) ясность и читаемость программ, что упрощает их сопровождение;
4) эффективность программ.

3.1. Теория структурного программирования

К *концепциям* структурного программирования относится отказ от использования оператора безусловного перехода (GoTo), замена его рядом других, более структурированных операторов и использование идей нисходящего программирования.

Основное назначение *нисходящего проектирования* – служить средством разбиения большой задачи на меньшие подзадачи так, чтобы каждую подзадачу можно было рассматривать независимо.

В предыдущем разделе была кратко описана суть метода нисходящего проектирования. Напомним, что при его использовании вначале проектируется общая структура алгоритма, без детальной проработки его отдельных частей. Затем разрабатываются блоки алгоритма, не детализированные на предыдущем шаге. В результате на каждом шаге разработки детализируется фрагмент алгоритма, т.е. решается более простая задача.

В основу структурного программирования положено *требование*, чтобы каждый модуль алгоритма (программы) проектировался с единственным входом и единственным выходом. Программа представляется в виде множества *вложенных* модулей, каждый из которых имеет один вход и один выход.

Базой для реализации структурированных программ является *принцип Бома и Джакопини*, в соответствии с которым всякая реальная программа может быть построена с использованием лишь *двух управляющих конструкций*.

По Бому и Джакопини логическая структура программы может быть выражена комбинациями *трех базовых структур*:

1) функционального блока;

2) конструкции принятия двоичного (дихотомического) решения;
3) конструкции обобщенного цикла.

Функциональный блок – это отдельный вычислительный оператор или любая другая реальная последовательность вычислений с единственным входом и единственным выходом. Изображается с помощью символа «Процесс» (рис. 3.1).

Рис. 3.1. Изображение функционального блока в структурном программировании

Конструкция принятия двоичного (дихотомического) решения обычно называется элементом If-Then-Else (если-то-иначе), разветвлением или ветвлением – структура, обеспечивающая выбор между двумя альтернативными путями вычислительного процесса в зависимости от выполнения некоторого условия. Изображается с помощью символов «Решение» и «Процесс» (рис. 3.2).

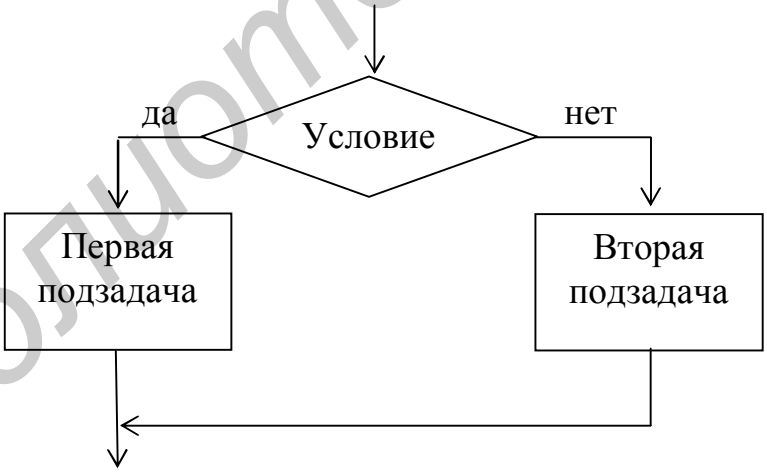


Рис. 3.2. Изображение конструкции If-Then-Else в структурном программировании

Конструкция обобщенного цикла – в качестве базовой конструкции структурного программирования используется цикл с предусловием,

называемый циклом «Пока» (Do-While). Изображается с помощью символов «Решение» и «Процесс» следующим образом (рис. 3.3).

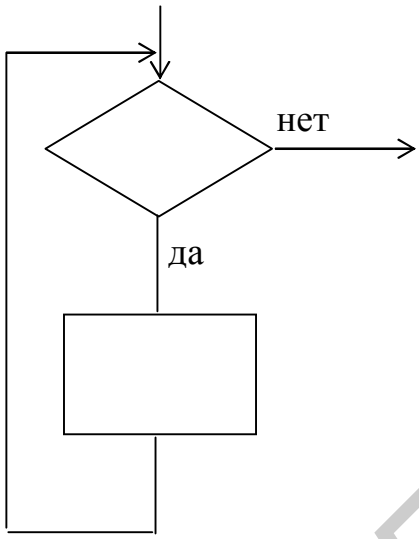


Рис. 3.3. Изображение конструкции обобщенного цикла в структурном программировании

Из рис. 3.1–3.3 видно, что логические конструкции (конструкция принятия двоичного решения и конструкция обобщенного цикла) имеют только один вход и один выход. Поэтому они могут рассматриваться как функциональные блоки. С учётом этого вводится *преобразование логических блоков в функциональный блок.*

Кроме того, всякая последовательность функциональных элементов, называемая *конструкцией следования* (рис. 3.4), также может быть приведена к одному функциональному блоку.

Рис. 3.4. Конструкция следования

Данные преобразования называются *преобразованиями Бома–Джакопини.* Их основу составляет принцип «чёрного ящика» (что-то с одним входом и одним выходом).

Таким образом, всякая программа, состоящая из функциональных блоков, операторов цикла и элементов If-Then-Else, поддаётся последовательному преобразованию к единственному функциональному блоку.

Эта последовательность преобразований может быть использована как средство понимания программы, доказательство ее правильности и *структурированности* программы.

Обратная последовательность преобразований может быть использована в процессе проектирования алгоритма (программы) по методу *нисходящего проектирования* – алгоритм (программа) разрабатывается, исходя из единственного функционального блока, который постепенно раскрывается в сложную структуру основных элементов.

3.2. Реализация структурного проектирования в современных языках программирования

Реализация теоретических основ структурного программирования базируется на следующих *правилах.* Все операции в программе должны представлять собой либо непосредственно исполняемые в линейном порядке выражения, либо одну из следующих *управляющих конструкций*:

1) вызовы подпрограмм – любое допустимое на конкретном языке программирования обращение к замкнутой подпрограмме с одним входом и одним выходом;

2) вложенные на произвольную глубину операторы If-Then-Else;

3) циклические операторы (цикл с предусловием, называемый циклом «Пока»).

Этих средств достаточно для составления структурированных программ. Однако иногда допускаются их некоторые *расширения*:

1) дополнительные конструкции организации цикла:
   - цикл с параметром как вариант цикла с предусловием;
   - цикл с постусловием, называемый в структурном программировании циклом «До», – для него характерно то, что тело цикла выполняется **до** проверки условия выхода из цикла; изображается, как показано на рис. 3.5;

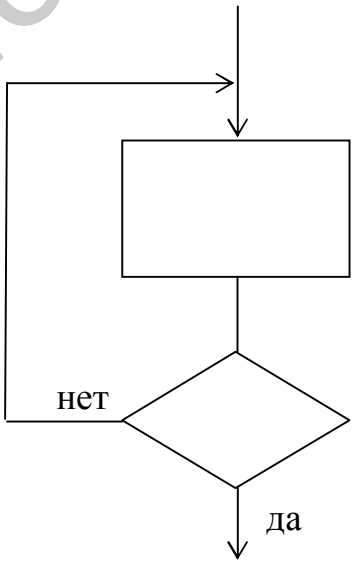


Рис. 3.5. Изображение цикла с постусловием в структурном программировании

2) подпрограммы с несколькими входами и несколькими выходами (например, один выход нормальный, второй – по ошибке);

3) применение оператора GoTo с жёсткими ограничениями (например, передача управления не далее чем на десять операторов или только вперёд по программе);

4) использование оператора Case как расширения конструкции If-Then-Else; в структурном проектировании программ конструкция Case представляется, как показано на рис. 3.6.

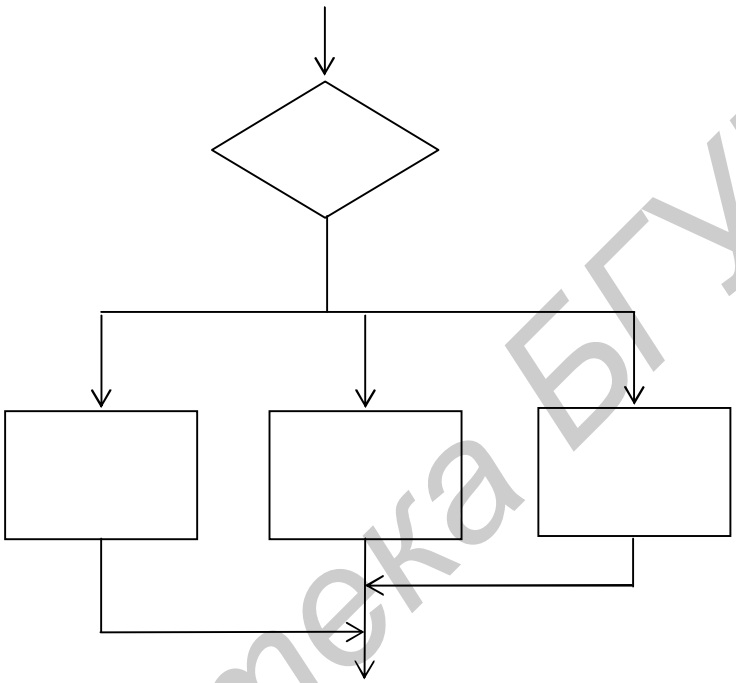


Рис. 3.6. Изображение конструкции Case в структурном программировании

3.3. Преобразование неструктурированных программ в структурированные

В общем случае произвольная программа не может быть преобразована в структурированную программу, которая реализует тот же алгоритм, построена с применением тех же конструкций и не использует дополнительных переменных. Такое преобразование возможно при использовании трех известных *методов*:

- дублирование кодов программы;
- введение переменной состояния;
- метод булевых признаков.

3.3.1. Метод дублирования кодов программы

Рассмотрим применение данного метода на примере неструктурированной программы типа «решетка». Алгоритм такой программы

схематично представлен на рис. 3.7.

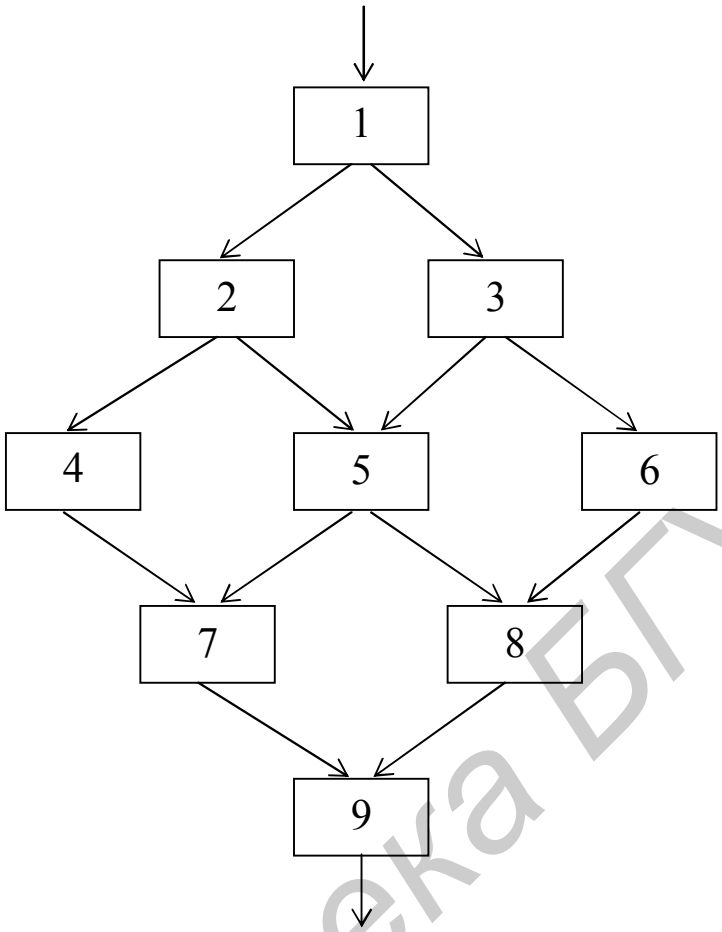


Рис. 3.7. Алгоритм неструктурированной программы типа «решетка»

Стрелки, соединяющие блоки на данной схеме алгоритма, представляют собой операторы перехода.

Программа, реализующая данный алгоритм, не является структурированной, так как не удовлетворяет условию «один вход – один выход».

Сущность метода дублирования кодов: дублируются те модули исходного алгоритма или программы, в которые можно войти из нескольких мест (кроме последнего блока).

В соответствии с этим в исходной схеме (см. рис. 3.7) необходимо дублировать модули 5 (в него можно войти из модулей 2 и 3), 7 (в него можно войти из модулей 4 и 5) и 8 (в него можно войти из модулей 5 и 6). Это приводит исходную схему к виду, приведенному на рис. 3.8.

Полученная схема является структурированной.

Чтобы доказать это, необходимо воспользоваться преобразованиями Бома–Джакопини, т.е. последовательно преобразовать схему к одному функциональному блоку с одним входом и одним выходом. Для этого необходимо выполнить несколько шагов.

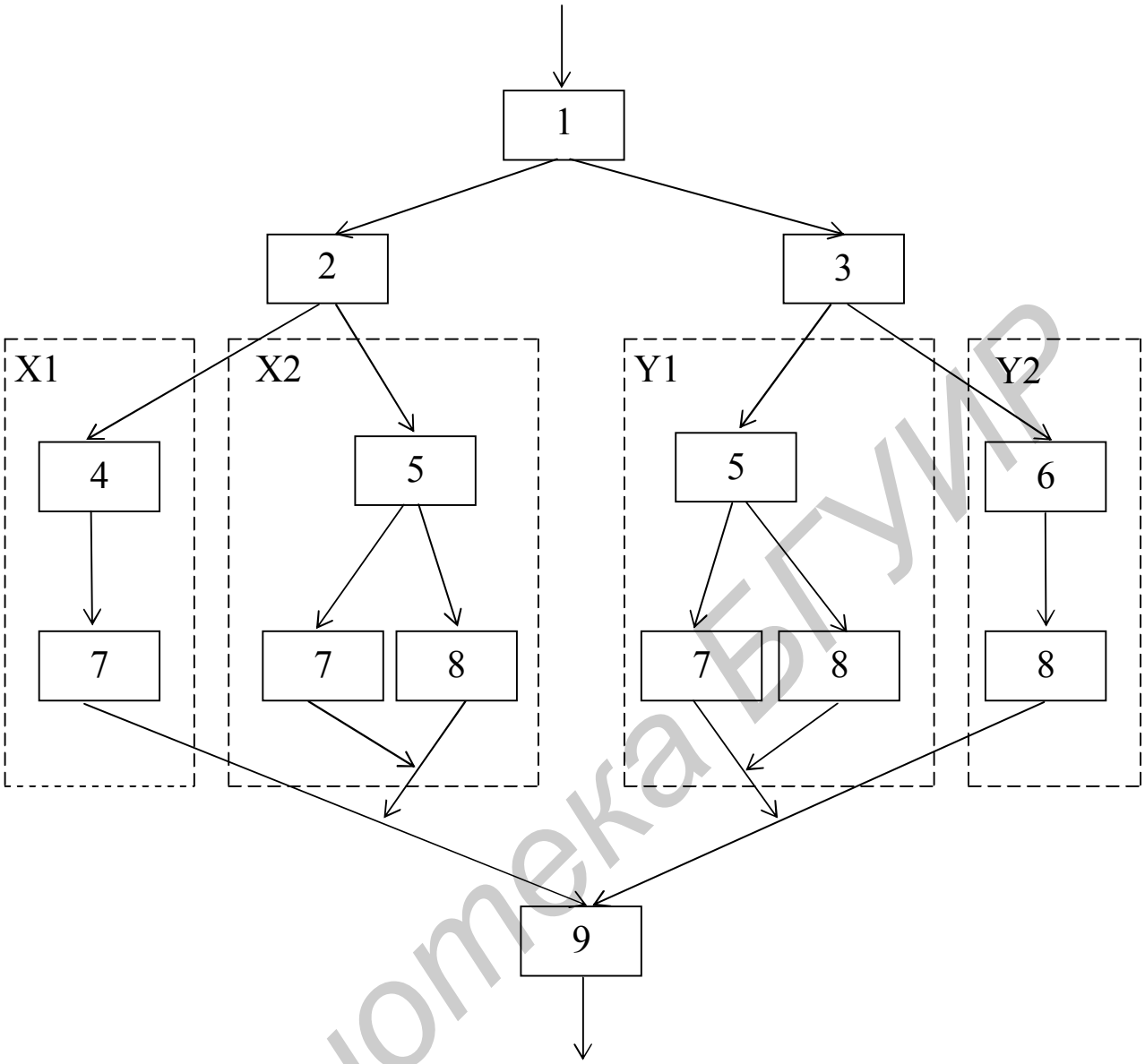


Рис. 3.8. Алгоритм программы, структурированной с помощью метода дублирования кодов

Шаг 1 преобразования

На вышеприведенной схеме (см. рис. 3.8) модули 4 и 7 представляют собой конструкцию следования. В соответствии с преобразованиями Бома–Джакопини они могут быть сведены к одному функциональному блоку X1. Модули 5, 7, 8 представляют собой конструкцию If-Then-Else с одним входом и одним выходом. Они также могут быть преобразованы к одному функциональному блоку (X2 и Y1). Аналогичные рассуждения справедливы для конструкции следования, состоящей из модулей 6, 8, – она сводится к функциональному блоку Y2. В результате схема, приведенная на рис. 3.8, принимает следующий вид (рис. 3.9).

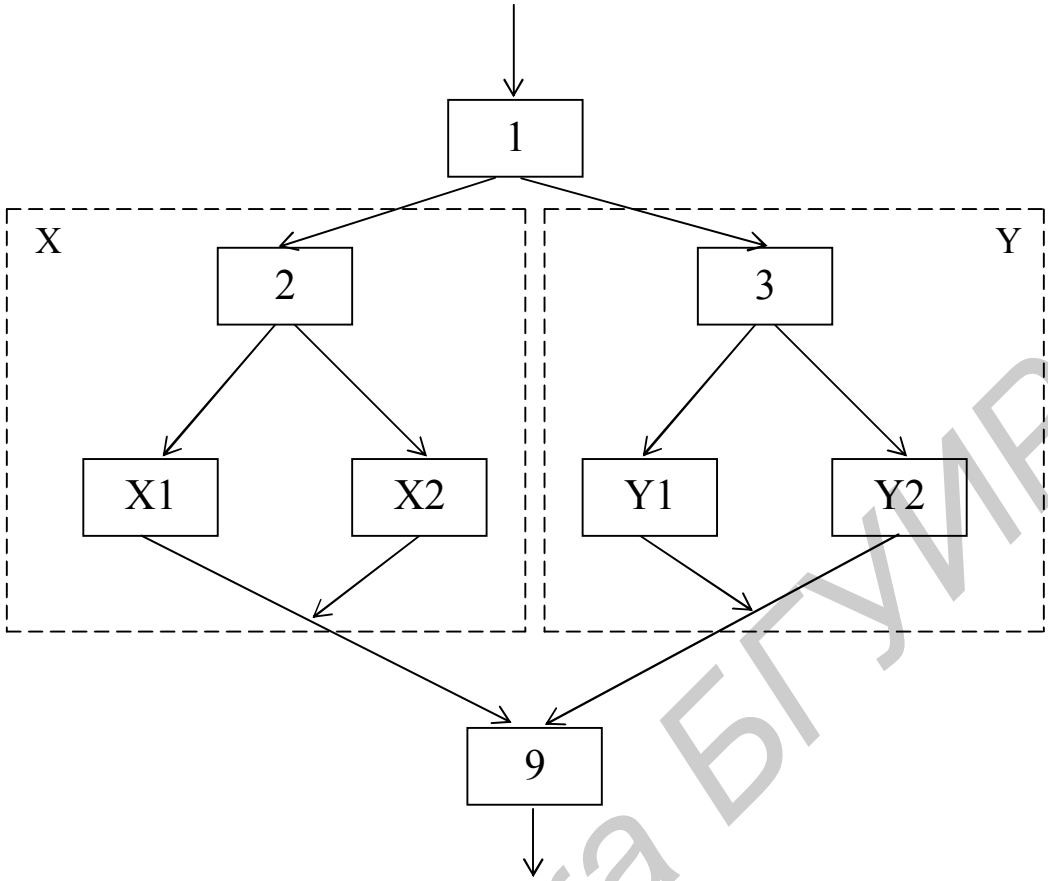


Рис. 3.9. Схема алгоритма программы после первого шага преобразований Бома–Джакопини

Шаг 2 преобразования

В полученной в результате выполнения шага 1 схеме группа модулей 2, X1, X2 и группа модулей 3, Y1, Y2 представляют собой конструкции If-Then-Else с одним входом и одним выходом. В соответствии с преобразованиями Бома–Джакопини их можно представить в виде функциональных блоков (блоков X и Y соответственно). В результате схема принимает вид, представленный на рис. 3.10.

Шаг 3 преобразования

В полученной в результате выполнения шага 2 схеме группа модулей 1, X, Y представляет собой конструкцию If-Then-Else с одним входом и одним выходом. В соответствии с преобразованиями Бома–Джакопини ее можно представить в виде функционального блока Z. В результате схема принимает вид конструкции следования (рис. 3.11).

Шаг 4 преобразования

В соответствии с преобразованиями Бома–Джакопини конструкцию следования можно представить в виде функционального блока.

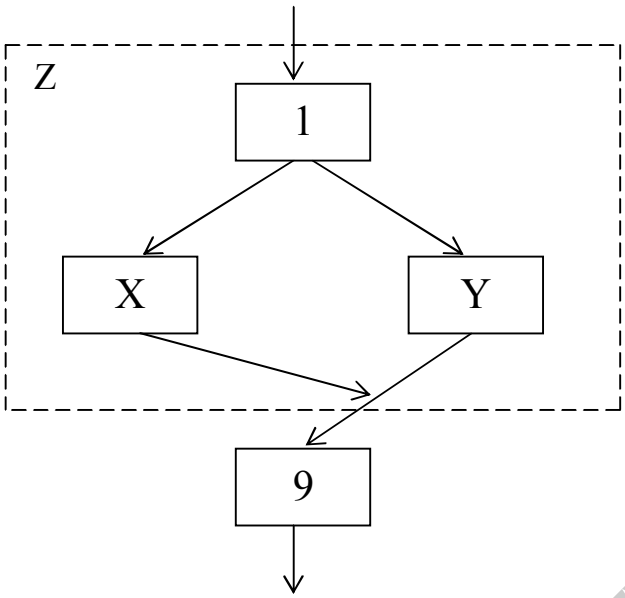


Рис. 3.10. Схема алгоритма программы после второго шага преобразований Бома–Джакопини

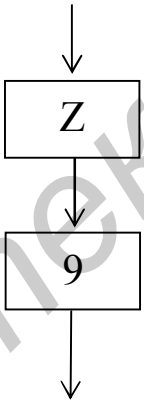


Рис. 3.11. Схема алгоритма программы после третьего шага преобразований Бома–Джакопини

Таким образом, с помощью четырех шагов преобразования доказано, что полученная в результате применения метода дублирования кодов схема алгоритма является структурированной.

Достоинством метода дублирования кодов является то, что его удобно использовать при нисходящем проектировании программ. Исходную задачу укрупненно можно представить в виде одного функционального блока, а затем постепенно разукрупнять ее через промежуточные схемы алгоритма к результирующей структурированной схеме (рассмотрите предыдущие схемы, соответствующие четырем шагам преобразования, в обратном порядке).

Недостатки метода дублирования кодов:

1) неприменимость к программам с циклами;

2) дополнительные затраты памяти для хранения дублируемых модулей.

Поэтому метод используется, если дублируемые модули содержат незначительное число операторов. Если модули велики, то вместо дублирования кодов необходимо использовать вызываемые подпрограммы с формальными параметрами.

3.3.2. Метод введения переменной состояния

Данный метод был впервые предложен Ашкрофтом и Манной.

Рассмотрим применение этого метода на примере неструктурированной программы, алгоритм которой схематично представлен на рис. 3.12.

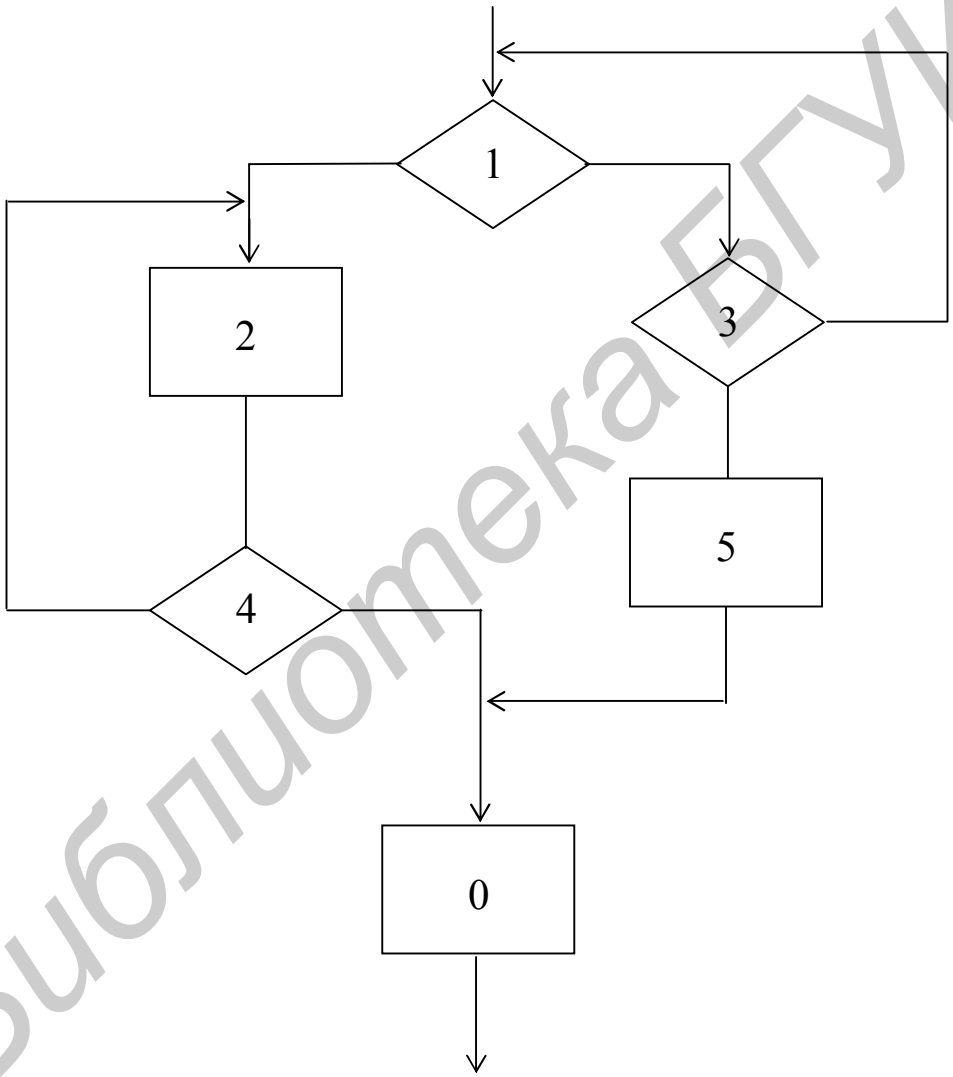


Рис. 3.12. Схема алгоритма исходной неструктурированной программы

Данная схема не является структурированной, так как из цикла, состоящего из блоков 1 и 3, существует два выхода. Таким образом, нарушено условие «один вход – один выход», которому должны удовлетворять структурированные схемы.

Процесс преобразования программы в структурированную состоит из нижеприведенной *последовательности шагов*.

1. Каждому блоку неструктурированной схемы приписывается номер. Обычно первому блоку присваивается 1, последнему – 0 (см. рис. 3.12).

2. В программу вводится дополнительная переменная целого типа (например I), называемая *переменной состояния*.

3. Функциональные блоки исходной схемы заменяются блоками, выполняющими помимо основных функций преобразование переменной I: переменной I присваивается значение, равное номеру блока-приёмника в исходной схеме.

4. Аналогично преобразуются логические блоки. При этом если в логическом блоке условие истинно, то это соответствует одному значению I, если ложно – другому.

5. Исходная схема пребразуется к виду, предложенному Ашкрофтом–Манной (рис. 3.13).

На данной схеме блоки **1a – na** являются аналогами соответствующих блоков исходной схемы и, помимо этого, присваивают значение переменной I.

В результате преобразований Ашкрофта–Манны исходная неструктурированная схема (см. рис. 3.12) принимает вид, представленный на рис. 3.14.

При выполнении алгоритма, реализованного по методу Ашкрофта–Манны, переменная состояния I устанавливается в начальное значение, равное номеру первого блока непреобразованной схемы (как правило, это единица). Затем осуществляется последовательный опрос переменной I, начиная с нуля и заканчивая максимальным номером блока исходной схемы (в нашем примере он равен пяти). Выполняется тот блок исходной схемы, номер которого соответствует текущему значению I. Помимо этого в I заносится значение, равное номеру того блока исходной схемы, который должен выполняться за текущим блоком. Когда значение I станет равно нулю, выполняется последний блок непреобразованной схемы (блок с номером ноль) и осуществляется выход из алгоритма.

Полученная по методу Ашкрофта–Манны схема алгоритма является структурированной. Для доказательства этого достаточно последовательно преобразовать данную схему к одному функциональному блоку.

Шаг 1 преобразования

Конструкции **1a, 3a** и **4a** представляют собой конструкции If-Then-Else с одним входом и одним выходом, конструкции **2a, 5a** являются конструкциями следования. Следовательно, они могут быть преобразованы к соответствующим функциональным блокам. Данный шаг преобразований и все последующие шаги поясняет рис. 3.15.

Следующие шаги преобразований необходимо проводить снизу вверх схемы.

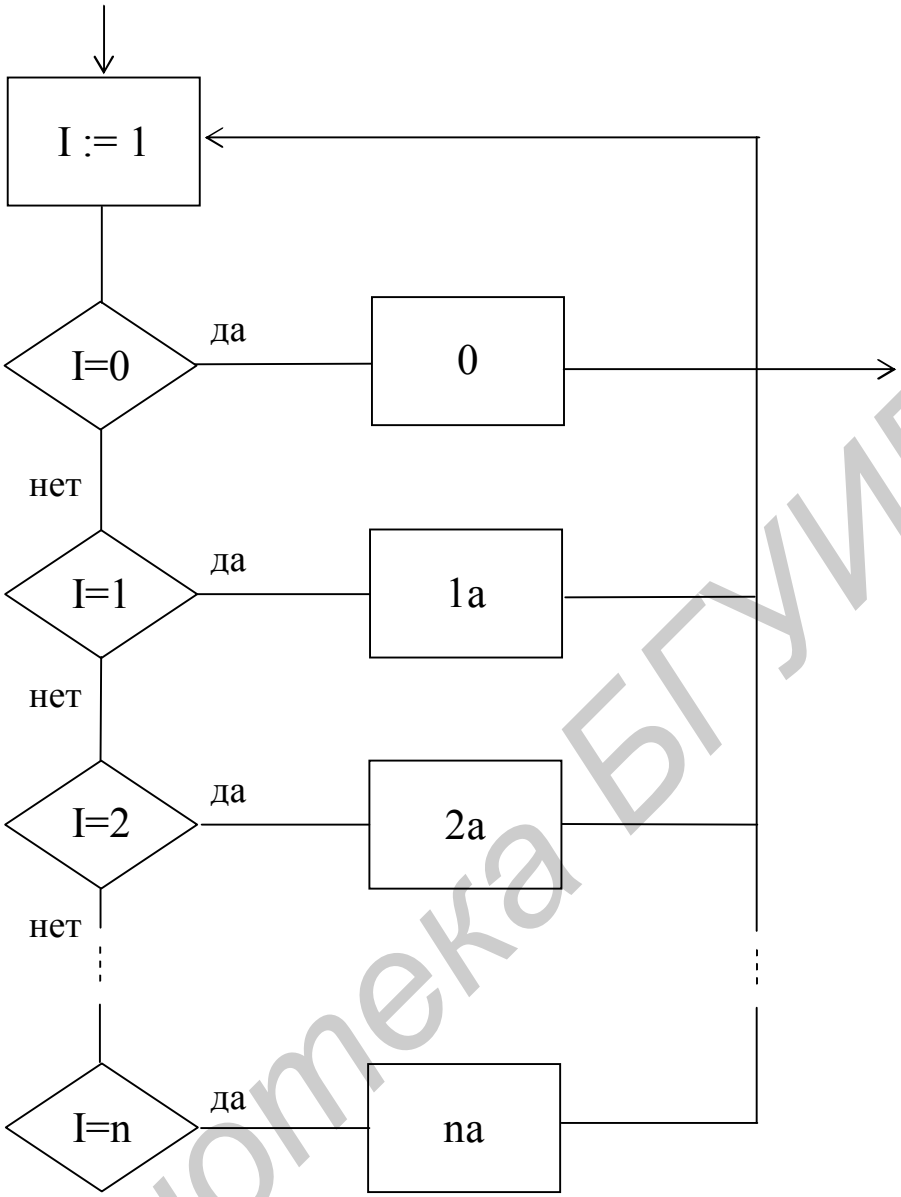


Рис. 3.13. Схема алгоритма Ашкрофта–Манны

Шаг 2 преобразования

Символ «Решение» с проверкой условия I=5 и блок **5a** представляют собой конструкцию If-Then-else (с одной ветвью) с одним входом и одним выходом. Поэтому данный символ «Решение» и блок **5a** могут быть заменены функциональным блоком **I**.

Вторая ветвь данного символа «Решение» может быть использована для повышения надежности программы и на рис. 3.15 не показана (возможность контроля непопадания значений I в диапазон 0 – **n**, где **n** – максимальный номер блока в исходной схеме). По сути, без проверки условия I=5 в алгоритме можно обойтись, перейдя на выполнение блока **5a** по ветви «нет» проверки условия I=4.

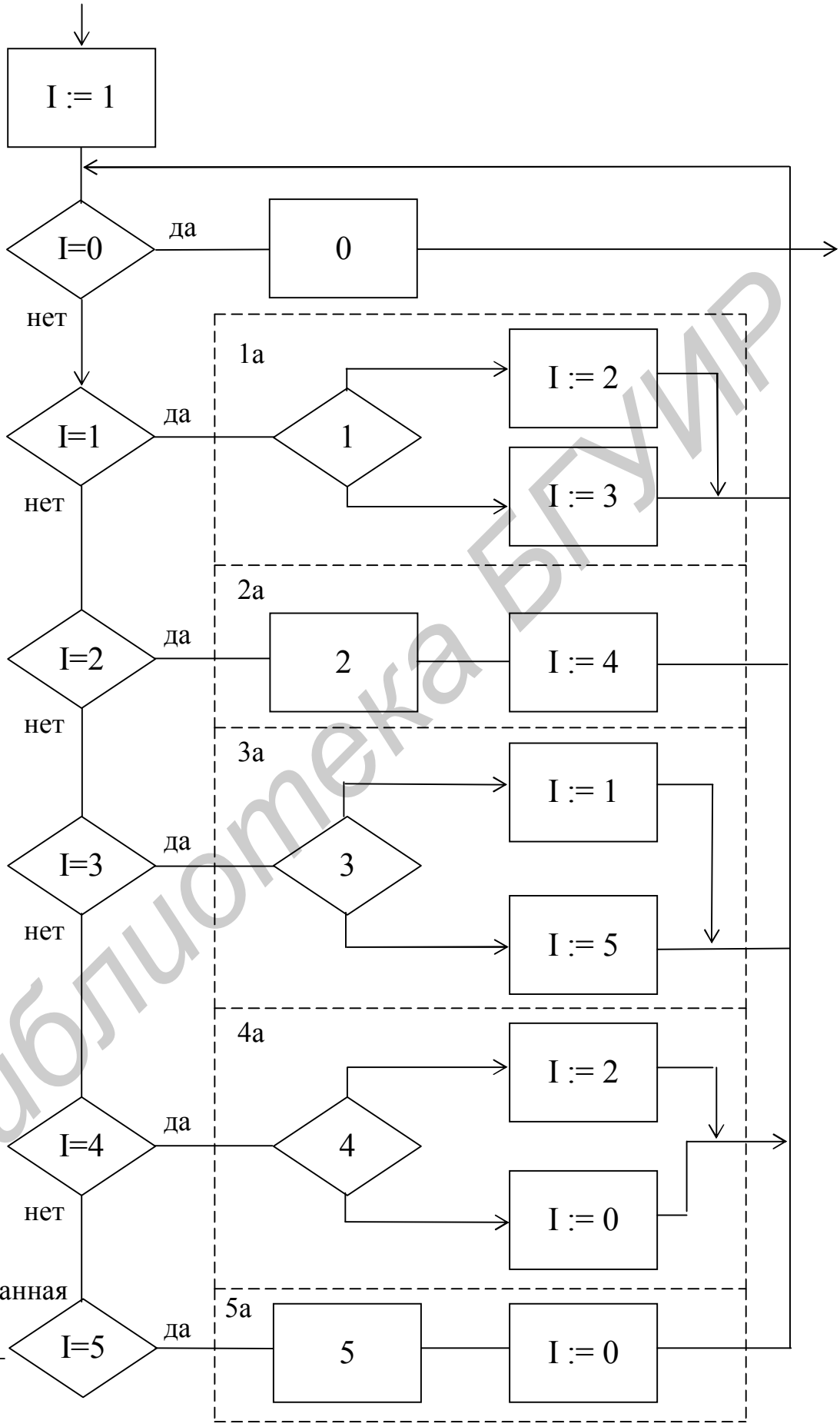


Рис. 3.14. Исходная схема, преобразованная по методу Ашкрофта–Манны

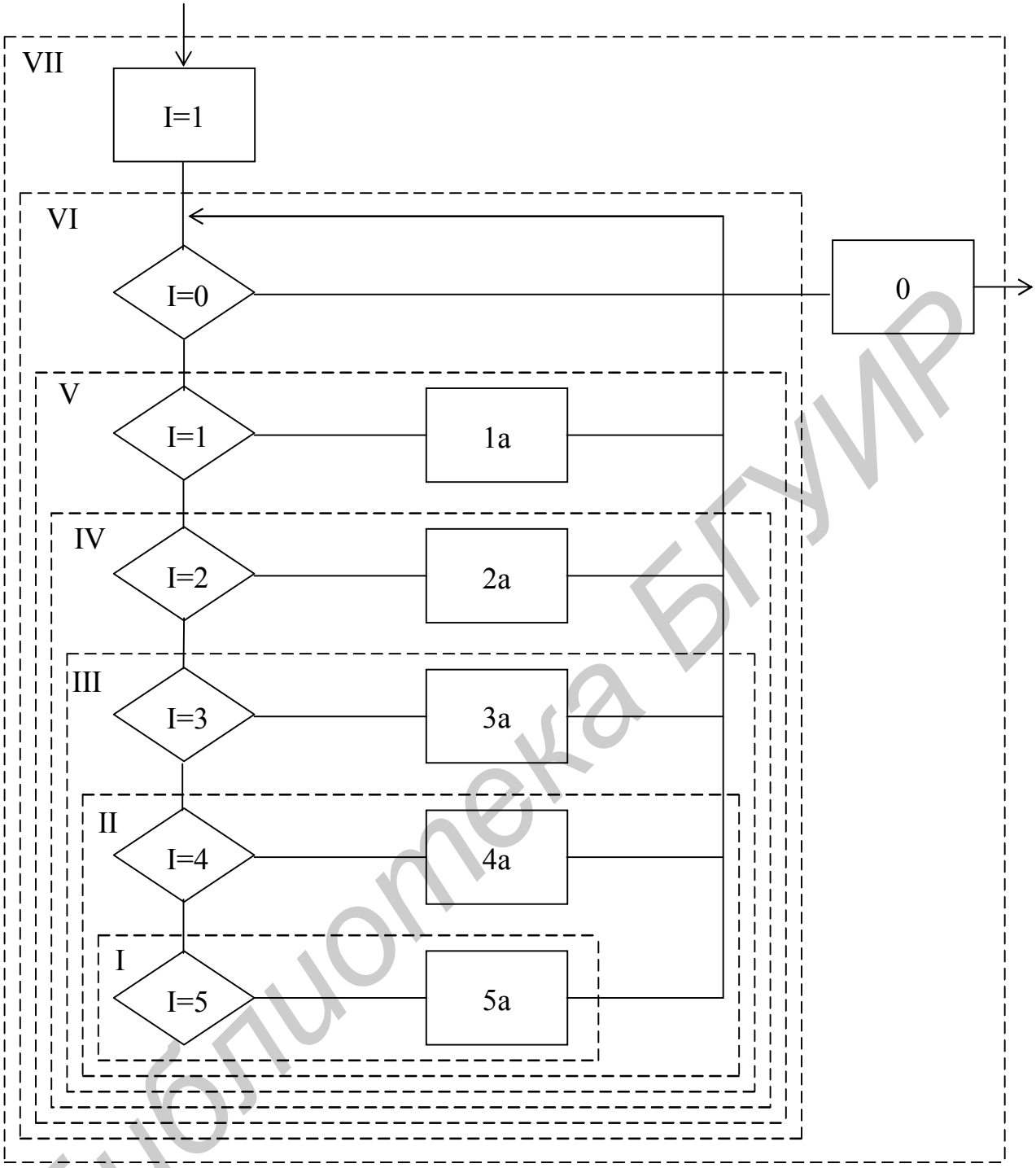


Рис. 3.15. Доказательство структурированности схемы алгоритма, полученной по методу Ашкрофта–Манны

Шаг 3 преобразования

Символ «Решение» с проверкой условия I=4 и блоки **I** и **4a** представляют собой конструкцию If-Then-else с одним входом и одним выходом. Поэтому они заменяются функциональным блоком **II**.

Шаг 4 преобразования

Символ «Решение» с проверкой условия I=3 и блоки **II** и **3а** представляют собой конструкцию If-Then-else с одним входом и одним выходом. Поэтому они заменяются функциональным блоком **III**.

Шаг 5 преобразования

Символ «Решение» с проверкой условия I=2 и блоки **III** и **2а** представляют собой конструкцию If-Then-else с одним входом и одним выходом. Поэтому они заменяются функциональным блоком **IV**.

Шаг 6 преобразования

Символ «Решение» с проверкой условия I=1 и блоки **IV** и **1а** представляют собой конструкцию If-Then-else с одним входом и одним выходом. Поэтому они заменяются функциональным блоком **V**.

Шаг 7 преобразования

Символ «Решение» с проверкой условия I=0 и блок **V** представляют собой конструкцию обобщенного цикла с одним входом и одним выходом. Поэтому они заменяются функциональным блоком **VI**.

Шаг 8 преобразования

Функциональный блок начальной установки I=1, блок **VI** и блок **0** представляют собой конструкцию следования с одним входом и одним выходом. Поэтому они заменяются функциональным блоком **VII**.

Таким образом, за восемь шагов преобразований Бома–Джакопини исходная схема, построенная по методу Ашкрофта–Манны, преобразована в один функциональный блок с одним входом и одним выходом. Это подтверждает, что она является структурированной.

Достоинства метода введения переменной состояния:

1) процесс преобразования программы отличается наглядностью и чёткостью;

2) любому блоку исходной схемы соответствует определённое состояние программы, что помогает выполнять тестирование и отладку программы;

3) метод применим к программам любой структуры (разветвляющимся и циклическим);

4) возможно автоматическое применение данного метода.

Недостатком метода является то, что структурированная форма схемы алгоритма сильно отличается от топологии исходной схемы.

3.3.3. Метод булева признака

Сущность данного метода: в программу, содержащую циклы, вводится некоторый признак. Начальное значение признака задаётся до цикла. Цикл выполняется, пока признак сохраняет своё исходное значение. Значение признака изменяется при наличии некоторых условий внутри цикла.

Рассмотрим применение метода булева признака на примере

преобразования неструктурированной схемы, приведенной на рис. 3.16.

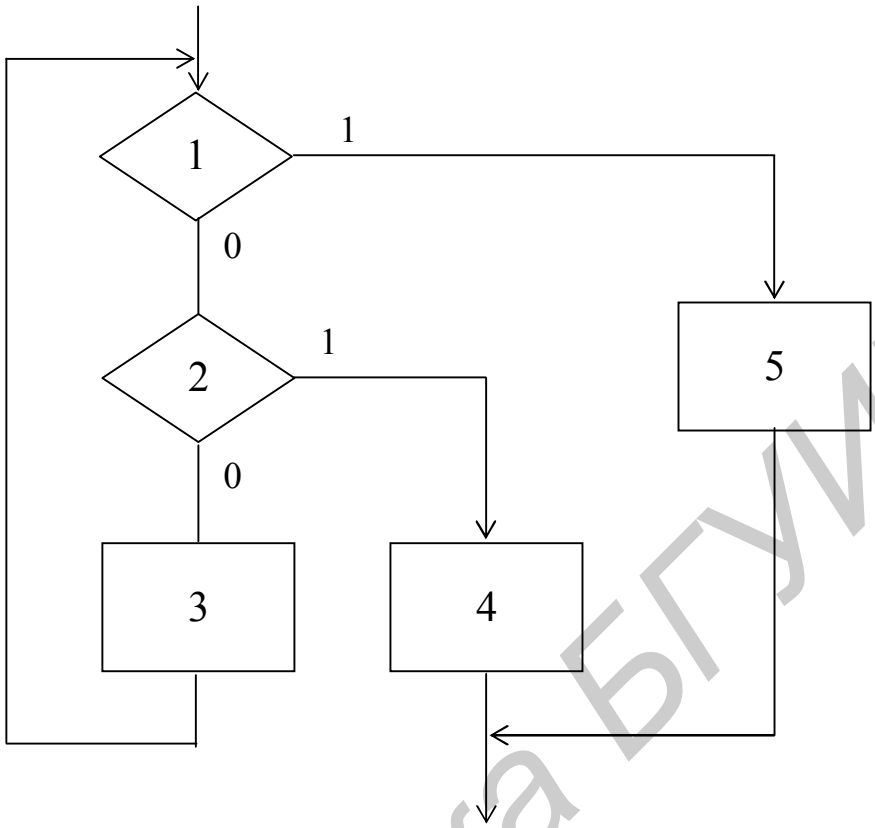


Рис. 3.16. Исходная неструктурированная схема

Схема алгоритма не является структурированной, поскольку входящий в нее цикл (блоки 1, 2, 3) содержит один вход и два выхода. На данной схеме 1 и 2 – некоторые условия, определяющие выполнение того или иного участка вычислений, значения 1 и 0 возле выходов символов «Решение» соответствуют логическим значениям «да» и «нет».

В соответствии с рассматриваемым методом в исходную схему вводится признак (например J). Схема приобретает структурированный вид (рис. 3.17) и легко реализуется конструкциями обобщенного цикла и принятия двоичного решения.

Принцип доказательства того, что данная схема является структурированной, подробно рассмотрен в п. 3.3.1 и 3.3.2. На рис. 3.17 последовательность преобразований Бома–Джакопини представлена схематично: блоки 4, J := 1 → функциональный блок I; 2, 3, I → II; 5, J := 1 → III; 1, II, III → IV; условие J = 1, IV → V; J := 0, V → VI. Каждый из этих шагов преобразований на схеме показан пунктирным функциональным блоком.

Достоинства метода булева признака:

1) компактность, экономичность;
2) топология исходной схемы изменяется незначительно.

Недостаток: метод предназначен для использования только в циклах.

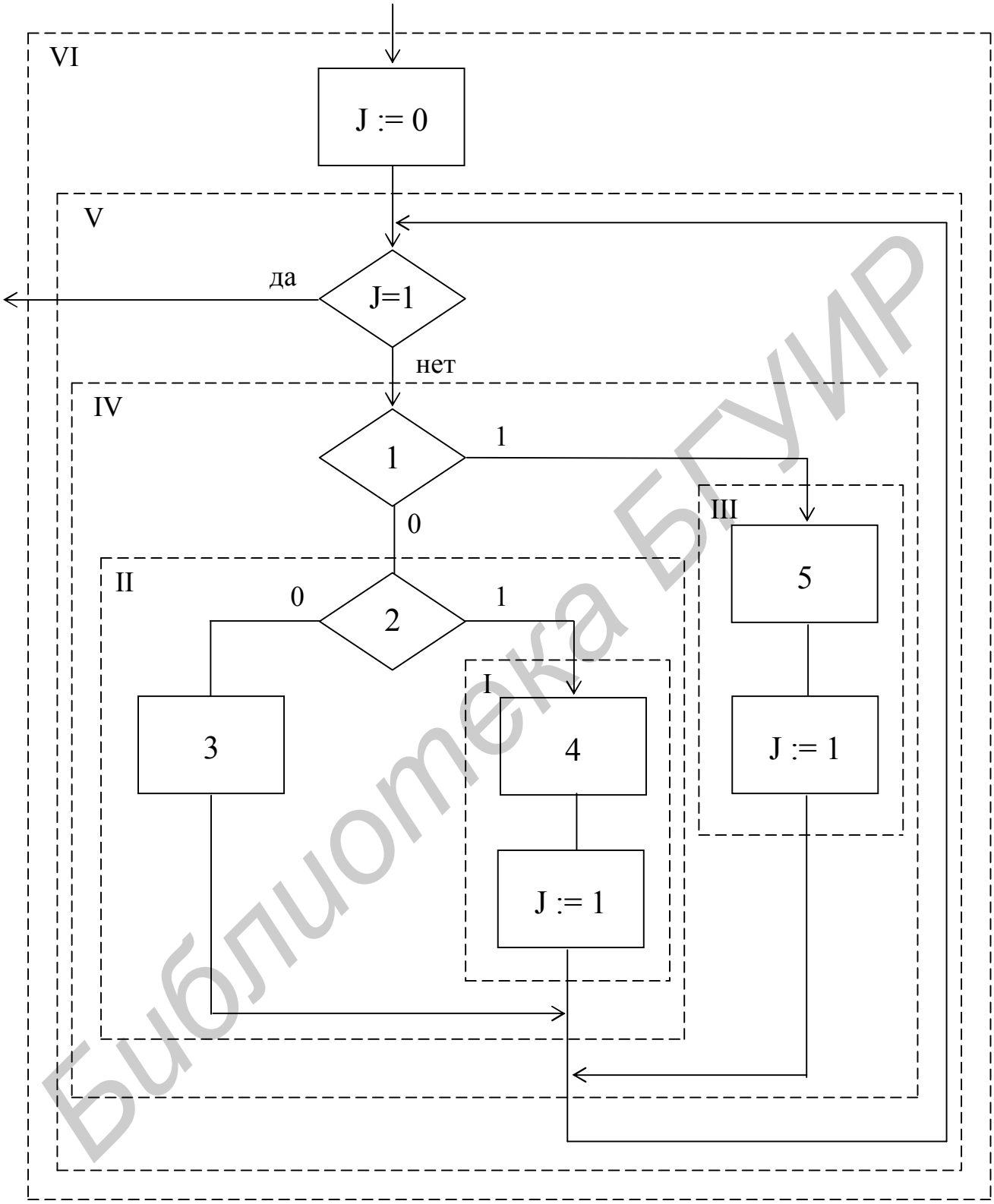


Рис. 3.17. Схема, преобразованная по методу булева признака

Иногда можно обойтись без специального признака, используя те условия, которые уже есть в исходной схеме. Например, исходную

неструктурированную схему (см. рис. 3.16) можно представить так, как показано на рис. 3.18.

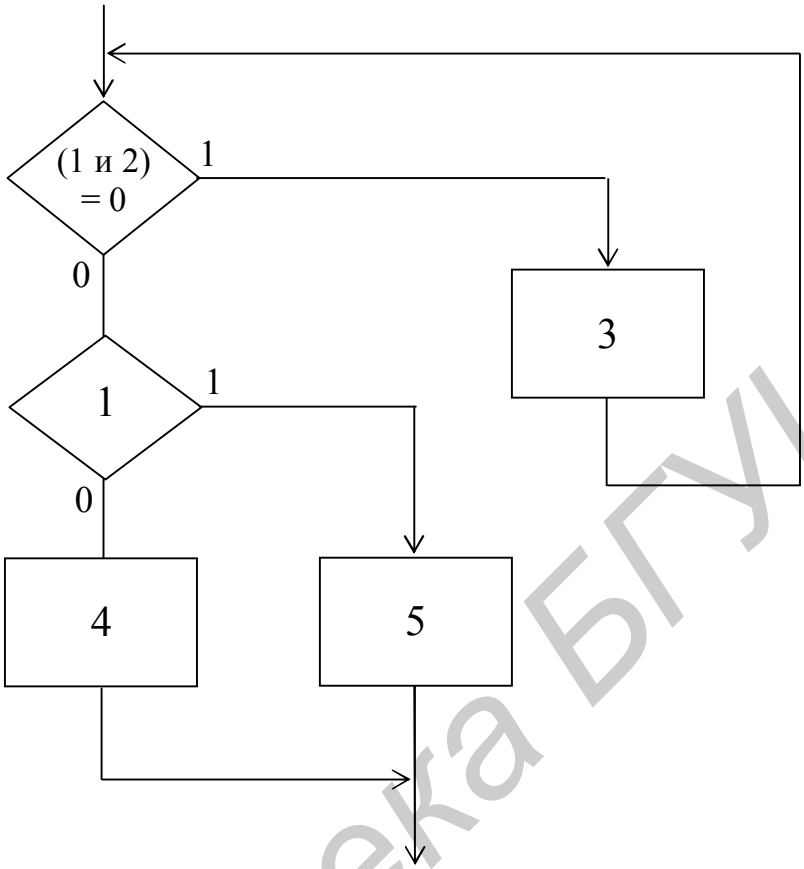


Рис. 3.18. Схема, преобразованная без использования специального признака

На данном рисунке условие «(1 и 2) = 0» означает, что тело цикла 3 будет выполнено в том случае, если оба условия 1 и 2 не выполняются (равны нулю; сравните с исходной неструктурированной схемой). Результирующая схема содержит обобщенный цикл с одним входом и одним выходом и конструкцию If-Then-Else, т.е. является структурированной.

Следует иметь в виду, что преобразование схемы к структурированному виду без дополнительного признака возможно только при небольшом количестве условий.

3.4. Способы графического представления структурированных схем алгоритмов

Многие специалисты в области теории программирования считают, что графическое представление алгоритмов в соответствии с ГОСТ 19.701-90 (ISO 5807-85) скрывает структуру структурированной программы, представляя структурированность недостаточно очевидной. Поэтому для представления структурированных схем алгоритмов были предложены специальные методы

графических обозначений. В данном пособии рассмотрены два из них – метод Дамке и диаграммы Насси–Шнейдермана.

3.4.1. Метод Дамке

М.Дамке предложил для конструкций структурированных схем алгоритмов специальные обозначения.

1. *Функциональный блок* по-прежнему обозначается прямоугольником (рис. 3.19).

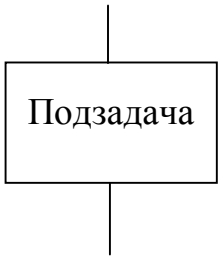


Рис. 3.19. Представление функционального блока по методу Дамке

2. *Конструкция If-Then-Else* изображается так, как показано на рис. 3.20.

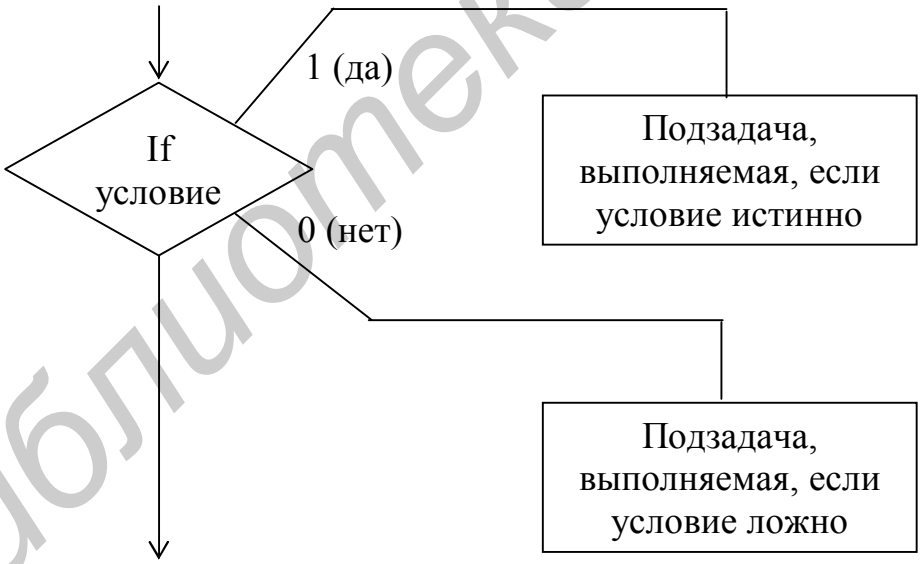


Рис. 3.20. Представление конструкции If-Then-Else по методу Дамке

Элементы с выполняемыми действиями находятся справа от символа «Решение».

3. *Конструкция Do-While* (цикл «Пока») изображается в соответствии с рис. 3.21.

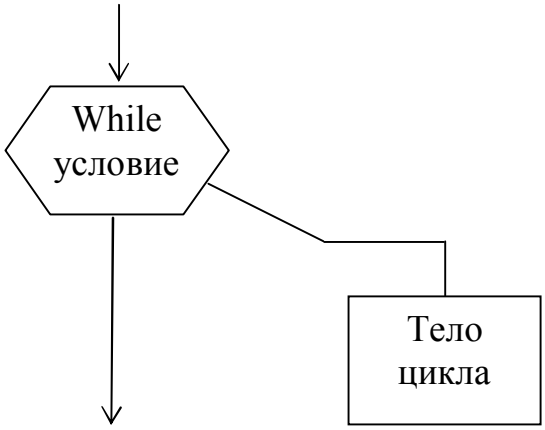


Рис. 3.21. Представление конструкции Do-While по методу Дамке

Тело цикла выполняется до тех пор, пока условие истинно. Условие проверяется первым. Графически это изображается положением шестиугольника **над** выполняемым телом цикла.

Помимо трех основных конструкций в структурном программировании допускаются *дополнительные конструкции*. В методе Дамке они изображаются следующим образом.

4. *Конструкция Repeat-Until* (цикл «До») изображается, как показано на рис. 3.22.

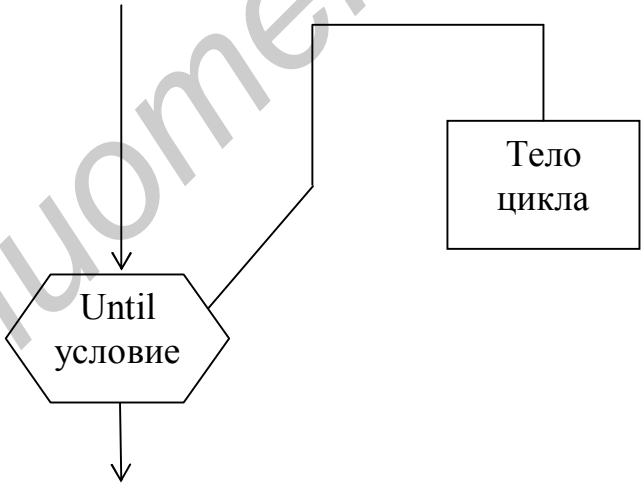


Рис. 3.22. Представление конструкции Repeat-Until по методу Дамке

Если условие истинно, осуществляется выход из цикла. Тело цикла выполняется до проверки условия. Графически это изображается положением тела цикла **над** шестиугольником.

5. *Конструкция цикла с параметром* изображается в соответствии с рис. 3.23.

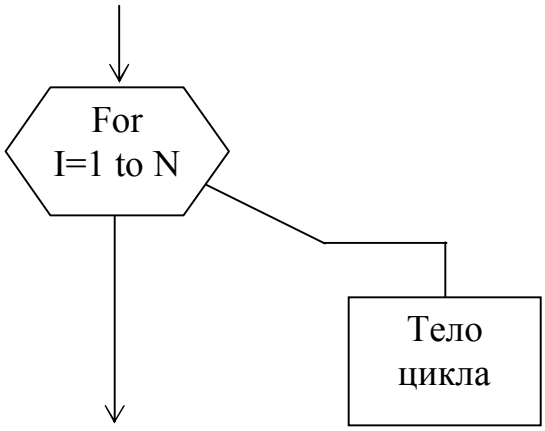


Рис. 3.23. Представление конструкции цикла с параметром по методу Дамке

6. *Конструкция Case* изображается, как показано на рис. 3.24.

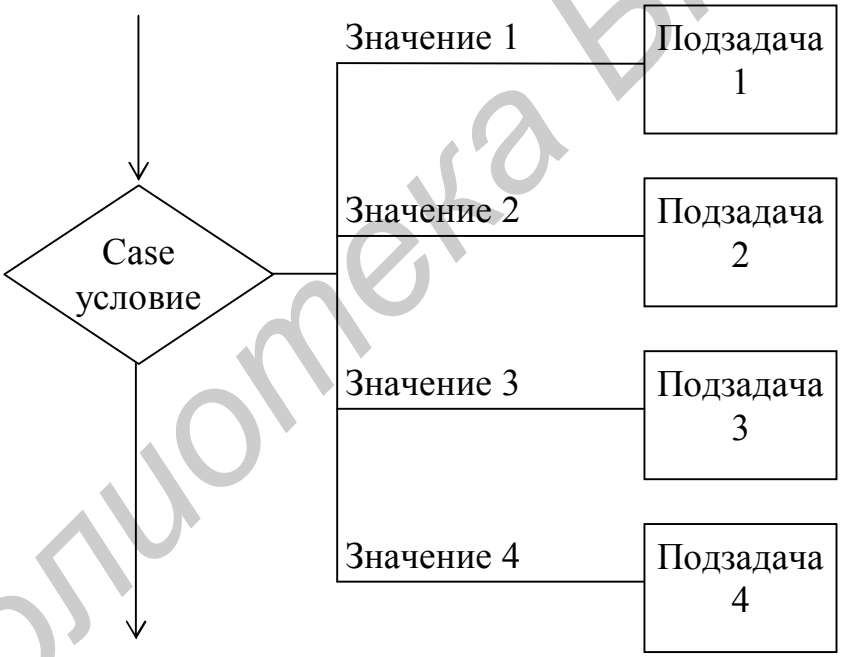


Рис. 3.24. Представление конструкции Case по методу Дамке

Основным принципом при составлении структурированных схем алгоритмов по методу Дамке является *принцип декомпозиции*. Он означает, что любой элемент, представляющий собой задачу, можно разделить на несколько элементов, образующих необходимые подзадачи.

Элементы в самой левой части схемы представляют укрупнённую структуру алгоритма. Затем элементы расширяются вправо по мере разделения каждого элемента на подзадачи. Чтобы исследовать любую подзадачу, достаточно анализировать только те элементы и управляющие структуры,

которые находятся справа от данной подзадачи.

Достоинства метода Дамке:

1) схема алгоритма, представленная с помощью данного метода, нагляднее, чем классическая, особенно для больших программ;

2) метод Дамке удобно использовать при разработке алгоритма по методу нисходящего проектирования;

3) метод Дамке удобен при коллективной разработке программ, так как позволяет проектировать независимо отдельные подзадачи.

Пример 3.1. Дан массив A, состоящий из N элементов. Найти наибольший из элементов массива (Amax) и его номер (Imax).

На рис. 3.25 представлена схема алгоритма решения данной задачи по методу Дамке.

3.4.2. Схемы Насси–Шнейдермана

Схемы Насси–Шнейдермана – это схемы, иллюстрирующие структуру передач управления внутри модуля с помощью вложенных друг в друга блоков.

Схемы используются для изображения структурированных схем и позволяют уменьшить громоздкость схем за счёт отсутствия явного указания линий перехода по управлению.

Схемы Насси–Шнейдермана называют ещё *структурограммами*.

Изображение основных элементов структурного программирования в схемах Насси–Шнейдермана выглядит следующим образом. Каждый блок имеет форму прямоугольника и может быть вписан в любой внутренний прямоугольник любого другого блока. Информация в блоках записывается по тем же правилам, что и в структурированных схемах алгоритмов (на естественном языке или языке математических формул).

1. Функциональный блок (блок обработки)

Изображается в виде прямоугольника (рис. 3.26).

Каждый символ схем Насси–Шнейдермана является блоком обработки (функциональным блоком). Каждый прямоугольник внутри любого символа также является блоком обработки.

2. Блок следования

Изображается, как показано на рис. 3.27.

Объединяет ряд следующих друг за другом процессов обработки.

3. Блок решения

Блок решения используется для представления конструкции If-Then-Else. Изображается в соответствии с рис. 3.28. Условие записывается в центральном треугольнике, варианты исполнения условия – в боковых треугольниках (варианты исполнения могут быть записаны в виде: 1, 0; да, нет; +, -). Процессы обработки обозначаются прямоугольниками.

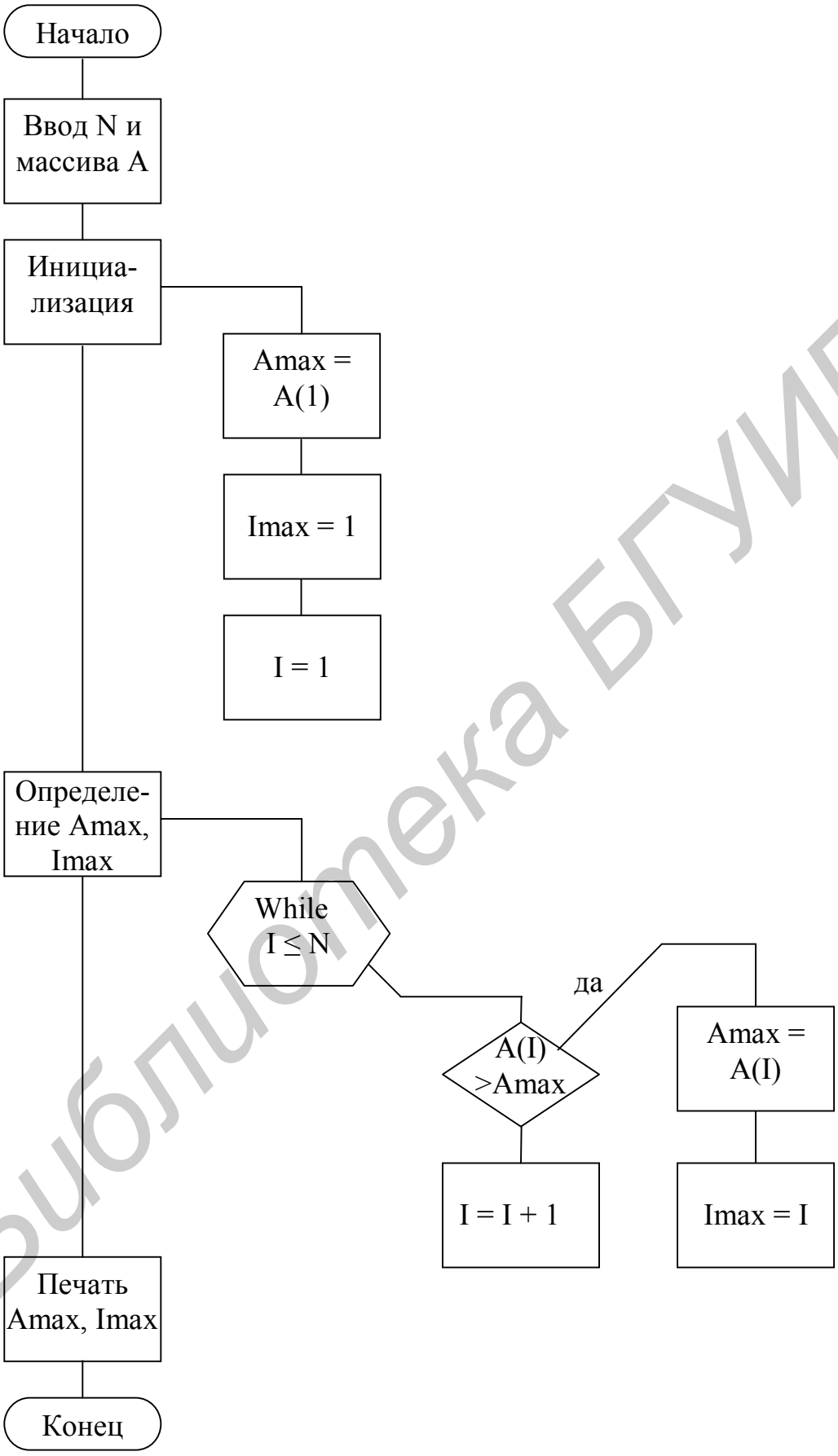


Рис. 3.25. Схема алгоритма поиска максимального элемента массива

Рис. 3.26. Изображение функционального блока в схемах Насси–Шнейдермана

Рис. 3.27. Изображение блока следования в схемах Насси–Шнейдермана

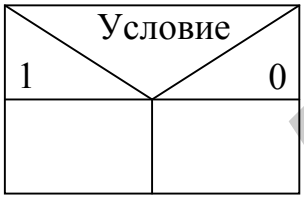


Рис. 3.28. Изображение блока решения в схемах Насси–Шнейдермана

4. Блок Case

Изображается, как показано на рис. 3.29.

Является расширением блока решения. Те варианты выхода из этого блока, которые можно точно сформулировать, размещаются слева от нижней вершины центрального треугольника. Остальные выходы объединяются в один, называемый выходом по несоблюдению условий и расположенный справа от нижней вершины. Если можно перечислить все возможные случаи, правую часть можно оставить незаполненной или совсем опустить, а выходы разместить по обе стороны центрального треугольника.

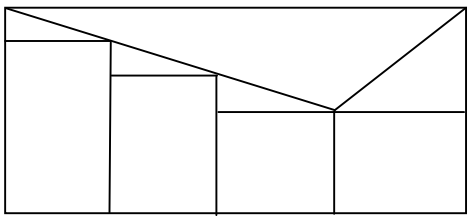


Рис. 3.29. Изображение блока Case в схемах Насси–Шнейдермана

По аналогии с конструкцией If-Then-Else условие записывается в центральном треугольнике, варианты исполнения условия – в боковых треугольниках. Процессы обработки обозначаются прямоугольниками.

5. Цикл «Пока»

Изображается в соответствии с рис. 3.30.

Обозначает циклическую конструкцию с предусловием, т.е. с проверкой условия в начале цикла (цикл While). Условие выполнения цикла размещается в верхней полосе.

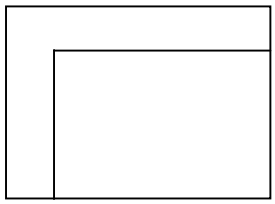


Рис. 3.30. Изображение циклической конструкции с предусловием в схемах Насси–Шнейдермана

6. Цикл «До»

Изображается в соответствии с рис. 3.31.

Обозначает циклическую конструкцию с постусловием, т.е. с проверкой условия после выполнения тела цикла (цикл Repeat-Until). Условие выполнения цикла размещается в нижней полосе.

Рис. 3.31. Изображение циклической конструкции с постусловием в схемах Насси–Шнейдермана

На рис. 3.32–3.34 приведены примеры простейших структурированных схем алгоритмов (слева) и соответствующих им схем Насси–Шнейдермана (справа). На данных рисунках Bi обозначает i-е условие, Di – i-е действие.

Пример 3.2. Дан массив A, состоящий из N элементов. Найти наибольший из элементов массива (Amax) и его номер (Imax).

Укрупненная и детализированная схемы Насси–Шнейдермана, реализующие эту задачу, представлены на рис. 3.35.

Схема алгоритма решения этой же задачи, представленная по методу Дамке, рассмотрена в предыдущем разделе (см. рис. 3.25).

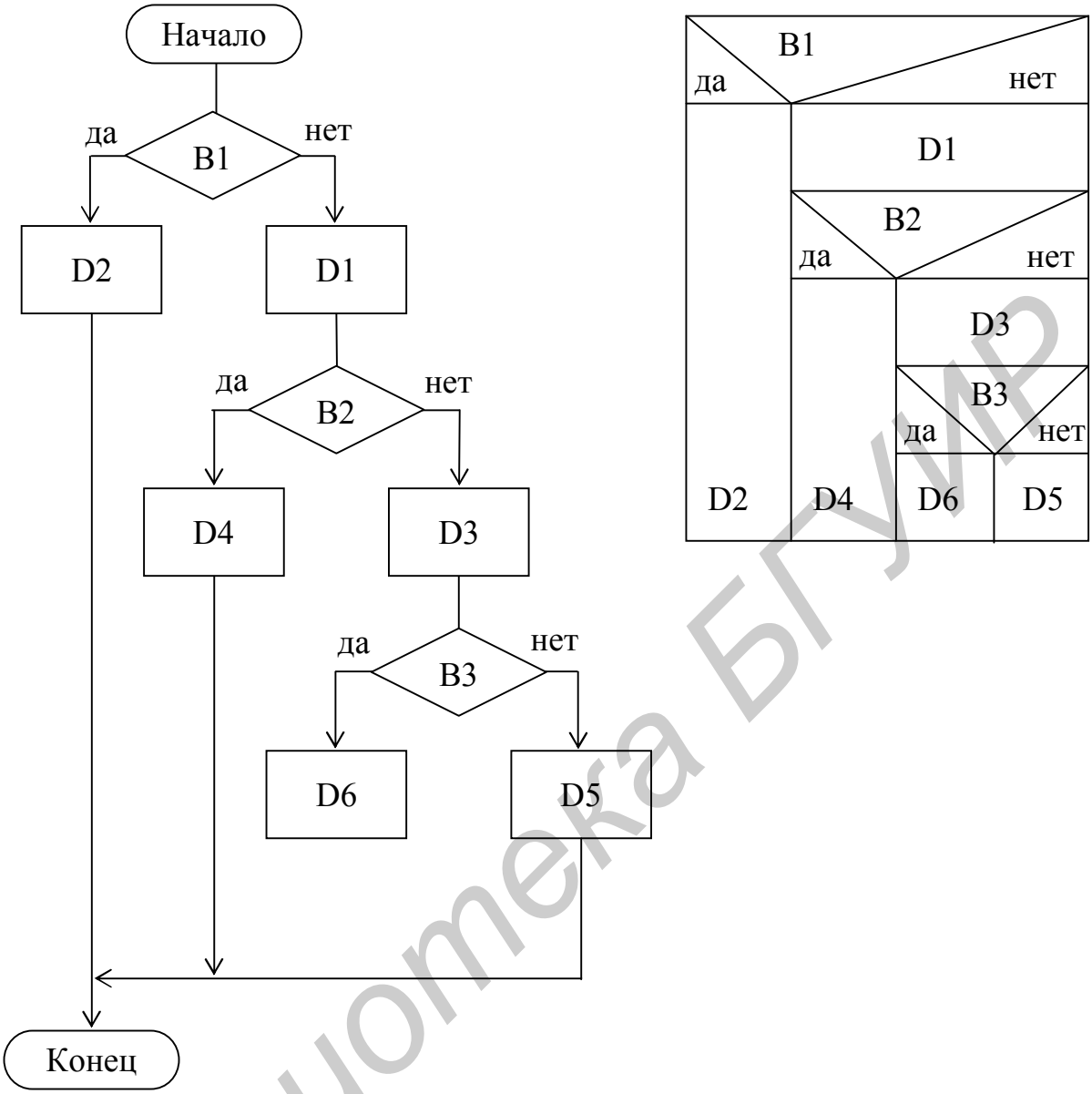


Рис. 3.32. Пример схемы Насси–Шнейдермана
для алгоритма с разветвлениями

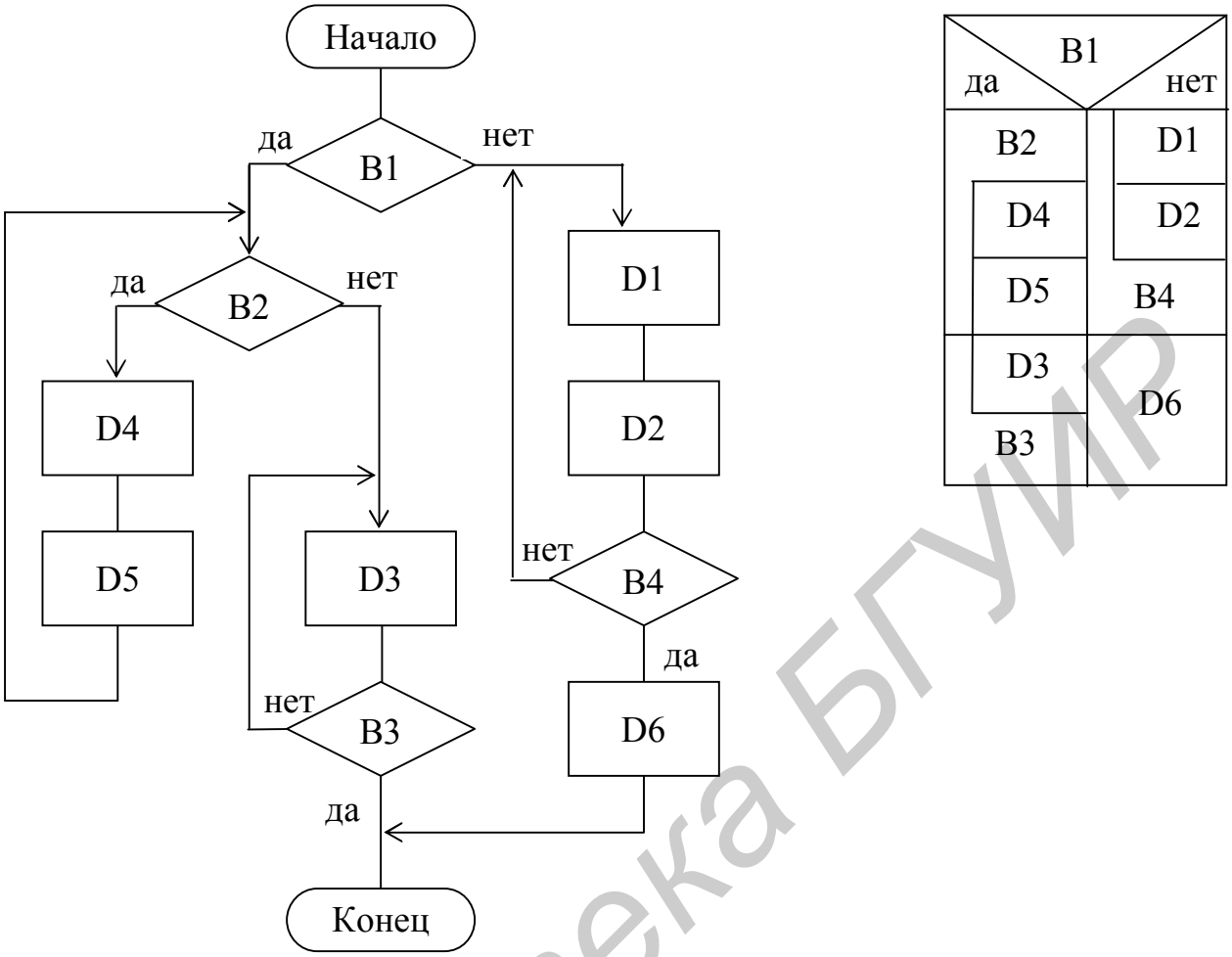


Рис. 3.33. Пример схемы Насси–Шнейдермана
для алгоритма, содержащего последовательность циклов

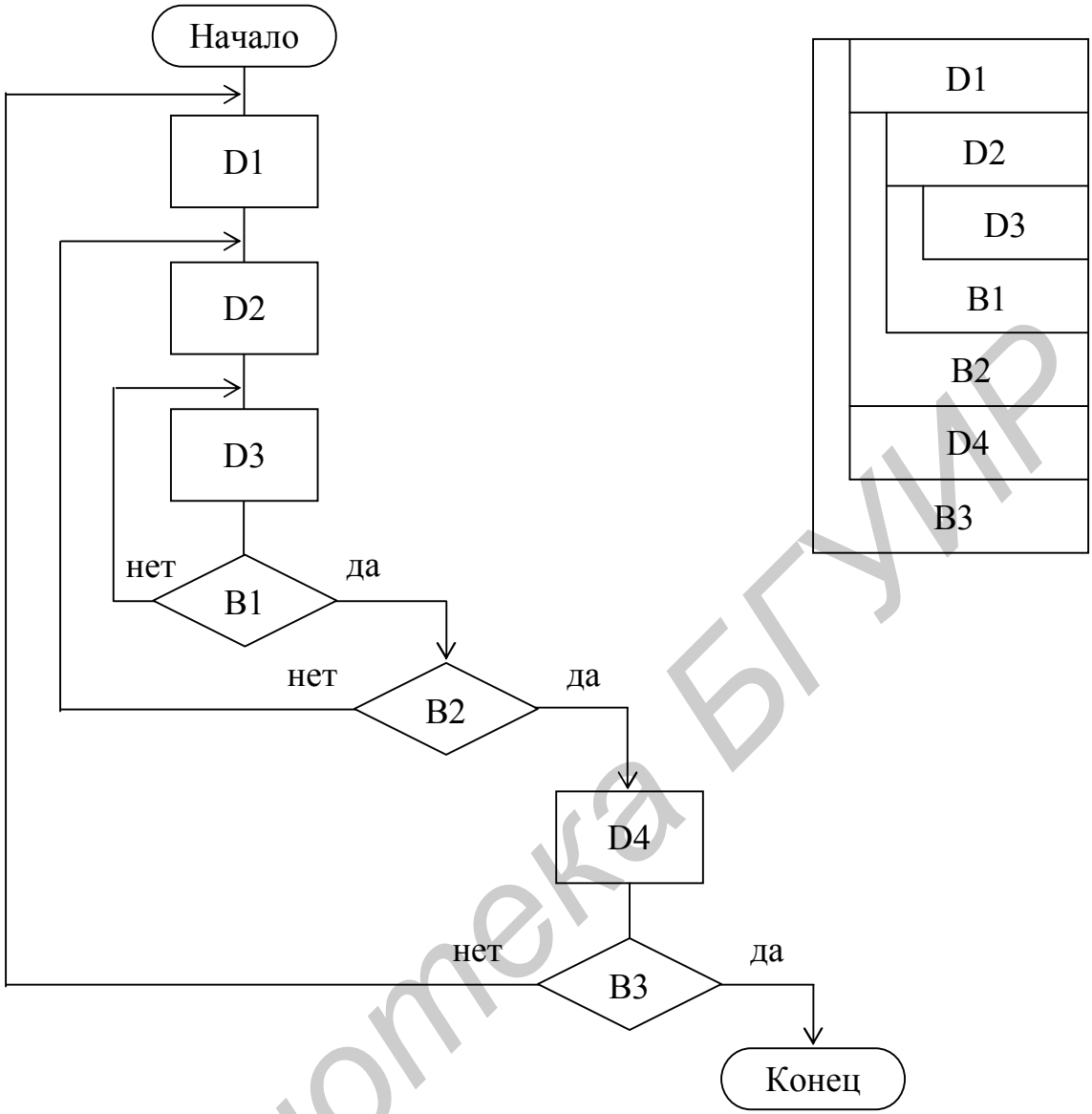


Рис. 3.34. Пример схемы Насси–Шнейдермана
для алгоритма, содержащего вложенные циклы

Укрупненная схема
Насси–Шнейдермана

Подробная схема
Насси–Шнейдермана

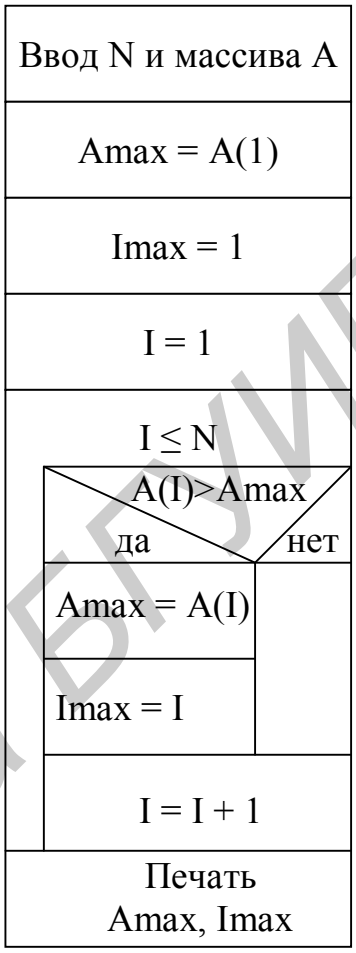


Рис. 3.35. Диаграмма Насси–Шнейдермана для алгоритма поиска максимального элемента массива и его номера

4. СТРУКТУРИРОВАНИЕ И ПРОЕКТИРОВАНИЕ ПРОГРАММ НА ЯЗЫКЕ ПАСКАЛЬ

4.1. Структурирование и оформление программ на языке Паскаль

Как отмечалось в подразд.1.4, язык Паскаль является структурированным языком, т.е. позволяет писать хорошо структурированные программы.

Структурирование программ, написанных на языке Паскаль, достигается за счет следующих факторов:

1) разбиение программы на отдельные разделы (заголовок, раздел описаний, раздел операторов);

2) наличие операторов языка, реализующих блоки структурного программирования:

- оператора присваивания, реализующего функциональный блок;
- условного оператора, реализующего конструкцию принятия двоичного решения;
- оператора цикла с предусловием (While), реализующего конструкцию обобщенного цикла «Пока»;

3) наличие составного оператора Begin **...** End и синтаксиса условного оператора и оператора цикла (возможность использования в них группы операторов за счет объединения их в операторные скобки), что позволяет реализовывать преобразования Бома–Джакопини;

4) наличие дополнительных конструкций организации цикла: цикла «До» (реализуется оператором цикла с постусловием Repeat **...** Until) и цикла с параметром (реализуется оператором For);

5) наличие подпрограмм (реализованных в виде процедур и функций).

Для достижения наглядности программы, для отражения вложенности управляющих структур друг в друга используется *правильное расположение текста отдельных операторов*.

Общее правило записи текста программы: служебные слова, которыми начинается и заканчивается тот или иной оператор, записываются на одной вертикали; все вложенные в него операторы записываются с отступом вправо. Это правило полностью соответствует принципу построения структурированных схем алгоритмов.

Синтаксис языка Паскаль позволяет писать несколько операторов в строке программы. Однако для удобства понимания и отладки программы желательно писать по одному оператору в строке.

4.2. Методология нисходящего проектирования программы, не использующая предварительную разработку схемы алгоритма

Язык Паскаль удобен для реализации *методов нисходящего проектирования программ,* и в частности *метода пошаговой детализации (декомпозиции).*

В разд. 2 и 3 описаны методы нисходящего проектирования программ, основанные на предварительной разработке схем алгоритмов.

В то же время существуют методологии пошагового проектирования программы, не использующие предварительную разработку схемы алгоритма.

Одна из таких методологий основана на использовании комментариев. Ее сущность заключается в следующем.

На каждом этапе нисходящего проектирования используются управляющие структуры и зарезервированные слова языка Паскаль. Правила обработки данных не детализируются, а описываются в виде комментариев.

На последующих этапах нисходящего проектирования блоки, представленные комментариями, частично детализируются, но сами комментарии не выбрасываются и т.д. В результате после окончания проектирования получается хорошо прокомментированная программа.

Комментарии в такой программе обычно делятся на следующие виды:

1) «заголовки» – объясняют назначения основных блоков программы на отдельных этапах пошаговой детализации;

2) «построчные» комментарии – описывают мелкие фрагменты программы;

3) «вводные» комментарии – помещаются в начале текста программы и задают общую информацию о программе (например, назначение программы, сведения об авторе, дата написания, используемый метод решения, время выполнения, требуемый объем памяти и т.п.).

Комментарии являются одним из наиболее эффективных средств облегчения понимания, тестирования, отладки и сопровождения программ. Отсутствие комментариев в программе является одним из признаков дилетантского подхода к программированию.

Пример 4.1. Использование упрощенного варианта методологии пошагового проектирования программы, основанной на использовании комментариев. Вычислить значение функции

$$Y = \mathrm{Sin}\,(X) = X - X^3 / 3! + X^5 / 5! - X^7 / 7! + \mathbf{...}$$

с точностью Eps = 0,0001.

1-й этап нисходящего проектирования

На данном этапе записывается вводный комментарий и определяется

укрупненная структура программы (в виде комментариев).

```
{Вычисление значения Y = Sin(X) с заданной точностью с помощью разложения функции в ряд. Разработчик – Иванов А. А. 20.2.03г.}
{Заголовок программы}
    {Описания}
    {Вычисления}
```

2-й этап нисходящего проектирования

На данном этапе оставляются предыдущие комментарии и, возможно, вводятся некоторые служебные слова языка Паскаль. Вводятся новые комментарии.

```
{Вычисление значения Y = Sin(X) с заданной точностью с помощью разложения функции в ряд. Разработчик – Иванов А. А. 20.2.03г.}
{Заголовок программы}
Program SinX;
    {Описания}
    {Вычисления}
    Begin
        {Чтение исходных данных}
        {Присвоение начальных значений}
        While  {Очередное слагаемое больше точности}
            Do  {Вычисление слагаемого и суммы ряда}
        {Вывод результатов}
    End.
```

3-й этап нисходящего проектирования

Оставляются предыдущие комментарии. Раздел «Описания» пока не изменяется (он будет детализироваться после детализации раздела «Вычисления»). Детализируется раздел «Вычисления». Вводятся служебные слова языка Паскаль и новые комментарии. Возможно введение построчных комментариев для пояснения отдельных операторов программы.

```
{Вычисление значения Y = Sin(X) с заданной точностью с помощью разложения функции в ряд. Разработчик – Иванов А. А. 20.2.03г.}
{Заголовок программы}
Program SinX;
    {Описания}
    {Вычисления}
    Begin
        {Чтение исходных данных}
        Readln (X, Eps);
        {Присвоение начальных значений}
        Y := X;                                  {Y – сумма ряда}
        N := 2;                                  {N – вспомогательная
```

```
                                        переменная для вычисления
                                        знаменателя}
    Vs := X;                            {Vs – очередное слагаемое}
    While  Abs(Vs) >= Eps       {Очередное слагаемое больше точности}
       Do                       {Вычисление слагаемого и суммы ряда}
          Begin
             Vs := –Vs*X*X/(2*N–1)/(2*N–2);   {Вычисление слагаемого}
             N := N + 1;
             Y := Y + Vs                      {Вычисление суммы}
          End;
    {Вывод результатов}
    Writeln (X, Y, Eps)
End.
```

Комментарии-заголовки принято писать перед соответствующим блоком программы, построчные комментарии – за соответствующим оператором.

4-й этап нисходящего проектирования

Оставляются предыдущие комментарии. Детализируется раздел «Описания». Вводятся описания элементов программы, используемых в разделе «Вычисления».

```
{Вычисление значения Y = Sin(X) с заданной точностью с помощью
разложения функции в ряд. Разработчик – Иванов А. А. 20.2.03г.}
{Заголовок программы}
Program SinX;
   {Описания}
   Var
      X, Y, Eps, Vs: Real;
      N: Integer;
   {Вычисления}
   Begin
      {Чтение исходных данных}
      Readln (X, Eps);
      {Присвоение начальных значений}
      Y := X;  {Y – сумма ряда}
      N := 2;   {N – вспомогательная переменная для вычисления знаменателя}
      Vs := X;  {Vs – очередное слагаемое}
      While  Abs(Vs) >= Eps       {Очередное слагаемое больше точности}
         Do                       {Вычисление слагаемого и суммы ряда}
            Begin
               Vs := –Vs*X*X/(2*N–1)/(2*N–2);   {Вычисление слагаемого}
               N := N + 1;
               Y := Y + Vs          {Вычисление суммы}
            End;
```

```
    {Вывод результатов}
    Writeln (X, Y, Eps)
End.
```

Данный пример иллюстрирует проектирование программы с использованием метода пошаговой детализации без предварительной разработки схемы алгоритма, правила комментирования программы и правила взаимного расположения исходного текста программы.

Очевидно, что при проектировании более сложных программ необходимо использование гораздо большего количества этапов.

КОНТРОЛЬНЫЕ ВОПРОСЫ ДЛЯ САМОСТОЯТЕЛЬНОЙ ПОДГОТОВКИ

Тема 1

1. Какую информацию содержит команда?
2. Что (или кто) управляет последовательностью выполнения команд в компьютере?
3. Какие группы языков программирования вы знаете?
4. Что такое машинно-ориентированные языки программирования?
5. Чем определяется уровень языка программирования?
6. К какой группе языков относится язык Паскаль?
7. Что такое система программирования?
8. Из каких компонентов состоит система программирования?
9. Какие виды модулей различают?

Тема 2

10. Что такое алгоритм?
11. Что такое метаязык?
12. Что такое схема алгоритма?
13. Что представляет собой символ в схемах алгоритмов?
14. Какие группы символов используются в схемах алгоритмов?
15. Как изображается символ, предназначенный для представления ввода/вывода данных?
16. Какой символ предназначен для отображения функции обработки данных?
17. Какой символ отображает функцию переключательного типа?
18. С помощью каких символов можно отобразить цикл?
19. Какой символ отображает выход во внешнюю среду и вход из внешней среды?
20. С помощью какого символа отображается предопределенный процесс, состоящий из одной или нескольких операций или шагов программы, которые определены в другом месте?
21. Какой символ отображает выход в другую часть схемы и вход из другой части этой схемы?
22. Как называется и как изображается символ, отображающий любой символ из групп символов процесса или символов данных и указывающий имя схемы с подробным представлением данного символа?
23. Назовите стандартное направление потока на схемах алгоритмов.
24. С каких сторон линии в схемах должны подходить к символу и выходить из него?

25. Как отображается вход из части той же схемы, расположенной на другой странице?

26. Какой вычислительный процесс называется линейным?

27. Какой вычислительный процесс называется разветвляющимся?

28. Какой вычислительный процесс называется циклическим?

29. Назовите классификацию циклов в соответствии с их взаимным расположением.

30. Назовите классификацию циклов в зависимости от местоположения условия выполнения цикла.

31. Какой циклический процесс называется итерационным?

Тема 3

32. Какие алгоритмы и программы называются структурированными?

33. Перечислите базовые конструкции структурного программирования.

34. В чем сущность преобразований Бома–Джакопини?

35. Что является средством доказательства структурированности программы или алгоритма?

36. В чем сущность метода дублирования кодов?

37. В чем сущность метода введения переменной состояния?

38. В чем сущность метода булева признака?

39. Как изображается конструкция Do-While в схемах алгоритмов по методу Дамке?

40. Как изображается конструкция Repeat-Until в схемах алгоритмов по методу Дамке?

41. Как изображается конструкция If-Then-Else в схемах алгоритмов по методу Дамке?

42. Как изображается декомпозиция конструкции в схемах Насси–Шнейдермана?

43. Как изображается конструкция If-Then-Else в схемах Насси–Шнейдермана?

44. Как обозначаются конструкции цикла с предусловием и с постусловием в схемах Насси–Шнейдермана?

Тема 4

45. За счет каких средств языка Паскаль достигается структурирование программ?

46. В чем заключается методология пошагового проектирования программы, основанная на использовании комментариев?

47. Назовите виды и нормы комментариев.

48. Какие правила используются для отражения вложенности управляющих структур друг в друга в тексте программ?

ЛИТЕРАТУРА

1. ГОСТ 19.701-90. Единая система программной документации. Схемы алгоритмов, программ, данных и систем. Условные обозначения и правила выполнения.

2. ГОСТ 19781-90. Обеспечение систем обработки информации программное. Термины и определения.

3. Бахтизин В.В., Глухова Л.А. и др. Методические указания по вычислительной практике и самостоятельной работе по курсам «Программирование» и «Конструирование программ и языки программирования» для студентов специальности «Вычислительные машины, комплексы, системы и сети», «Программное обеспечение ЭВМ и автоматизированных систем» и слушателей спецфакультета переподготовки по направлению «Микропроцессорные системы». Ч. 1–4. – Мн.: МРТИ, 1989–1992.

4. Бахтизин В.В., Глухова Л.А. Лабораторный практикум по курсам «Конструирование программ и языки программирования» и «Программирование» для студентов специальностей «Программное обеспечение ЭВМ и автоматизированных систем», «Вычислительные машины, комплексы, системы и сети». Ч. 1: Конструирование программ с использованием процедур. – Мн.: МРТИ, 1993.

5. Бахтизин В.В., Глухова Л.А. Лабораторный практикум по курсам «Конструирование программ и языки программирования» и «Программирование» для студентов специальностей «Программное обеспечение ЭВМ и автоматизированных систем», «Вычислительные машины, комплексы, системы и сети». Ч. 3: Конструирование программ с использованием функций. – Мн.: МРТИ, 1995.

6. Введение в язык Паскаль: Учеб. пособие / В.Г. Абрамов, Н.П. Трифонов, Г.П. Трифонова. – М.: Наука, 1988.

7. Зиглер К. Методы проектирования программных систем. – М.: Мир, 1985.

8. Фаронов В.В. Турбо Паскаль 7.0. Начальный курс: Учеб. пособие. – М.: Нолидж, 1999.

9. Электронные вычислительные машины: Практ. пособие для вузов / Под ред. А.Я. Савельева: В 8 кн. Кн. 2: Основы информатики. – М.: Высш. шк., 1991.

10. Электронные вычислительные машины: Практ. пособие для вузов / Под ред. А.Я. Савельева: В 8 кн. Кн. 3: Технология подготовки задач для решения на ЭВМ. – М.: Высш. шк., 1991.

11. Электронные вычислительные машины: Практ. пособие для вузов / Под ред. А.Я. Савельева: В 8 кн. Кн. 3: Алгоритмизация и основы программирования. – М.: Высш. шк., 1987.

Св. план 2003, поз.36

Учебное издание

Глухова Лилия Александровна,
Бахтизин Вячеслав Вениаминович

ОСНОВЫ АЛГОРИТМИЗАЦИИ И СТРУКТУРНОГО ПРОЕКТИРОВАНИЯ ПРОГРАММ

Учебное пособие
по курсам
«Основы алгоритмизации и программирования»
и «Технология разработки программного обеспечения»
для студентов специальности 40 01 01
«Программное обеспечение информационных технологий»
дневной формы обучения

Редактор Т.А. Лейко
Корректор Е.Н. Батурчик
Компьютерная верстка Т.В. Шестакова

Подписано в печать 1.10.2003. Формат 60x84 1/16. Бумага офсетная.
Печать ризографическая. Гарнитура «Таймс». Усл. печ. л. 4,3.
Уч.-изд. л. 4,2. Тираж 100 экз. Заказ 228.

Издатель и полиграфическое исполнение:
Учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники».
Лицензия ЛП № 156 от 30.12.2002.
Лицензия ЛВ № 509 от 03.08.2001.
220013, Минск, П. Бровки, 6.