ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«ОРЕНБУРГСКИЙ ГОСУДАРСТВЕННЫЙ АГРАРНЫЙ УНИВЕРСИТЕТ»

А. Д. Тарасов, А. Г. Матвеев

МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ К УЧЕБНОЙ ПРАКТИКЕ

ПО ПОЛУЧЕНИЮ ПЕРВИЧНЫХ ПРОФЕССИОНАЛЬНЫХ УМЕНИЙ, В ТОМ ЧИСЛЕ ПЕРВИЧНЫХ УМЕНИЙ И НАВЫКОВ НАУЧНО-ИССЛЕДОВАТЕЛЬСКОЙ ДЕЯТЕЛЬНОСТИ

Специальность 10.05.03 Информационная безопасность автоматизированных систем

Специализация Информационная безопасность автоматизированных систем критически важных объектов

Форма обучения очная

Оренбург

Издательский центр ОГАУ

2015

УДК 198.2
М 33

Рекомендовано к изданию редакционно-издательским советом Оренбургского государственного аграрного университета (председатель совета – Г.В. Петрова)

Издано в авторской редакции

Тарасов А. Д., Матвеев А. Г.
Методические рекомендации к учебной практике по программированию / А. Д. Тарасов, А. Г. Матвеев. – Оренбург: Издательский центр ОГАУ, 2015. – 26 с.

Методические рекомендации раскрывают цели и задачи учебной практики по программированию, тематику и правила оформления расчетно-пояснительной записки. Предлагаемые задачи соответствуют учебной программе дисциплины. Приведены примеры решения типовых задач.

1. Цели и задачи учебной практики по программированию

Изучение дисциплины «Языки программирования» является важной частью профессиональной подготовки студентов, обучающихся по направлениям подготовки «Информационная безопасность автоматизированных систем».

Цель учебной практики по программированию заключается в совершенствовании навыка программирования, позволяющего решать задачи на компьютере, в закреплении и демонстрации знаний, полученных при изучении курса «Языки программирования» и «Алгоритмические языки и программирование». Выполнение задач учебной практики требует знаний материала информатики, высшей математики, умения программирования на языке “Паскаль” версии 6.0 или выше, творческого подхода в составлении алгоритма поставленных задач.

Задачи учебной практики заключаются:

- в формировании навыков использования методов алгоритмизации для подготовки задач к решению на компьютере;
- в совершенствовании навыков работы с компьютером при решении поставленных задач;
- в закреплении знаний языков программирования на основе языка “Паскаль” и общего принципа программирования помогающего при изучении других языков.
- в выработке умения самостоятельно изучать состояние поставленной в задании проблемы, анализировать изученный материал, привлекать для решения задачи знания из различных учебных дисциплин и принимать обоснованные решения;

Основными этапами учебно-вычислительной практики являются:

- анализ поставленной задачи, выбор метода решения.
- составление алгоритма поставленной задачи.
- составление программы на компьютере.
- оформление расчетно-пояснительной записки.
- защита.

Основными документами для организации учебной практики являются стандарты, учебные программы. Задания на практику должны быть индивидуальными и примерно одинаковыми по степени сложности. Задание содержит одну задачу и выдается студенту независимо от текущей успеваемости. Задание подписывает руководитель проекта и утверждает заведующий кафедрой. Кафедра должна обеспечивать студентов соответствующей методической и справочной литературой.

2. Оформление расчетно-пояснительной записки.

Учебная практика включает в себя расчетно-пояснительную записку объёмом до 10 листов размером А4, в редакторе Word, оформленную по обычным правилам курсовых проектов:

шрифт: размер – 14, Times New Roman, начертание – обычный, выравнивание по ширине;

абзац: красная строка 1.25, междустрочный интервал одинарный;

поля: сверху и снизу – 1.5 см, слева – 3 см., справа – 1 см;

нумерация страниц внизу страницы, справа, первая страница - титульный лист не нумеруется.

Пояснительная записка должна содержать следующие разделы:

1. Титульный лист.
2. Задание на практику.
3. Алгоритм решения задачи.
4. Блок-схема алгоритма.
5. Текст программы на языке “Паскаль”.

Образец титульного листа приведен в приложении 1.

Задание, выданное руководителем практики, записывается на втором листе пояснительной записки (на титульном листе тема не пишется). Текст задания должен полностью совпадать с выданным.

Алгоритм представляет собой последовательность действий необходимых для решения задачи оформленную в виде нумерованного списка. Для самых простых задач достаточно одноуровневого списка. Обычно необходимо использовать подпункты, например:

1) Задать переменную x.
2) если x>0 тогда
2.1) y=ln(x)
2.2) вывести y на экран
3) если x<=0 тогда
3.1) вывести сообщение “x должен быть больше 0”
3.2) вернуться к моменту задания переменной x – пункт 1

Если в задаче используются математические методы явно не указанные в задании, необходимо перед алгоритмом записать общий принцип работы метода. Метод решения оформляется простым текстом. При необходимости описание метода может включать в себя формулы, рисунки или таблицы оформленные по стандартам рефератов. Далее в самом алгоритме метод расписывается по пунктам как процесс решения конкретной задачи.

Блок-схема это графическое отображение алгоритма. Блок-схема представляет собой совокупность геометрических фигур соединенных между

собой связями (линиями потока), отражающими последовательность выполнения действий.

Геометрические фигуры называют блоками. Каждый блок имеет определенное смысловое значение и отображает некоторый этап решения задачи. Начертание блоков, их размеры и отображаемые ими функции определены соответствующим ГОСТом.

Текст внутри блока должен отражать пункт алгоритма, записанный в математической форме. Текст не должен содержать правила оформления, используемые в каком-либо языке программирования. Недопустимо использовать ключевые слова языка, нестандартные знаки препинания, имена функций не принятые в математике и любые другие символы, используемые только в программе, а не в алгоритме, например:

‘readln(x)’ вместо ‘ ввод x’

‘y : = ln(x)’ вместо ‘y = ln(x)’

‘y = sqr(x)’ вместо ‘$y = x^2$’

Блок-схема, размещенная на нескольких листах должна содержать соединительные блоки.

Если в программе используется пользовательская процедура или функция, допустимо оформление отдельной блок-схемы для процедуры и обозначение в основной блок–схеме процесса вызова процедуры в виде одного блока действия.

Допускается применять свой размер шрифта в блок-схемах, если текст не помещается в блоках.

Блок-схемы должны быть выполнены четко, ровно, в черно-белом исполнении, хорошо видимые на печати. Необходимо соблюдать все правила оформления. Подписывать блок-схему как рисунок не нужно.

Текст программы должен полностью совпадать с представляемой на защиту рабочей версией программы. Из-за особенностей текста допустимо не использовать красные строки и выравнивание по ширине.

Блок-схема и текст программы должны начинаться с новой страницы.

Пояснительная записка распечатывается на бумаге форматом А4, приносится в день защиты и сдается в скоросшивателе. На защиту также выносится рабочая версия программы на любом электронном носителе. Программа должна быть записана на носитель только в виде текста – файла с расширением .PAS, в других форматах, например исполняемый файл, программа не принимается и не проверяется. Руководитель должен иметь возможность открыть программу в текстовом редакторе языка “Паскаль”, и запустить на выполнение. Электронная версия записки, а также программа на электронном носителе не сдается. При наличии ошибок в оформлении или при защите на неудовлетворительную оценку записка возвращается студенту и приносится в день повторной защиты.

3. Блок-схемы

3.1 Обозначение блоков

Рассмотрим обозначение основных блоков, применяемых в блок-схемах алгоритмов.

1) Для обозначения начала, окончания или прерывания процесса выполнения программы используют блоки НАЧАЛО-КОНЕЦ (рис. 1).

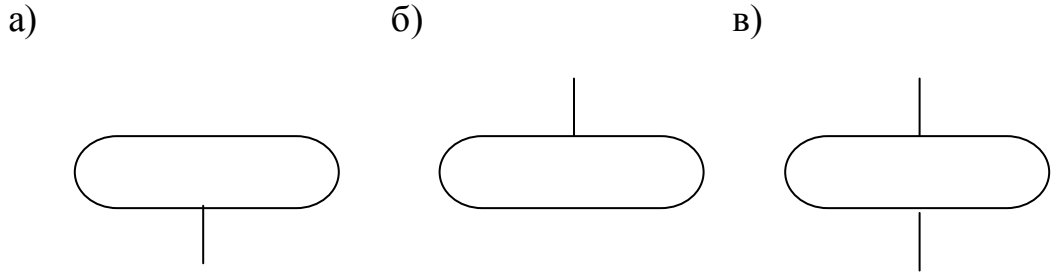

Рисунок 1 – Изображение блоков группы НАЧАЛО-КОНЕЦ:
а) начало; б) конец; в) прерывание

2) Процесс ввода данных пользователем или отображения результатов, изображается с помощью блока ВВОД-ВЫВОД (рис. 2).

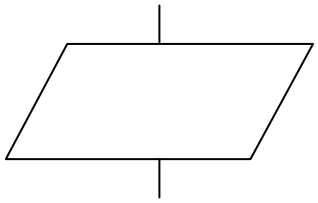

Рисунок 2 – Изображение блока ВВОД-ВЫВОД

3) Блок, в котором происходит обработка данных (выполнение операции или группы операций) и размещение результатов обработки в ячейки памяти (например, y = 5, y = 2 x + 3 и т. д.), носит название ПРОЦЕСС (блок ДЕЙСТВИЯ) (рис. 3).

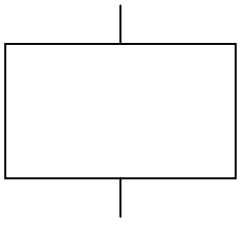

Рисунок 3 – Изображение блока ПРОЦЕСС

Блоком действия также обозначают операции, для которых не применимы остальные виды блоков, например вывод графического изображения.

4) Процесс разветвления алгоритма в зависимости от некоторого условия изображается с помощью блока УСЛОВИЕ (рис. 4).

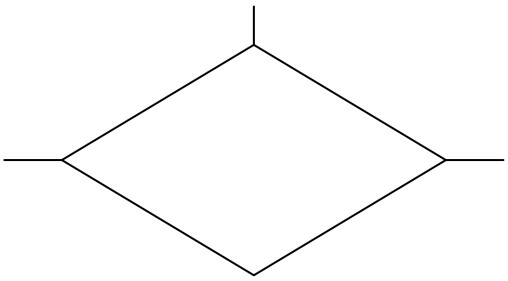

Рисунок 4 – Изображение блока УСЛОВИЕ

5) Цикл, в котором заранее известно число шагов (цикл с параметрами) обозначается с помощью блока ЦИКЛ С ПАРАМЕТРАМИ (блок ЦИКЛА) (рис. 5).

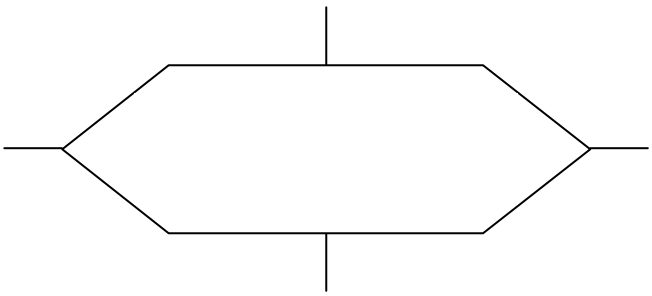

Рисунок 5 – Изображение блока ЦИКЛ С ПАРАМЕТРАМИ

Другие виды циклов не используют данный блок.

6) Если блок-схема алгоритма вычислительного процесса занимает много места и требуется переносить часть блок-схемы на другой лист (или в другое место на том же листе), то используется блок с названием СОЕДИНИТЕЛЬ (рис. 6). Внутри блока указывается номер того блока, к которому (от которого) ведет разорванная линия потока.

Рисунок 6 – Изображение блока СОЕДИНИТЕЛЬ

3.2 Блок-схемы базовых алгоритмических структур

Практически все блок-схемы алгоритмов состоят из базовых составляющих, которые комбинируются между собой – блок действия одной алгоритмической структуры заменяется другой структурой. Базовые структуры бывают следующих видов.

1) Линейный алгоритм.

Линейным называется вычислительный процесс, в котором предусматривается получение результата путем однократного выполнения последовательности действий при любых значениях исходных данных. Характерной особенностью линейного вычислительного процесса является то, что направление вычислений не зависит от исходных данных и промежуточных результатов.

Алгоритм линейного вычислительного процесса графически может быть представлен в виде блок-схемы КОМПОЗИЦИЯ (СЛЕДОВАНИЕ) – объединение нескольких следующих друг за другом блоков ПРОЦЕСС (рис. 7).

Рисунок 7 – Блок-схема линейного алгоритма

Как правило, алгоритм линейного вычислительного процесса является основой для блок-схем, так как в "чистом виде" такие вычислительные процессы представляют собой алгоритмы простейших задач.

2) Разветвляющийся алгоритм.

Разветвляющимся вычислительным процессом называется процесс, направление вычислений в котором зависит от результата проверки некоторого условия.

Алгоритм разветвляющегося вычислительного процесса предусматривает выбор одной из нескольких возможных альтернатив (последовательностей действий) в зависимости от значения исходных данных или промежуточных результатов. Каждую из этих последовательностей называют ветвью алгоритма. Блок-схема алгоритма разветвляющегося вычислительного процесса содержит, по крайней мере, один блок УСЛОВИЕ.

Графической интерпретацией алгоритма разветвляющегося вычислительного процесса является блок-схема АЛЬТЕРНАТИВА (или ВЕТВЛЕНИЕ), в которой может быть предусмотрен полный (рис. 8) или неполный (рис. 9) выбор.

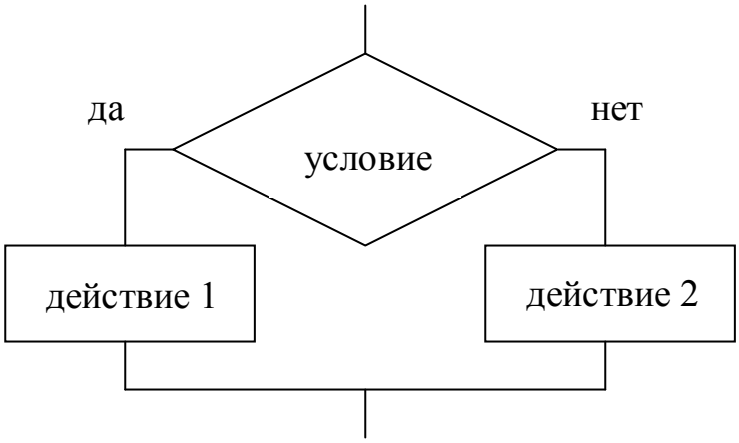

Рисунок 8 – Блок-схема АЛЬТЕРНАТИВА с полным выбором

В блок-схеме АЛЬТЕРНАТИВА с полным выбором в зависимости от результата проверки условия выполняется только действие ветви "да" или только действие ветви "нет". В блок-схеме АЛЬТЕРНАТИВА с неполным выбором в зависимости от результата проверки условия либо выполняется действие ветви "да" (действие 1), либо оно пропускается.

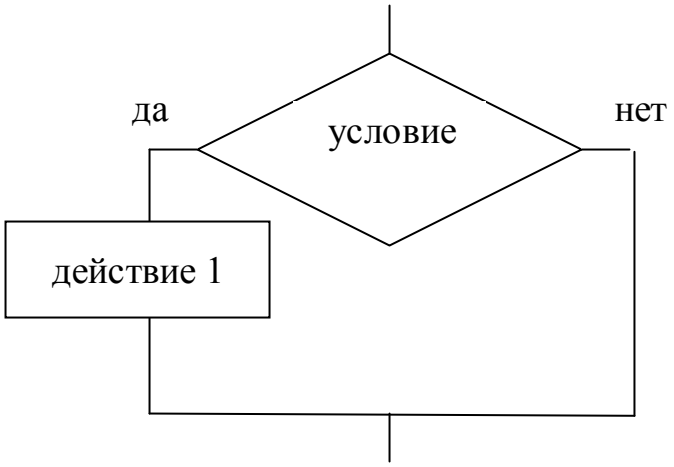

Рисунок 9 – Блок-схема АЛЬТЕРНАТИВА с неполным выбором

Разновидностью блок-схемы АЛЬТЕРНАТИВА является блок-схема ВЫБОР (рис. 10). В ней, в зависимости от нескольких условий, выполняется одно из предусмотренных действий.

Обычно в языке программирования присутствует оператор заменяющий собой несколько операторов проверки условия. Но такая структура все равно обозначается блок-схемой ВЫБОР.

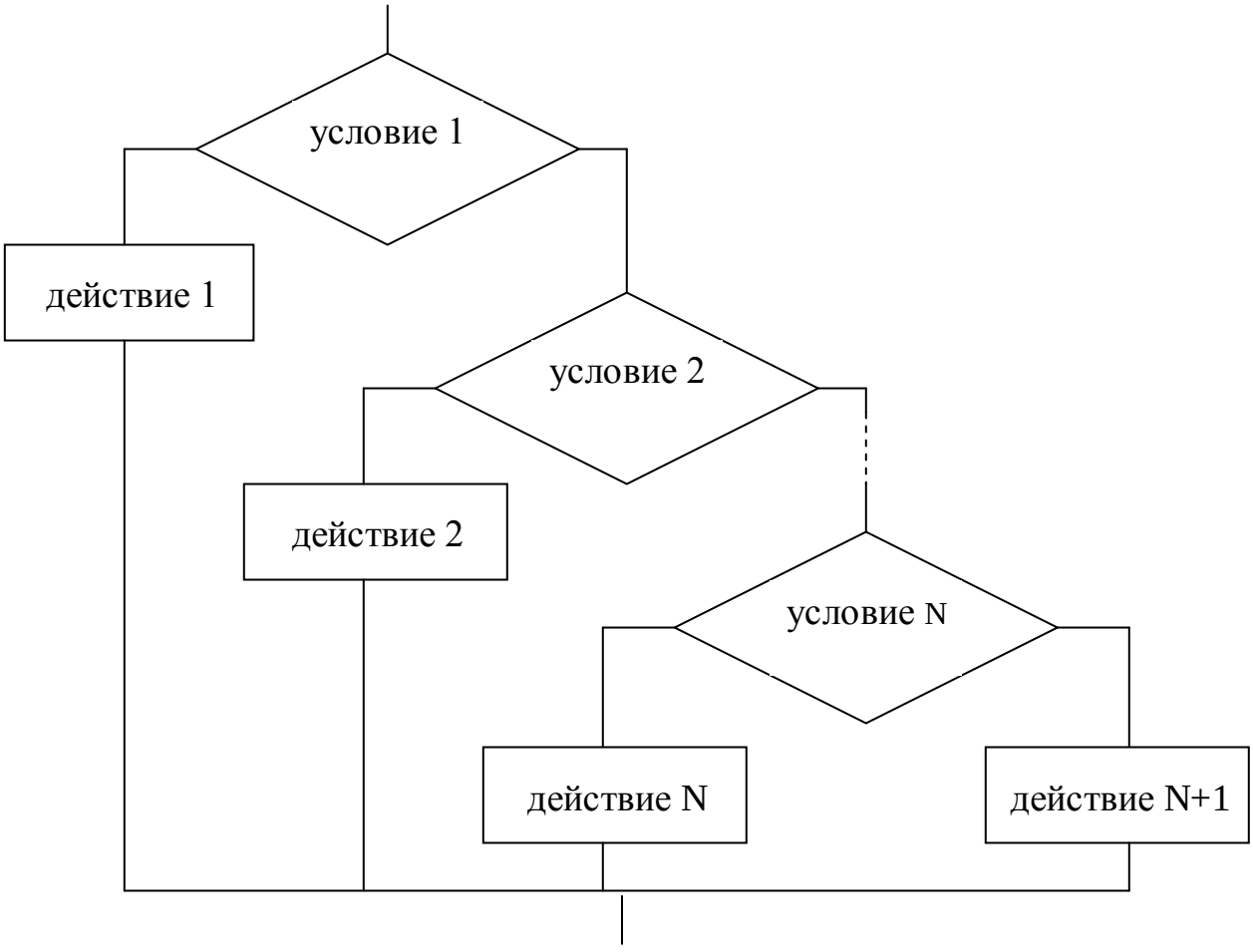

Рисунок 10 – Блок-схема ВЫБОР

3) Циклический алгоритм.

Циклическим называется вычислительный процесс, в котором получение результата обеспечивается путем многократного повторения некоторой последовательности действий.

Графической интерпретацией алгоритма циклического вычислительного процесса является блок-схема ИТЕРАЦИЯ (или ЦИКЛ). Различают несколько разновидностей блок-схем ЦИКЛ: цикл с параметрами, цикл с предусловием и цикл с постусловием.

Блок-схема цикла с параметрами представлена на рис. 11.

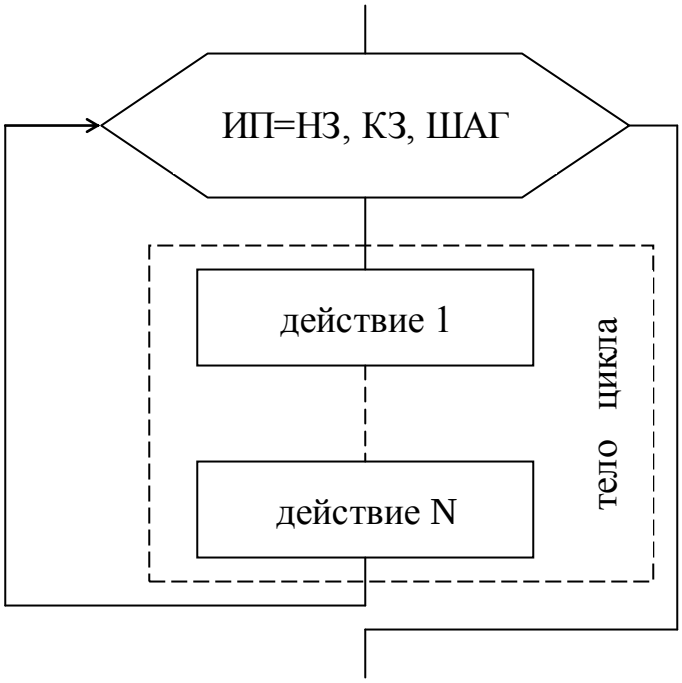

Рисунок 11 – Блок-схема ЦИКЛА с параметрами

На рисунке приняты следующие сокращения:

ИП – имя переменной, в которую заносится значение параметра;

НЗ – начальное значение параметра;

КЗ – конечное значение параметра;

ШАГ – величина приращения параметра после каждого выполнения тела цикла.

Тело цикла представляет собой любой вычислительный процесс и выполняется столько раз, сколько разных значений примет параметр в заданных пределах от начального значения до конечного значения с указанным шагом. Цикл с параметрами относится к циклу с явно заданным числом повторений (подходит для алгоритма, в котором число повторений известно заранее).

Цикл с предусловием и цикл с постусловием относятся к так называемым итерационным циклам. В таких циклических вычислительных процессах число повторений цикла заранее не известно. Выход из цикла осуществляется не после того, как цикл повторится заданное число раз, а при выполнении определенного условия, связанного с проверкой значения изменяющейся в теле цикла величины. Блок-схема цикла с предусловием представлена на рис. 12, а блок-схема цикла с постусловием – на рис. 13.

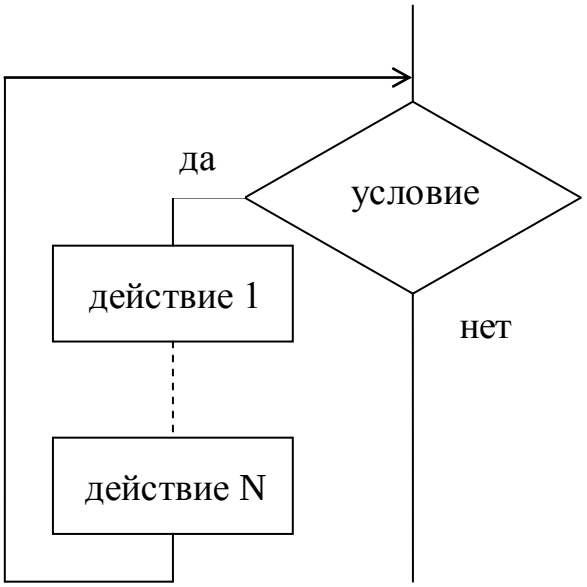

Рисунок 12 – ЦИКЛ с предусловием

Суть алгоритма цикла с предусловием можно изложить следующим образом: пока выполняется условие – повторять действия. В таких циклах возможны ситуации, когда тело цикла не выполняется ни разу (например, если при первой же проверке не выполняется условие, то сразу происходит выход из цикла).

В цикле с постусловием тело цикла выполняется не менее одного раза. При этом действия, предусмотренные в теле цикла, выполняются до тех пор, пока не выполнится заданное условие.

Рассмотренные блок-схемы циклов позволяют описать простые циклические вычислительные процессы. При решении сложных задач может возникнуть необходимость внутри одного цикла организовать дополнительно один или несколько других циклов. Такие циклы называются вложенными. При этом цикл, внутри которого создается другой цикл, называется внешним, а цикл, создаваемый внутри другого – внутренним. Правила организации как внешнего, так и внутреннего циклов те же, что и для простых циклов.

Составная блок-схема простой задачи изображена на рис. 14. В ней линейная структура включает в себя разветвление.

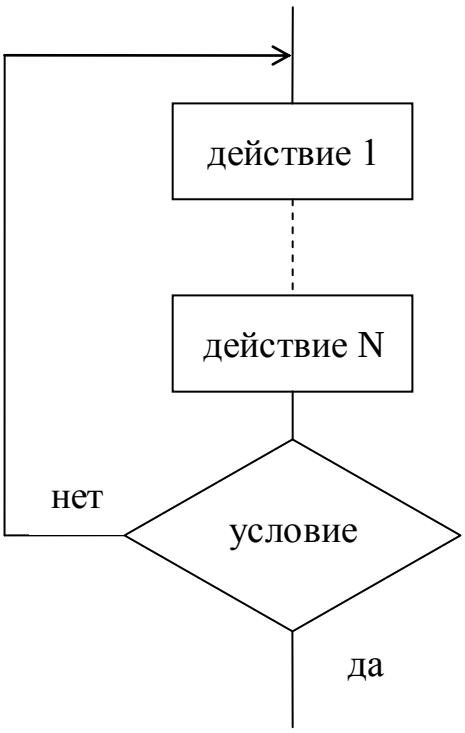

Рисунок 13 – ЦИКЛ с постусловием

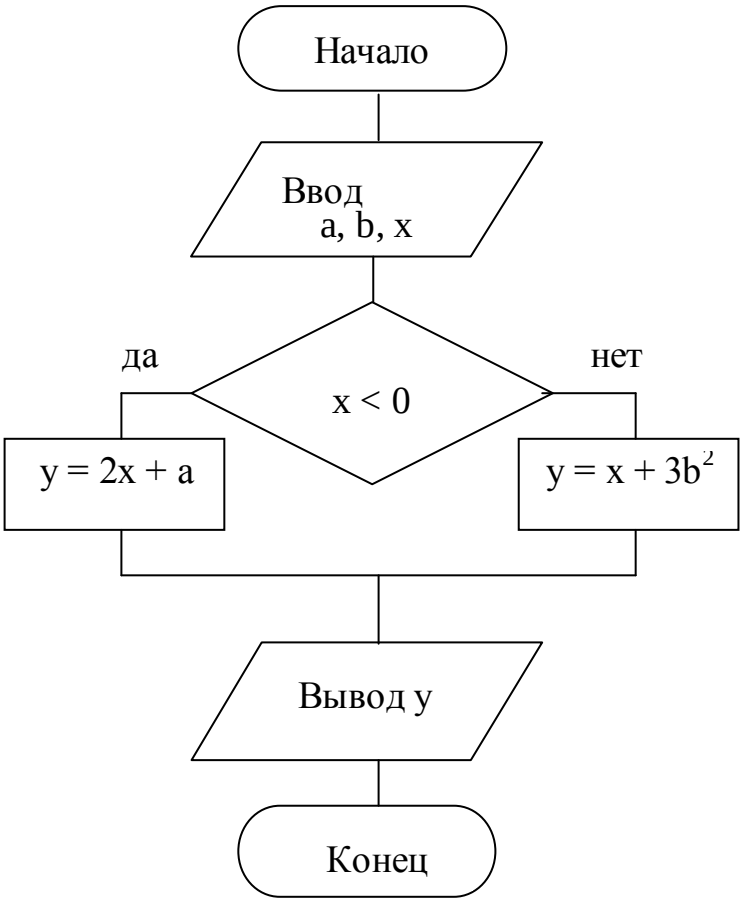

Рисунок 14 – Пример блок-схемы

4. Рекомендации по выполнению учебной практики

4.1 Технология программирования

В разработке программ традиционно выделяют следующие этапы: формализация постановки (перевод задачи на язык математической символики), выбор либо разработка методов решения, разработка алгоритма, кодирование (перевод алгоритма на алгоритмический язык в соответствии с его синтаксисом), тестирование, отладка, защита, сдача в эксплуатацию, сопровождение (авторский надзор). Формализация и алгоритмизация часто носят нестандартный характер и не имеют единых методов. Остановимся подробнее на отладке и тестировании.

Отладка — процесс выявления, локализации и устранения ошибок в алгоритме и реализующей его программе — осуществляется с помощью тестирования. Не будем останавливаться на выявлении синтаксических ошибок — это с разной эффективностью реализуется трансляторами; речь пойдет о программе, которая благополучно транслируется, принимает исходные данные и выдает (хоть какие-то) результаты. Задача — убедиться в корректности алгоритма, то есть получить уверенность, что программа выдает результаты, соответствующие задаче и исходным данным. Показано, что посредством испытаний либо теоретически невозможно доказать правильность программы; можно лишь спровоцировать появление и устранить в ней как можно больше ошибок.

Тест — совокупность исходных данных для программы вместе с ожидаемыми результатами (с учетом формы представления последних). Тесты разрабатываются до, а не вовремя или после разработки программы, дабы избежать провокационного влияния стереотипов алгоритма на тестирование. Готовится не один тест, а их совокупность — *набор тестов,* призванный охватить максимум ситуаций. Испытание программы проводится сразу на всем наборе с протоколированием и анализом результатов.

Набор тестов называется *полным,* если он позволяет активизировать все ветви алгоритма. Набор тестов назовем *не избыточным,* если удаление из него любого теста лишает его полноты. Таким образом, искусство тестирования сводится к разработке полного и не избыточного набора тестов, а технология — к испытанию программы на всем наборе после внесения в нее каждого исправления. Удачно подобранные тесты позволяют не только констатировать факт наличия ошибок, но и *локализовать* их, то есть найти место в программе, виновное в получении неверных результатов.

В наборе тестов выделяют три группы:

• «тепличные» — проверяющие программу при корректных, нормальных исходных данных самого простого вида;

• «экстремальные» — на границе области определения, в ситуациях, которые могут произойти и на которые нужно корректно реагировать;

• «запредельные» — за границей области определения — ситуации, бессмысленные с точки зрения постановки задачи, но которые могут произойти из-за ошибок пользователя или программ, поставляющих исходные данные для тестируемой программы.

Требование надежности программирования: принимать данные, если они корректны, и получать для них правильные результаты либо отвергать их как некорректные, по возможности с анализом некорректности.

Подготовка тестов может оказаться довольно трудоемкой работой; но в некоторых случаях этот процесс можно автоматизировать. Так, некоторые задачи некритичны к содержимому входных массивов; правильность результата можно проверить визуальным сопоставлением с исходными данными, каковы бы они ни были. Один из приемов — *рандомизация* — подготовка случайных исходных данных.

Практически во всех трансляторах реализован *датчик случайных чисел,* — как правило, стандартная функция с именем Random, возвращающая случайное число из интервала [0;1). С помощью простейшего датчика можно получать практически любые распределения. Так, если ξ равномерно распределено на [0;1), то выражение $x=(b-a)\xi+a$ дает действительное число, равномерно распределенное на отрезке *[a;b),* а конструкция на Паскале k:=Ord(p>Random) присваивает переменной k единицу с вероятностью *p* и нуль — с вероятностью (1 - *p).* Аналогично можно генерировать монотонные (возрастающие или убывающие) последовательности (например обращение в цикле: a[i]:=a[i-1]+Random позволяет получить случайный неубывающий массив), а также «замусоренные» массивы — частично искаженные случайным шумом.

Другой прием — программная подготовка тестов в виде массивов или файлов существенной размерности, обладающих заданными свойствами, с известным ожидаемым результатом основной программы. Для этого необходимо написать специальную программу — генератор тестов, но это менее трудоемко, чем подготовка тестов вручную.

Таким образом, защита разработки сводится к демонстрации работоспособности программы на совокупности тестов, а именно к совместной защите набора тестов, алгоритма и программы.

4.2 Эффективность алгоритмов и программ

Эффективность алгоритма определяется потребляемыми ресурсами компьютера, а именно *быстродействием* (по количеству выполняемых операций, с учетом трудоемкости каждой из них, то есть в конечном итоге времени решения задачи определенной размерности) и общим *объемом оперативной памяти,* выделяемой (запрашиваемой) под данные. Эти показатели порой противоречивы: повышение быстродействия может потребовать дополнительных расходов памяти, либо наоборот. Если можно улучшить один показатель без ущерба для другого, следует этого добиваться; при возникновении же дилеммы в современных условиях следует отдавать

предпочтение экономии памяти в ущерб производительности, так как тактовая частота компьютеров растет опережающими темпами в сравнении с объемом оперативной памяти.

Следует иметь в виду, что алгоритмы и программы отлаживаются и испытываются, как правило, на модельных примерах небольшой размерности, так что на современных компьютерах не удается почувствовать их трудоемкости. Однако всегда подразумевается использование разработанных программ на реальных задачах существенной размерности, где показатели эффективности становятся уже критичными. В то же время некоторые задачи, имеющие факториальную зависимость трудоемкости от размерности, в состоянии «подвесить» любой современный компьютер даже на маленьких задачах, так что проблема эффективности не снимается с ростом производительности компьютеров.

Приведем ряд мер, которые можно рекомендовать для повышения эффективности.

1. Не использовать рабочие массивы того же порядка размерности, что и обрабатываемый или создаваемый, если это возможно. При обработке двумерного массива допустимо выделение одномерного рабочего массива для временного хранения строки или столбца матрицы.

2. Выбирать, где это возможно, короткие типы данных: Byte вместо Word; String[20] вместо String и т. д.

3. Использовать поименованные константы вместо неоднократного повторения констант-«близнецов».

4. При обращении к процедурам (особенно рекурсивным) параметры передавать преимущественно по адресу, а не по значению. Переменные, используемые в процедурах как рабочие, объявлять локальными, а не глобальными.

5. Выбирать алгоритмы, эффективные по числу операций, оценив предварительно порядок зависимости трудоемкости от размерности (логарифмическая, линейная, полиномиальная, факториальная и т. д.).

6. Избегать вычислений в циклах выражений, не зависящих от параметра цикла (например, Sin(Pi/n), где n не меняется в цикле) имея в виду, что трансцендентные функции вычисляются трудоемким разложением в ряд.

7. Прекращать вычисления, когда результат достигнут, либо очевидно, что он не может быть достигнут за приемлемое время. Для этого использовать циклы типа while ... do или repeat ... until вместо for ... do. Не рекомендуется обращаться к средствам принудительного завершения типа GoTo, Halt, Break и др., так как они сильно снижают наглядность программы и часто приводят к неожиданным последствиям.

8. Выбирать, где это возможно, наименее трудоемкие операции. Так, например, выражение n div k вычисляется быстрее, чем Trunc(n/k), к тому же дает гарантированно точный результат; Ord(Odd(n)) более эффективно, чем n mod 2.

5. Типовые задачи и методы их решения

5.1 Линейные алгоритмы

Задачи этой темы сводятся к разработке и программированию простых линейных алгоритмов, поэтому в решении не стоит использовать операторы цикла и массивы.

Многие задачи даны в содержательной постановке, то есть в терминах той предметной области, которой они обязаны своим происхождением. Для решения требуется провести математическую и алгоритмическую постановки, то есть вывести (или найти в литературе) необходимые формулы и разработать алгоритм.

Во многих задачах подразумевается использование целочисленной арифметики: вычисление целого частного от деления и остатка: операции div и mod в Паскале и др., а также вычисления с процентами.

Пример. Правительство гарантирует, что инфляция в новом году составит *p%* в месяц. Какого роста цен за год можно ожидать?

Решение. Простое произведение 12p% не будет верным, нужно найти так называемые сложные проценты. Если за каждый месяц цены возрастут в 1+p/100 раз, то за год рост цен составит $(1+p/100)^{12}$ раз, или прирост в процентах:

$$s = \left[\left(1+\frac{p}{100}\right)^{12} - 1\right] \times 100\%$$

Программа на Паскале в этом случае может иметь следующий вид.

```
Program Super_Procent;
var a, p, s: Real;
Begin
Write ('Введите процент месячной инфляции ');
Readln (p);
a:= Exp(Ln(1+p/100)*12); {кратность роста цен}
{Так как в Паскале нет возведения в степень, используется тождество:
a^b = exp(ln(a)*b)}
s:=(a-1)*100;
Writeln ('Годовой рост цен составит ', a:10:2,' раз. или ' s:10:2,'
процентов');
end.
```

Для испытания программы можно взять тесты (p - процент месячной инфляции; s - результат - годовой прирост цен в процентах):

- p = 0%; s = 0%;
- p = 1%; s = 12,68%;
- p = 10%; s = 213,84%.

5.2 Циклические и итерационные алгоритмы

В задачах этой темы реализуется тот или иной циклический процесс, который выполняется либо за заранее известное число шагов, либо до достижения некоторого условия (итерационные алгоритмы). В последнем случае полезно подстраховаться от появления «вечного цикла», которое может возникнуть из-за разных ошибок в программе и алгоритме, из-за некорректных данных либо вследствие накопления погрешностей. Для этого (хотя бы на этапе отладки) достаточно поставить лимит числа шагов (с выдачей сообщения в случае его исчерпания).

В итерационных алгоритмах заданная погрешность используется для проверки модуля разности найденного приближенного и точного значений, однако если последнее неизвестно, допустимо оценивать разность между соседними итерациями либо, например, при решении уравнений, модуля разности левой и правой частей уравнения или при отыскании корня функции - модуля ее значения. Тонкости применения этих подходов оставим специальным дисциплинам — программисту часто приходится разрабатывать алгоритмы, не владея в полном объеме соответствующей теорией.

Не стоит выделять память в виде массивов для хранения промежуточных итераций - достаточно получения окончательного результата и (в итерационных алгоритмах) числа шагов, проделанных до достижения условия - для оценки скорости сходимости алгоритма. На этапе отладки можно просто отображать промежуточные результаты (трассировать программу).

Часто в задачах для вычисления очередного слагаемого удобно рекуррентно использовать предыдущее слагаемое, а не организовывать дополнительный (внутренний) цикл.

Выражения для исследуемых функций, разумеется, не поддаются вводу; их полезно оформить в виде подпрограмм или просто отдельно выделенной прокомментированной строкой.

Пример. Проверить численно справедливость следующего разложения: $e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \ldots + \frac{x^n}{n!} + \cdots$ и оценить скорость сходимости, найдя число слагаемых, необходимое для достижения заданной погрешности ε.

Решение. Для заданного *x* вычислим левую часть, используя встроенную функцию Exp(x), и будем вычислять частичную сумму ряда

правой части $s_n = \sum_{i=0}^{n} \frac{x^i}{i!}$ до тех пор, пока она не будет отличаться от левой части менее чем на заданную погрешность ε.

Заметим, что для вычисления каждого слагаемого ряда $u_i = \frac{x^i}{i!}$ требуется возведение в степень (трудоемкая операция) и вычисление факториала (а это — дополнительный цикл). Однако нетрудно видеть, что каждое очередное слагаемое можно рекуррентно вычислить через предыдущее: $u_i = \frac{x}{i} u_{i-1}$, что требует всего двух операций.

Решение можно проиллюстрировать следующей программой:

```
Program Iteration;
var x, Eps, y, s, u: real;
    n: Integer;
const Limit = 100;          {ограничение на число шагов}
begin
Write ('Задайте аргумент и погрешность: ');
Readln (x, Eps);
y := Exp(x); {левая часть}
s := 1; {частичная сумма}
u := 1; {первое слагаемое}
n := 1; {число шагов}
repeat
u := x/n*u; {очередное слагаемое}
s :=s+u;
n := n+1;
until (Abs (y-s) <= Eps) or (n>=Limit);
if n >= Limit then
Wrileln (n, ' шагов не хватило для достижения точности')
else
begin
Writeln ('Левая часть: ', y:15:6);
Writeln ('Правая часть: ', s:15:6);
Writeln ('Погрешность ', Eps:10:6, ' достигнута за ',n, ' шагов');
end
end.
```

Замечание. Владение циклическими операторами позволяет в полной мере реализовать защиту от использования некорректных исходных данных. Напомним, что программа должна отвергать при вводе некорректные данные

и устойчиво работать при корректных. Пусть, например, по условию задачи $0<x\leq\pi$, кроме того, разумно потребовать $\varepsilon>0$. Тогда ввод можно организовать следующим образом:

```
var Okey: Boolean:
…
repeat {повторять, пока не будет правильного ввода}
Write ('Задайте аргумент от 0 до ',Pi, ' и положительную
погрешность ');
Readln (x, eps);
Okey := (x>0) and (x<=Pi) and (Eps>0);
if not Okey then
Writeln ('Неверные данные, повторите ввод');
until Okey;
```

Кроме того, после отладки итерационной программы полезно определить ее область применения, то есть найти, при каких *x* и ε процесс сходится за приемлемое время.

5.3 Элементарная машинная графика

В задачах этого раздела требуется построить предложенное статическое (неподвижное) изображение на экране дисплея. В необходимых случаях предусмотреть масштабирование для эффективного использования площади экрана. Графики функций строить в декартовой системе координат с обоснованным выбором масштаба изображения. Полезно вместо точечного изображения использовать кусочно-линейную аппроксимацию с достаточно мелким шагом.

При защите программ оценивается не только качество и надежность программы, но и качество постановки (конкретизация, детализация проработки) и дизайн полученного изображения.

Пример. Рассмотрим задачу: заполнение экрана полуокружностями заданного диаметра, напоминающими рисунок рыбьей чешуи. В приведенной ниже программе расчет количества полуокружностей по горизонтали и вертикали ведется в зависимости от разрешения дисплея - из требования мобильности программы.

```
Program Fish:
{Заполнить экран "рыбьей чешуёй"}
uses Graph, Crt;
const a=180; b=270; c=360;
{углы для рисования дуг}
var gd,gm,r,x,y,n,m,i,j,s,q,mx,my: Integer;
okey: Boolean;
begin
```

```
ClrScr;
Writeln('Введите размер (радиус) чешуи от 5 до 100:');
repeat
Readln(r);
okey:=(r>=5) and (r<=100);
if not okey then
Writeln ('Повторите, пожалуйста!');
until okey;
gd:=Detect;
InitGraph(gd, gm, 'c:\bp\bgi\');
SetColor(6);
mx:=GetMaxX;
my:=GetMaxY; {разрешение экрана}
n:=mx div (2*r);
{количество "чешуек", входящих на экран по горизонтали}
m:=my div r;
{количество "чешуек", входящих на экран по вертикали}
s:=(|mx-n*2*r) div 2:
{отступ от края по горизонтали}
x:=s:
q:=(my-m*r) div 2;
{отступ от края по вертикали}
y:=q;
RecTangle(x,y,mx-x,my-y); {рамка}
my:=-1; {индикатор чётности номера строки}
for i:=1 to m do
begin
my:=-my;
for j:=1 to n do
begin
Arc(x+my*r,y,b,c,r);
Arc(x+r,y,a,b,r);
x:=x+2*r
end;
x:=r*(i mod 2)+s;
y:=y+r:
end;
Readln;
Closegraph;
end.
```

Тестирование сводится к испытанию программы при различных (допустимых) размерах элементарных чешуек.

5.4 Простейшие операции над массивами

При работе с массивами рекомендуется опробовать механизм использования массивов переменной размерности. Если (например, в Паскале) при объявлении массивов размерность задается константами, следует объявлять максимально разумные размерности, а затем вводить переменные размерности и циклы организовывать уже по ним.

Обращаем внимание на аккуратность использования индексов: общепринято в матрицах, например, первым индексом обозначать номер строки, а вторым - номер столбца (или количество соответственно).

При обработке массивов (вводимых с клавиатуры или генерируемых случайно) рекомендуется использовать «эхопечать» - вывод на дисплей всего введенного массива в наглядной форме, с рациональным использованием площади экрана - для визуального контроля правильности ввода и демонстрации соответствия результатов введенным данным.

Следует максимально ограничивать себя в выделении дополнительных (рабочих) массивов размерности того же порядка, что и обрабатываемые, а также искать эффективные по трудоемкости алгоритмы: во многих задачах возможно решение за один проход (просмотр) массива, хотя провокационно напрашивается многопроходный алгоритм. Кроме того, в «пассивных» задачах, посвященных анализу или поиску в заданном массиве, следует воздерживаться от искажений массива в своих целях.

Пример. Реализуем на массивах выполнение строевых команд «ряды сдвой» и «сомкнись». То есть из массива *A(2n)* элементы с четными индексами нужно перенести в начало массива *B(n),* а оставшиеся — сдвинуть к началу массива *A.*

Пересылка из *A* в *B* очевидна и выполняется в одном цикле с пересчетом индексов. Для удаления образовавшихся «пустот» в *A* возможны следующие подходы:

- передвинуть все элементы, начиная с 3-го и до конца, на одну позицию, затем начиная с 5-го — еще на одну позицию, и так далее — за несколько проходов исходного массива;
- передвигать каждый элемент массива *A* на новое место, вычисляя каждый раз, на сколько позиций он сдвинется — при этом требуется только один проход;
- совместить пересылку в *B* со сдвигом в *A* — все в одном цикле.

Очевидно, наиболее эффективен третий прием, он и реализован в предлагаемой программе.

```
Program Move;
var A: Array [1..20] of Integer;
B: Array [1..10] of Integer:
k, n, i: Byte;
begin
Write ('Сколько всего чисел? (четное число)');
Readln (n);
```

```
For i:=1 to n do
begin
Write (i, '-й элемент ');
Readln (A[i]);
end;
{эхопечать введенного массива}
Writeln ('Заданный массив' :45);
for i:=1 to n do Write (A[i]:4);
Writeln;
k:=n div 2;
for i:=1 to k-1 do
begin
B[i]:=A[2*i]; {пересылка}
A[i+1]:=A[2*i+1]
{сдвиг очередного из оставшихся}
end;
B[k]:=A[2*k];
{последняя пересылка больше нет сдвигов}
{распечатка результатов}
Writeln ('Результат: массив A':25);
for 1:=1 to k do Write (A[i]:4);
Writeln;
Writeln ('массив B':25);
for i:=1 to k do Write (B[i]:4);
Writeln
end.
```

Испытывать эту программу можно на любом массиве (размерностью до 20) любых разнородных чисел (желательно, не более чем 3-разрядных - для удобства восприятия результатов).

Рекомендуемая литература

1. Тарасов А.Д. Основы программирования на языке Паскаль: Учебное пособие. – Оренбург: Издательский центр ОГАУ, 2006

2. Попов В.Б. Turbo Pascal для школьников: Учеб. пособие. – 3-е доп. изд. – М.: Финансы и статистика, 2004. – 528 с.: ил.

3. Фараонов В.В. Turbo Pascal: Учебное пособие. – СПб.: Питер, 2009. – 367 с.: ил.

4. Фараонов В.В. Turbo Pascal 7.0. Практика программирования: учебное пособие / В.В. Фараонов. – М.: КНОРУС, 2009. – 416 с.

Приложение 1

ФГБОУ ВО «ОРЕНБУРГСКИЙ ГОСУДАРСТВЕННЫЙ АГРАРНЫЙ УНИВЕРСИТЕТ»

Кафедра Техносферной и информационной безопасности

УЧЕБНАЯ ПРАКТИКА ПО ПРОГРАММИРОВАНИЮ

Выполнил:
студент Иванов И. И.
21 группа КОИБАС

Научный руководитель:
Сидоров С. С.

Оренбург – 2015

Андрей Дмитриевич Тарасов
Александр Геннадьевич Матвеев

МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ К УЧЕБНОЙ ПРАКТИКЕ ПО ПРОГРАММИРОВАНИЮ

Для направлений подготовки:
090900.62 – Информационная безопасность (профиль –информационная безопасность автоматизированных систем);
090303.65 - информационная безопасность автоматизированных систем
Форма обучения: очная

Издательский центр ОГАУ
460795, г. Оренбург, ул. Челюскинцев, 18.
Тел.: (3532) 77-61-43