

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Методические указания
по выполнению практических работ
по дисциплине
«ПАРАДИГМЫ ПРОГРАММИРОВАНИЯ»
для студентов направления подготовки
02.04.01 Математика и компьютерные науки
профиль «Искусственный интеллект в кибербезопасности»

Ставрополь
2026

Содержание

Введение	3
Практическое занятие 1 и 2	5
Практическое занятие 3	11
Практическое занятие 4 и 5	13
Практическое занятие 6 и 7	20
Практическое занятие 8 и 9	36
Практическое занятие 10 и 11	54
Практическое занятие 12 и 13	58
Практическое занятие 14	65
Практическое занятие 15 и 16	70
Практическое занятие 17 и 18	74
Практическое занятие 19 и 20	87
Практическое занятие 21 и 22	93
Практическое занятие 23 и 24	107
Практическое занятие 25 - 27	123

Введение

Целью дисциплины: формирование набора универсальных и общепрофессиональных компетенций будущего магистра по соответствующему направлению подготовки.

Задачи дисциплины:

- формирование у студентов теоретических знаний и практических навыков по парадигмам программирования;
- практическое освоение методов реализации программных решений в различных парадигмах;
- освоение синтаксиса и концепций языков программирования, соответствующих различным парадигмам
- приобретение навыков проектирования программных систем с учетом выбранной парадигмы.

Перечень планируемых результатов обучения по дисциплине (модулю), соотнесённых с планируемыми результатами освоения образовательной программы

Код, формулировка компетенции	Код, формулировка индикатора	Планируемые результаты обучения по дисциплине, характеризующие этапы формирования компетенций, индикаторов
ПК-1 Способен демонстрировать базовые знания математических и естественных наук, основ программирования и информационных технологий	ИД-1 ПК-1 в профессиональной деятельности показывает фундаментальные знания математики и естественных наук ИД-2 ПК-1 решает проблемы профессиональной деятельности используя средства программирования и информационных технологий	Знает основные парадигмы программирования, их концептуальные основы, отличия и особенности. Знает как выбранная парадигма влияет на структуру и стиль написания программ. Знает основные языки программирования, реализующие различные парадигмы, и их синтаксис. Умеет анализировать задачи и выбирать наиболее подходящую парадигму для их решения. Умеет реализовывать программы с использованием различных парадигм, применяя

		соответствующие концепции и конструкции. Умеет оценивать преимущества и ограничения различных подходов при проектировании программных решений. Владеет навыками создавать программные решения в разных парадигмах на практике; анализировать и сравнивать реализованные программы с точки зрения эффективности, читаемости и удобства сопровождения обосновывать выбор конкретной парадигмы при решении конкретных задач.
--	--	--

Данные методические указания к практическим разработаны в соответствии с рабочей программой по дисциплине «Парадигмы программирования» для студентов направления подготовки 02.04.01 Математика и компьютерные науки профиль «Искусственный интеллект в кибербезопасности».

Практическое занятие 1 и 2

Тема: «Жизненный цикл программного продукта».

Цель работы

Изучить процессы и стадии жизненного цикла программного продукта (ЖЦПП), а также модели разработки программного обеспечения (ПО).

Методические указания

Стандарты ЖЦПП.

Документация:

ГОСТ 34.601-90 - разработка и испытания автоматизированных систем (АС), в частности ПО, регламентированы

ГОСТ 34.602--89 - стадии создания АС

ГОСТ 34.603--92 - ТЗ на создание АС.

ГОСТ 19.xxx - Единая система программной документации.

ГОСТ 34. 201-89. ИТ. Виды, комплектность и обозначение документов при создании автоматизированных систем.

РД 50-34. 698-90. Методические указания. Информационная технология.

Автоматизированные системы. Требования к содержанию документов.

Регламентация процессов предприятия:

ГОСТ Р ИСО МЭК 12207-99 - Информационные технологии. Процессы жизненного цикла программного обеспечения

Качество программных средств:

ГОСТ 28806 - Качество программных средств. Термины и определения

ГОСТ 28195 - Оценка качества программных средств. Общие положения

ГОСТ 9126 - Информационная технология. Оценка программного продукта.

Характеристики качества и руководящие указания по их применению.

Зарубежные документы

DOD

Впервые формализованный и утвержденный стандарт жизненного цикла был утвержден в 1985 г. (уточнен в 1988 г.) для проектирования ПО систем военного назначения по заказам Министерства обороны США -- стандарт **DOD-STD-2167 A**. Для замены стандартов **DOD-STD-2167 A, 7935 A, 1703** Министерством обороны США в декабре 1994 г. утвержден Военный стандарт **MIL-STD-498** - Разработка и документирование программного обеспечения.

DOD 5200.28-STD - разработка и утверждение ТЗ

ISO

Наиболее полно и подробно ЖЦ, технология создания и обеспечения качества сложного ПО отражены в двух представленных ниже стандартах ISO:

Стандарт **ISO 12207:1995** - Процессы жизненного цикла программных средств и **ISO 9126** - формирование характеристик качества ПО.

ISO 15271: 1998. (ГОСТ Р-2002). ИТ. Руководство по применению ISO 12207.

ISO 16326: 1999. (ГОСТ Р-2002). ИТ. Руководство по применению ISO 12207 при административном управлении проектами.

ISO 15504 -1- 9: 1998. ТО. Оценка и аттестация зрелости процессов жизненного цикла программных средств. Ч. 1. Основные понятия и вводное руководство. Ч. 2. Эталонная модель процессов и их зрелости. Ч. 3. Проведение аттестации. Ч. 4. Руководство по проведению аттестации. Ч. 5. Модель аттестации и руководство по показателям. Ч. 6. Руководство по компетентности аттестаторов. Ч. 7. Руководство по применению при усовершенствовании процессов. Ч. 8. Руководство по применению при определении зрелости процессов поставщика. Ч. 9. Словарь.

ISO 15846: 1998. ТО. Процессы жизненного цикла программных средств.

Конфигурационное управление программными средствами.

SPICE - Software Process Assessment (ISO/IEC 15504)

Оценка процессов разработки ПО. Данный стандарт предназначен для оценки процессов, так или иначе связанных с разработкой ПО. Стандарт содержит рекомендации по оценке (rating) процессов и представлению результатов оценки, то есть позволяет "измерять" их качество, для чего используются, в том числе, различные атрибуты процессов. На основании оценки вырабатываются рекомендации по улучшению.

Эскизное рабочее и техническое проектирование - **ISO/IEC DTR 7498-2-89, ISO/IEC DTR 10181-1, ISO/IEC DTR 11586-1, ISO/IEC DTR 10745**

Разработка ПО - **ISO/IEC 9798-91, ISO/IEC 11574-94, ISO/IEC DTR 10736, ISO/IEC CD 13888 M30Sec**

Эксплуатация ПО - **ISO/IEC DTR 13335-1,2,3**

В части регламентации процессов предприятия:

ИСО/ТО 10006:1997 (R) “Менеджмент качества. Руководство качеством при административном управлении проектами”;

ISO 15846, ISO 10007 - стандарты по менеджменту конфигурации программных средств;

ISO 9000 – 2000; группы ГОСТ Р 9000х;

ISO/IEC TR 15504 – Оценка процессов жизненного цикла ПО (Information technology – Software process assessment).

Перечень основных стандартов в области обеспечения качества программных средств:

ISO 9000 -3: 1997. Стандарты в области административного управления качеством и обеспечения качества. Часть 3. Руководящие положения по применению стандарта ISO 9001 при разработке, поставке и обслуживании программного обеспечения.

ISO 9000: 2000. (ГОСТ Р-2001). Система менеджмента (административного управления) качества. Основы и словарь.

ISO 9001: 2000. (ГОСТ Р-2001). Система менеджмента (административного управления) качества. Требования.

1809004: 2000. (ГОСТ Р-2001). Система менеджмента (административного управления) качества. Руководство по улучшению деятельности.

ISO 10005: 1995 – Административное управление качеством. Руководящие указания по программам качества.

ISO 10006: 1997 – Руководство по качеству при управлении проектом.

ISO 10007: 1995 - Административное управление качеством. Руководящие указания при управлении конфигурацией.

ISO 10013: 1995 – Руководящие указания по разработке руководств по качеству.

ISO 10011-1-3: 1990. Руководящие положения по проверке систем качества. Ч. 1. Проверка. Ч. 2. Квалификационные критерии для инспекторов-аудиторов систем качества. Ч. 3. Управление программами проверок.

ISO 9126: 1991. (ГОСТ-1993). ИТ. Оценка программного продукта. Характеристики качества и руководство по их применению.

ISO 14598 -1- 6: 1998 -2000. Оценивание программного продукта. Ч. 1. Общий обзор. Ч. 2. Планирование и управление. Ч. 3. Процессы для разработчиков. Ч. 4. Процессы для покупателей. Ч. 5. Процессы для оценщиков. Ч. 6. Документирование и оценивание модулей.

ISO 9126-1-4. (проекты). ИТ. Качество программных средств: Ч. 1. Модель качества. Ч. 2. Внешние метрики. Ч. 3. Внутренние метрики. Ч. 4. Метрики качества в использовании.

ISO 14756: 1999. ИТ. Измерение и оценивание производительности программных средств компьютерных вычислительных систем.

ISO 12119: 1994. (ГОСТ Р-2000). ИТ. Требования к качеству и тестирование.

Тестирование, сопровождение и интерфейсы:

ISO 13210: 1994. ИТ. Методы тестирования для измерения соответствия стандартам POSIX.

ISO 9945-1: 1990 (IEEE 1003. 1). ИТ. Интерфейсы переносимых операционных систем. Ч. 1. Интерфейсы систем прикладных программ (язык Си).

ISO 9945-2: 1992 (IEEE 1003. 2). ИТ. Интерфейсы переносимых операционных систем. Часть 2. Команды управления и сервисные программы.

ISO 14764: 1999. (ГОСТ Р-2002). ИТ. Сопровождение программных средств.

Безопасность

ISO 15408 -1 - 3: 1999. (ГОСТ Р -2002). Методы и средства обеспечения безопасности. Критерии оценки безопасности информационных технологий. Ч. 1. Введение и общая модель. Ч. 2. Защита функциональных требований. Ч. 3. Защита требований к качеству.

ISO 13335 - 1 - 5: 1996 -1998. ИТ. ТО. Руководство по управлению безопасностью. Ч. 1. Концепция и модели обеспечения безопасности информационных технологий. Ч. 2. Планирование и управление безопасностью информационных технологий. Ч. 3. Техника управления безопасностью ИТ. Ч. 4. Селекция (выбор) средств обеспечения безопасности. Ч. 5. Безопасность внешних связей.

ISO 10181: 1 -7. ВОО. 1996 - 1998. Структура работ по безопасности в открытых системах. Ч. 1. Обзор. Ч. 2. Структура работ по аутентификации. Ч. 3. Структура работ по управлению доступом. Ч. 4. Структура работ по безотказности. Ч. 5. Структура работ по конфиденциальности. Ч. 6. Структура работ по обеспечению целостности. Ч. 7. Структура работ по проведению аудита на безопасность.

IEC 61508: 1 -6: 1998 - 2000. Функциональная безопасность электрических / электронных и программируемых электронных систем. Часть 3. Требования к программному обеспечению. Часть 6. Руководство по применению стандартов IEC 61508-2 и IEC 61508-3.

ISO 15910: 1999. (ГОСТ Р-2002) ИТ. Пользовательская документация программных средств.

ISO 6592: 1986. ОИ. Руководство по документации для вычислительных систем.

ISO 9294: 1990. (ГОСТ 1993 г.). ТО. ИТ. Руководство по управлению документированием программного обеспечения.

ISO 14102: 1995. ИТ. Оценка и выбор CASE-средств.

ISO 14471: 1999. ИТ. Руководство по адаптации CASE-средств.

IEEE

Процессы жизненного цикла для развития программного обеспечения. Охватывает полный жизненный цикл ПО, в котором выделяются шесть крупных базовых процессов. Эти процессы детализируются 16 частными процессами. В последних имеется еще более мелкая детализация в совокупности на 65 процессов-работ. Содержание каждого частного процесса начинается с описания общих его функций и задач и перечня действий -- работ при последующей детализации. Для каждого процесса в стандарте представлена входная и результирующая информация о его выполнении и краткое описание сущности процесса.

Регламентация процессов предприятия:

IEEE Std 610.12-1990. IEEE Standard Glossary of Software Engineering Terminology.

IEEE 1074 – Жизненный цикл разработки программных средств (Процессы жизненного цикла для развития программного обеспечения. Охватывает полный жизненный цикл ПО, в котором выделяются шесть крупных базовых процессов. Эти процессы детализируются 16 частными процессами. В последних имеется еще более мелкая детализация в совокупности на 65 процессов - работ. Содержание каждого частного процесса начинается с описания общих его функций и задач и перечня действий -- работ при последующей детализации. Для каждого процесса в стандарте представлена входная и результирующая информация о его выполнении и краткое описание сущности процесса.).

Порядок разработки и документирования ИС и ПО:

IEEE 1063-1987 - Standard for Software User Documentation.

IEEE 830-1994 - Рекомендуемая практика формирования спецификаций программного обеспечения.

IEEE 829 – Планирование тестирования программных средств.

ANSI/IEEE 1008-1986. Тестирование программных модулей и компонентов ПС.

ANSI/IEEE 1012-1986. Планирование верификации и подтверждения достоверности качества (валидации) программных средств.

Задания для выполнения

1. Внесите в таблицу данные о категориях процессов ЖЦПП в соответствии с ГОСТ Р ИСО/МЭК 12207—2010.

	1.	
	2.	
	1.	
	2.	
	...	...

	5.	
	1.	
	2.	
	...	...
	7.	
	1.	
	2.	
	...	...
	11.	
	1.	
	2.	
	...	...
	7.	
	1.	
	2.	
	...	...
	8.	
	1.	
	2.	
	3.	

2. Внесите в таблицу данные по моделям разработки программного обеспечения.

Наименование модели	Характеристика	Достоинства	Недостатки
Модель кодирования и устранения ошибок (Code and fix)			

Каскадная модель или "водопад" (Waterfall)			
V-образная модель - разработка через тестирование (V- model)			
Инкрементная модель (Incremental Model)			
Итеративная или итерационная модель (Iterative Model)			
Спиральная модель (Spiral Model)			
Прототипная модель (Prototype Model)			

Практическое занятие 3

Тема: «Развитие парадигм программирования»

Цель работы

Познакомиться с основными парадигмами программирования.

Задания для выполнения

1. Разделиться на группы и подготовить доклад и презентацию на одну из тем:
 - 1) «Процедурное программирование»
 - 2) «Функциональное программирование»
 - 3) «Логическое программирование»
 - 4) «Объектно-ориентированное программирование»
2. Представить доклад и презентацию на занятии.

Методические указания

Время выступления 15 минут.

Доклад должен отражать

- историческую справку,
- современное состояние,
- перспективы дальнейшего развития,
- краткую характеристику языков программирования, соответствующих парадигме.

Презентация должна быть представлена в формате ppt, pptx или pdf.

Список рекомендованной литературы:

1. Городняя, Л. В. Парадигма программирования: курс лекций / Л. В. Городняя ; Новосиб. гос. ун-т. – Новосибирск : РИЦ НГУ, 2015. – 206 с.
2. Пратт Т. Языки программирования. Разработка и реализация. / Т. Пратт, М. Зелковиц; Под общей редакцией А.Матросова. – СПб.: Питер, 2002. 688 с.
3. Сошников Д. В. Функциональное программирование на языке F#. /Д. В. Сошников. – М.: ДМК Пресс, 2011.
4. Хендерсон П. Функциональное программирование. / П. Хендерсон. – М.: Мир, 1983. 349 с.
5. Бадд Т. Объектно-ориентированное программирование в действии. СПб.: Питер, 1997. 464 с.
6. Кауфман В. Ш. Языки программирования: концепции и принципы. М.: ДМК Пресс, 2010. 464 с.
7. Роганов Е.А. Основы информатики и программирования: Учебное пособие / Е.А. Роганов. — М.: МГИУ, 2001. — 315 с.
8. Себеста Р.У. Основные концепции языков программирования: 5-е издание / Себеста Р.У. — М.: Вильямс, 2001. — 672 с.
9. Что такое парадигмы программирования и зачем они нужны
<https://practicum.yandex.ru/blog/paradigmy-programmirovaniya/>

10. Что такое парадигмы программирования <https://sky.pro/media/cto-takoe-paradigmy-programmirovaniya/>

Практическое занятие 4 и 5

Тема: Языки низкого уровня. Машинные коды

Цель работы

Изучить основы машинной арифметики, представления чисел в прямом, обратном и дополнительном кодах и арифметических операций над ними.

Методические указания

Понятие и виды систем счисления

Под **системой счисления** (СС) понимается способ представления любого числа посредством алфавита символов, называемых цифрами.

По количеству символов, используемых для записи числа, системы счисления подразделяются на **позиционные** и **непозиционные**.

Если для записи числа используется бесконечное множество символов и значение цифры не зависит от ее положения в ряду цифр, изображающих число, то система счисления называется **непозиционной**.

Позиционные системы счисления для записи чисел используют ограниченный набор символов, называемых цифрами, и величина числа зависит не только от набора цифр, но и от того, в какой последовательности записаны цифры, т.е. от позиции, занимаемой цифрой. Количество цифр, используемых для записи числа, называется основанием системы счисления, в дальнейшем его обозначим **q** . В ЭВМ применяют позиционные СС с недесятичным основанием: двоичную, восьмеричную, шестнадцатеричную и др.

В двоичной СС используется две цифры: 0 и 1; восьмеричная СС имеет восемь цифр: 01234567, шестнадцатеричная – шестнадцать, причем первые 10 цифр совпадают по написанию с цифрами десятичной системы счисления, а для обозначения оставшихся шести цифр применяются большие латинские буквы, т.е. для шестнадцатеричной системы счисления получим набор цифр: 0123456789ABCDEF.

Число в позиционной системе счисления с основанием q может быть представлено в виде полинома по степеням q , а в общем виде это правило запишется так:

$$X_{(q)} = x_{n-1}q^{n-1} + x_{n-2}q^{n-2} + \dots + x_1q^1 + x_0q^0 + x_{-1}q^{-1} + x_{-2}q^{-2} + \dots + x_{-m}q^{-m},$$

где $X_{(q)}$ — запись числа в системе счисления с основанием q ;

x_i — натуральные числа меньше q , т.е. цифры;

n — число разрядов целой части;

m — число разрядов дробной части.

Перевод из десятичной системы счисления в двоичную, шестнадцатеричную и восьмеричную СС:

1. Исходное целое число делится на основание системы счисления, в которую переводится (на 2 - при переводе в двоичную СС, на 16 - при переводе в шестнадцатеричную СС, на 8 - в восьмеричную СС); получается частное и остаток,
2. Получаемые частные делить на основание новой СС до тех пор, пока частное не станет меньше основания системы счисления, в которую выполняется перевод.
3. Все полученные остатки и последнее частное преобразуются в соответствии с таблицей

перевода в цифры той системы счисления, в которую выполняется перевод;

4. Формируется результирующее число: младший разряд которого – первый остаток от деления, а старший – последнее частное.

Перевод из двоичной, шестнадцатеричной и восьмеричной систем счисления в десятичную: для перевода двоичного, восьмеричного, шестнадцатеричного числа в десятичную СС достаточно представить число в виде полинома, подставить в него известные коэффициенты и вычислить сумму.

Перевод из двоичной системы счисления в восьмеричную, шестнадцатеричную СС:

Исходное двоичное число разбивают на группы по три (четыре) цифры, начиная с младших разрядов (слева направо). При этом дополняются при необходимости нулями крайние левую и правую группы. Затем каждую группу из трех (четырех) цифр заменяют соответствующей восьмеричной (шестнадцатеричной) цифрой.

Если дробь: двигаясь от точки сначала влево, а затем вправо, разбивают двоичное число на группы по три (четыре) разряда, дополняя при необходимости нулями крайние левую и правую группы. Затем каждую группу из трех (четырех) разрядов заменяют соответствующей восьмеричной (шестнадцатеричной) цифрой.

Перевод из восьмеричной, шестнадцатеричной системы счисления в двоичную СС:

Для перевода восьмеричного (шестнадцатеричного) числа в двоичную СС достаточно заменить каждую цифру восьмеричного (шестнадцатеричного) числа соответствующим трехразрядным (четырёхразрядным) двоичным числом. Затем необходимо удалить крайние нули слева, а при наличии точки — и крайние нули справа. Для перевода правильной десятичной дроби в другую систему эту дробь надо последовательно умножать на основание той системы счисления, в которую она переводится. При этом умножаются только дробные части. Дробь в новой системе записывается в виде целых частей произведений, начиная с первого.

Конечной десятичной дроби в другой системе счисления может соответствовать бесконечная (иногда периодическая) дробь. В этом случае количество знаков в представлении дроби в новой системе берется в зависимости от требуемой точности.

Для перевода неправильной десятичной дроби в систему счисления с недесятичным основанием необходимо отдельно перевести целую часть и отдельно дробную.

Необходимо отметить, что целые числа остаются целыми, а правильные дроби - дробями в любой системе счисления.

При проектировании ЭВМ, создании инструментального и прикладного программного обеспечения разработчикам приходится решать вопрос о представлении в ЭВМ числовых данных. Для решения большинства прикладных задач обычно достаточно использовать целые и вещественные числа. Запись целочисленных данных в запоминающем устройстве ЭВМ не представляет затруднений: число переводится в двоичную систему и записывается в прямом коде. Диапазон представляемых чисел в этом случае ограничивается количеством выделенных для записи разрядов. Для вещественных данных обычно используются две формы записи: **число с фиксированной точкой (ЧФТ)** и **число с плавающей точкой (ЧПТ)**.

Большинство универсальных ЭВМ работает с числами, представленными с плавающей точкой, а большинство специализированных - с фиксированной точкой. Однако целый ряд машин работает с числами в этих двух форматах.

В общем виде способ представления чисел сильно влияет на характер программирования.

Так, программирование для ЭВМ, работающих в системе с фиксированной точкой, значительно усложняется, поскольку помимо алгоритмических трудностей этот процесс требует ещё отслеживания положения точки.

Числа с фиксированной точкой

Оговоримся, что разрядная сетка машины имеет постоянное число разрядов n .

Форма записи числа с фиксированной точкой использовалась в основном на ранних этапах развития вычислительной техники. Запись числа с фиксированной точкой обычно имеет знаковый и цифровой разряды. **Фиксированная точка** означает, что на этапе конструирования ЭВМ было определено, сколько и какие разряды машинного слова отведены под изображение целой и дробной частей числа. Запятая в разрядной сетке может быть зафиксирована, в принципе, после любого разряда.

Пример 1. Ячейка с целой и дробной частью.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	2^{n-1}					2^1	2^0	2^{-1}	2^{-2}						2^{-m}
Знак числа	Целая часть (n разрядов)							Дробная часть (m разрядов)							

Как частный случай числа с фиксированной точкой может быть рассмотрена запись целого числа (в этом случае все разряды, кроме знакового, используются для записи целой части).

Пример 2. Ячейка с записью целого числа.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	2^{n-1}														2^0
Знак числа	Цифровые разряды(n)														

К достоинствам использования чисел с фиксированной точкой относятся простота выполнения арифметических операций и высокая точность изображения чисел.

К недостаткам - небольшой диапазон представления чисел.

При представлении чисел с **фиксированной точкой** считают, что **точка** всегда находится перед старшим разрядом, а все числа, которые участвуют в вычислениях, считаются по абсолютной величине меньше единицы:

$$|X| < 1.$$

Введём две характеристики чисел: **диапазон изменения** и **точность представления**.

Диапазон **изменения** характеризуется теми пределами, в которых могут находиться числа, с которыми оперирует машина.

$$|X|_{max} = 0.111 \dots 1 \overset{n \text{ разрядов}}{\sim} = 1 - 2^{-n}.$$

Самое маленькое число, отличное от нуля:

$$|X|_{min} = 0.00 \dots 01 \underset{n \text{ разрядов}}{\sim} = 2^{-n}.$$

Таким образом, **диапазон** чисел, с которыми работает ЭВМ, есть:

$$|X|_{min} \leq |X| \leq |X|_{max}, \text{ то есть } 2^{-n} \leq |X| \leq 1 - 2^{-n}.$$

Иными словами, числа, которые выходят за **диапазон** изменения, в ЭВМ не могут быть представлены точно. Если

$$|X| < |X|_{min} = 2^{-n},$$

то такое число воспринимается как нуль. Если

$$|X| > |X|_{max} = 1 - 2^{-n},$$

то такое число воспринимается как бесконечно большое. Этим двум случаям соответствуют понятия *машинного нуля* и *машинной бесконечности*.

При оптимальном округлении **абсолютная ошибка** вычисляется по формуле:

$$|\Delta X| \leq 0.5 \cdot 2^{-n}.$$

Минимальная относительная ошибка вычисляется так:

$$|\delta X|_{\min} = \frac{|\Delta X|}{|X|_{\max}} = \frac{0.5 \cdot 2^{-n}}{1 - 2^{-n}} \approx 2^{-(n+1)},$$

так как при достаточно большом n $1 - 2^{-n} \approx 1$.

Максимальная относительная ошибка вычисляется по формуле:

$$|\delta X|_{\max} = \frac{|\Delta X|}{|X|_{\min}} = \frac{0.5 \cdot 2^{-n}}{2^{-n}} = 0.5,$$

Ошибка представления числа зависит от величины самого числа и способа округления и находится в диапазоне:

$$2^{-(n+1)} \leq |\delta X| \leq 0.5.$$

Заметим, что для малых чисел ошибка может достигать большой величины.

Числа с плавающей точкой

В ЭВМ с *плавающей точкой* число представляется в виде:

$$X = \pm M_x \cdot q^{\pm p},$$

где M_x - *мантисса* числа, q - основание системы счисления, p - *порядок*.

Положение точки определяется значением порядка p . С изменением порядка точка перемещается (плавает) влево или вправо.

Пример 3. $125_{10} = 12.5 \cdot 10^1 = 1.25 \cdot 10^2 = 0.125 \cdot 10^3 = 0.0125 \cdot 10^4 = \dots$

Для установления однозначности при записи чисел принята **нормализованная форма** записи числа. Мантисса нормализованного числа может изменяться в диапазоне: $1/q \leq |m| < 1$. Таким образом, в нормализованных числах цифра после точки должна быть значащей.

Пример 4.

$$0.0832 \cdot 10^3 \text{ ненормализованное число} = 0.832 \cdot 10^2 \text{ нормализованное число}$$

Для представления чисел в машинном слове выделяют группы разрядов для изображения мантиссы, порядка, знака числа и знака порядка:

а) представление чисел в формате полуслова

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Знак m	Знак p	Порядок (4 разряда)				Мантисса (10 разрядов)									

б) представление чисел в формате слова

0	1	2	3	4	5	6	7	8	9	10	11	12	...	31
													...	
Знак m	Знак p	Порядок (7 разрядов)							Мантисса (23 разряда)					

Таким образом, числа с плавающей точкой позволяют увеличить диапазон обрабатываемых чисел, но при этом точность изображения чисел определяется только разрядами мантиссы и уменьшается по сравнению с числами с фиксированной точкой. При записи числа в

формате слова диапазон представимых чисел будет от $-1 \cdot 2^{127}$ до $1 \cdot 2^{127}$ ($2^{127} - 10^{38}$), а точность определяется мантиссой, состоящей из 23 разрядов. Точность может быть повышена путем увеличения количества разрядов мантиссы. Это реализуется путем представления чисел с так называемой двойной точностью (используется **формат двойного слова**).

Абсолютная ошибка представления числа в ЭВМ с *плавающей точкой* удовлетворяет неравенству

$$|\Delta X| \leq 0.5 \cdot 2^{-m}.$$

Так как $2^{-1} \leq |M_x| \leq 1 - 2^{-m}$, то **минимальная относительная ошибка**

$$|\Delta X|_{\min} = (0.5 \cdot 2^{-m}) / (1 - 2^{-m}) \gg 2^{-(m+1)},$$

при большом m , а **максимальная относительная ошибка** вычисляется по формуле:

$$|\Delta X|_{\max} = (0.5 \cdot 2^{-m}) / (2^{-1}) = 2^{-m}.$$

Видно, что относительная ошибка в ЭВМ с *плавающей точкой* не зависит от *порядка* числа.

При этом *точность* представления больших и малых чисел изменяется незначительно.

Теоретически *плавающая точка* имеет преимущества перед *фиксированной точкой*. Но соответствующее устройство получается намного сложнее. К тому же специфика выполнения операций с *плавающей точкой* требует большего числа микроопераций, что приводит к снижению быстродействия ЭВМ. Однако *плавающая точка* снимает с программиста обязанность отслеживать положение *точки* в вычислениях и значительно упрощает сам процесс программирования вычислительных задач.

Перевод чисел в двоично-десятичную систему

В вычислительной технике использование данной системы обусловлено возможностью применения элементов с двумя устойчивыми состояниями при сохранении преимуществ десятичной системы. Как правило, двоично-десятичная система реализуется в устройствах ввода-вывода числового материала. Она строится на сочетании принципов двоичной и десятичной систем счисления. В ней расположение десятичных разрядов сохраняется, цифра каждого десятичного разряда представляется **тетрадой**

Десятичные числа представляются с использованием либо **упакованного (PACK)**, либо **неупакованного (UNPACK)** формата (его еще называют **зонным форматом**).

В упакованном формате в каждом байте числа кодируются две цифры, в неупакованном – одна. Для кодирования десятичных цифр используется в основном естественный двоичный код, обычно называемый кодом «8421» (по весам разрядов двоичной тетрады).

Частным случаем неупакованного формата является **код ASCII** (American Standard Code for Interchange Information), используемый в ПК. В этом коде десятичная цифра представляется в младшей тетраде байта, а старшая тетрада принимает стандартное значение 0011. Упакованный формат обычно называют **BСD-форматом** (или BCD-числом – Binary Coded Decimal).

Задания для выполнения

1. Перевести следующие числа в десятичную систему счисления:
 - а) 110111_2 ;
 - б) 10110111.1011_2 ;
 - в) 563.44_8 ;
 - г) 721.35_8 ;
 - д) $1C4.A_{16}$;
 - е) $9A2F.B5_{16}$.

2. Перевести следующие числа из десятичной ПСС в ПСС с основаниями 2, 8, 16:
 - а) 463;
 - б) 1209;
 - в) 362;
 - г) 3925;
 - д) 11355.
3. Перевести следующие числа из десятичной ПСС в ПСС с основаниями 2, 8, 16 (точность вычислений - 5 знаков после точки):
 - а) 0.0625;
 - б) 0.345;
 - в) 0.225;
 - г) 0.725;
 - д) 217.375;
 - е) 31.2375;
 - ж) 725.03125;
 - з) 8846.04.
4. Перевести следующие числа в двоичную систему счисления, используя триады и тетрады соответственно:
 - а) 1725.326₈;
 - б) 341.34₈;
 - в) 7BF.52A₁₆;
 - г) 3D2.C₁₆.
5. Перевести следующие числа из одной ПСС в другую, используя триады и тетрады соответственно:
 - а) $11011001.01011_2 \rightarrow x_8$;
 - б) $1011110.1101_2 \rightarrow x_8$;
 - в) $1101111101.0101101_2 \rightarrow x_{16}$;
 - г) $110101000.100101_2 \rightarrow x_{16}$.
6. Считая число u_p отрицательным, представить его в двоичном коде с плавающей точкой, после чего нормализовать это число (2 знаковых разряда – один разряд для знака мантиссы, один – для знака порядка, 7 разрядов – для представления порядка и 23 разряда – для мантиссы).
7. Представить отрицательное число u_p в прямом, обратном и дополнительном кодах.
 - а) $u_5 = 2431.0345$, $q = 11$;
 - б) $u_{12} = A49.72412$; $q = 7$;
 - в) $u_6 = 4610.3426$, $q = 13$;
 - г) $u_{14} = 9B4.A0714$; $q = 9$.
8. Заданные числа u и v представить в двоично-десятичной форме:
 - в упакованном формате (BCD);
 - в неупакованном формате (ASCII).
 - а) $u = 1950$, $v = 0.135$;
 - б) $u = 226$, $v = 0.033$;
 - в) $u = 1875$, $v = 0.56$;
 - г) $u = 693$, $v = 0.792$.

Контрольные вопросы:

1. Что называется системой счисления?
2. На какие два типа можно разделить все системы счисления?
3. Какие системы счисления применяются в вычислительной технике: позиционные или непозиционные? Почему?
4. Что называется основанием системы счисления?
5. Охарактеризуйте двоичную, восьмеричную, шестнадцатеричную системы счисления: алфавит, основание системы счисления, запись числа.
6. По каким правилам выполняется сложение двух положительных целых чисел?
7. Каковы правила выполнения арифметических операций в двоичной системе счисления?
8. Для чего используется перевод чисел из одной системы счисления в другую?
9. Сформулируйте правила перевода чисел из системы счисления с основанием p в десятичную систему счисления и обратного перевода: из десятичной системы счисления в систему счисления с основанием p . Приведите примеры.
10. Как выполнить перевод чисел из двоичной СС в восьмеричную и обратный перевод? Из двоичной СС в шестнадцатеричную и обратно? Приведите примеры.
11. По каким правилам выполняется перевод из восьмеричной в шестнадцатеричную СС и наоборот? Приведите примеры.
12. Чему равны веса разрядов слева от точки, разделяющей целую и дробную части в двоичной системе счисления (восьмеричной, шестнадцатеричной)?
13. Чему равны веса разрядов справа от точки, разделяющей целую и дробную части, в двоичной системе счисления (восьмеричной, шестнадцатеричной)?

Литература

1. Япарова Ю.А. ЛАБОРАТОРНЫЙ ПРАКТИКУМ ПО ДИСЦИПЛИНЕ «АРХИТЕКТУРА АППАРАТНЫХ СРЕДСТВ». – Для студентов специальности 230111 «Компьютерные сети». – Калининград: АНО «Калининградский бизнес колледж», 2015.- 76 с.
2. Городня, Л. В. Парадигма программирования: курс лекций / Л. В. Городня ; Новосиб. гос. ун-т. – Новосибирск : РИЦ НГУ, 2015. – 206 с.

Практическое занятие 6 и 7

Тема: Языки низкого уровня. Программирование на ассемблере.

Цель работы. Получить основные навыки использования пакетов MASM и TASM при программировании на языке Ассемблера. Исследовать работу и научиться использовать команды пересылки, загрузки и операций с флагами. Познакомиться на практике с режимами адресации.

Методические указания

Основные инструментальные средства

Пакет MASM удобное средство для создания программ на языке Ассемблера для реального и защищённого режимов процессоров семейства X86.

Основные инструментальные средства, входящие в состав пакета:

- а) `rw` - интегрированная среда, включающая в себя редактор текстов программ и позволяющая компилировать модули и запускать программы не выходя из неё;
- б) `cv` - интегрированная среда предназначенная для отладки и дизассемблирования программ;
- в) `masm` - компилятор с языка Ассемблера;
- г) `link` - компоновщик объектных модулей в исполняемую программу;
- д) `ml` - компилятор и компоновщик в одном лице.

Компилятор `masm` рекомендуется запускать с ключом `/Zi`, а компоновщик - с ключом `/Co`; при этом в объектные модули будет записана отладочная информация.

Аналогичные опции рекомендуется установить и в среде `rw`.

Пакет TASM не имеет единой интегрированной среды, но по популярности не уступает пакету MASM.

Основные инструментальные средства, входящие в состав пакета:

- а) `tasm` - компилятор с языка Ассемблера;
- б) `tlink` - компоновщик объектных модулей в исполняемую программу;
- в) `td` - интегрированная среда, предназначенная для отладки и дизассемблирования программ.

Учитывая отсутствие интегрированной среды в пакете TASM, рекомендуется составить пакетные файлы для компиляции, компоновки и отладки, например так (предполагается, что пакет TASM установлен в каталог `C:\TASM`, исходный текст программы записан в файл `main.asm`, а соответствующие пакетные файлы находятся в том же каталоге, что и `main.asm`):

а) компиляция и компоновка (файл `COMPILE.BAT`):

```
@echo off
cls
c:\tasm\bin\tasm /zi main.asm
if errorlevel 1 goto ERROR
c:\tasm\bin\tlink /v main.obj
if errorlevel 1 goto ERROR
exit
:ERROR
```

```
echo ERROR!  
б) отладка (файл DEBUG.BAT):  
@echo off  
cls  
if not exist main.exe echo main.exe не существует!  
if not exist main.exe goto EXIT  
c:\tasm\bin\td main.exe  
:EXIT
```

Создание первого проекта в Microsoft Macro Assembler

Запустите PWB.exe (не забудьте сначала выполнить New-vars.bat из каталога с PWB.exe)

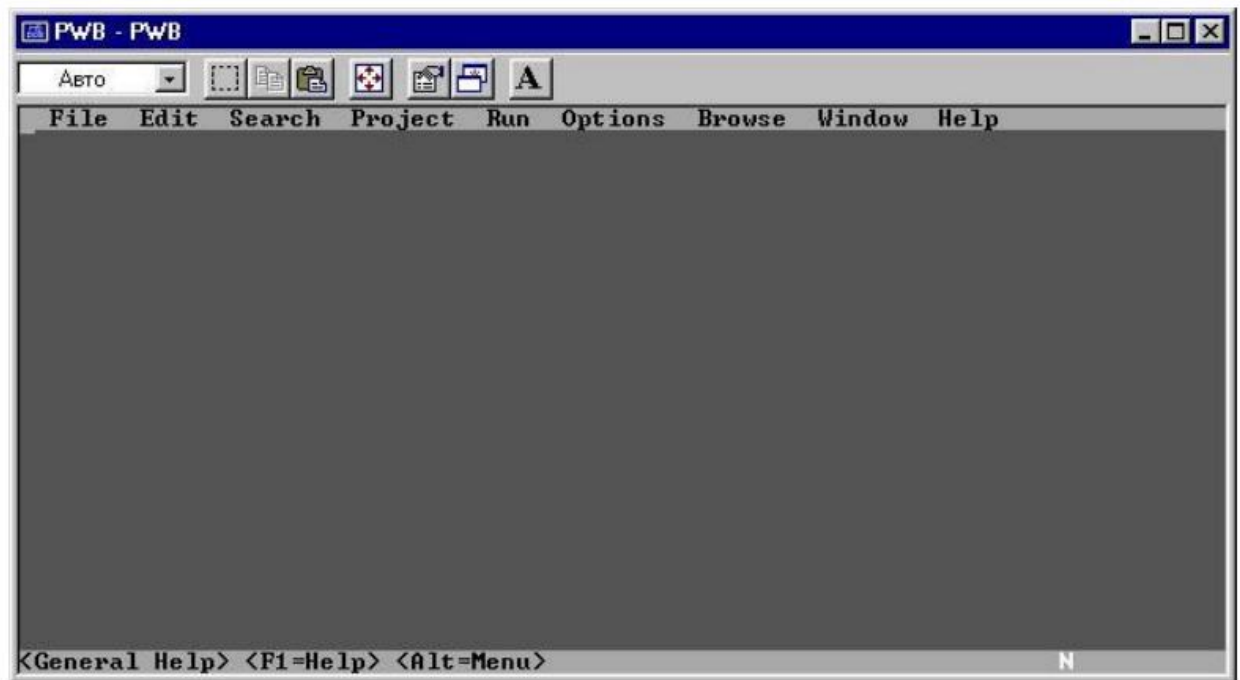


Рисунок 1 – Терминал

Выполните следующую команду меню:

Project -> New Project..., где введёте имя вашего проекта.

Например: Hello.



Рисунок 2 – Создание нового проекта

File -> New для создания в вашем проекте файла для исходного текста.

Введите код программы и выполните:

Project -> Compile File: Hello.asm;

Project -> Build: Hello.exe;

Run -> Execute: Hello.exe;

Run -> Debug: Hello.exe

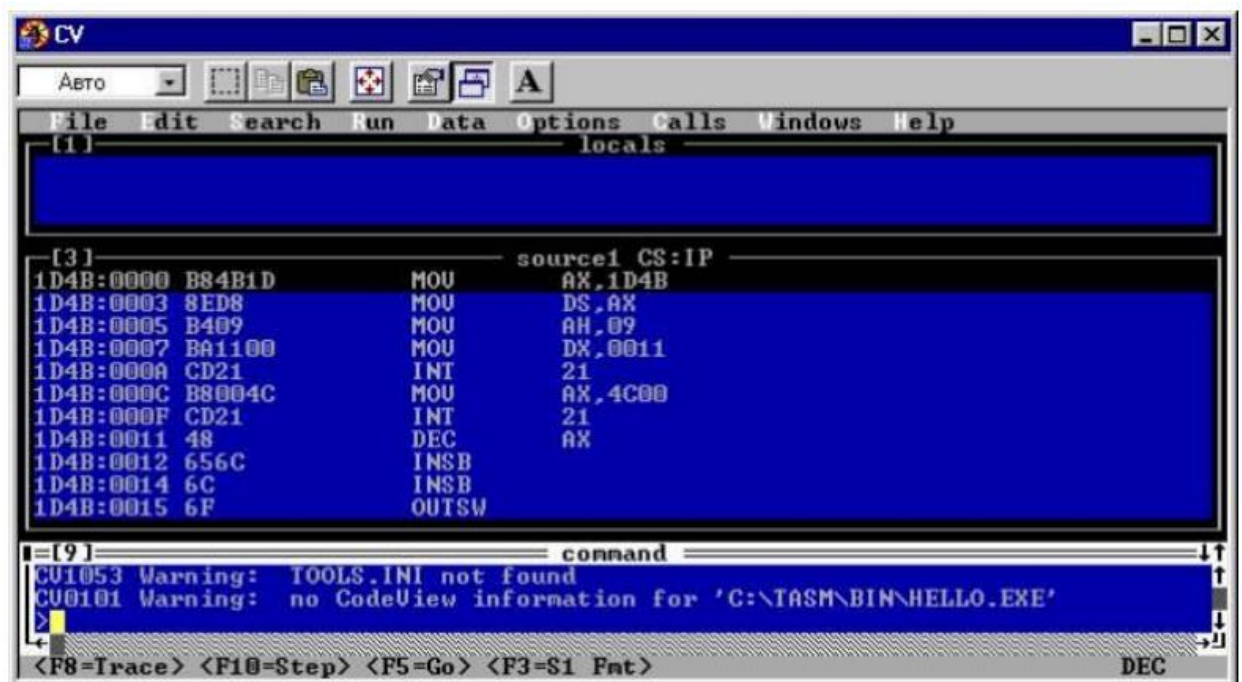


Рисунок 3 – Просмотр регистров

Для того чтобы увидеть регистры, воспользуйтесь командой меню

Windows -> Register или комбинацией "горячих" клавиш Alt+7.

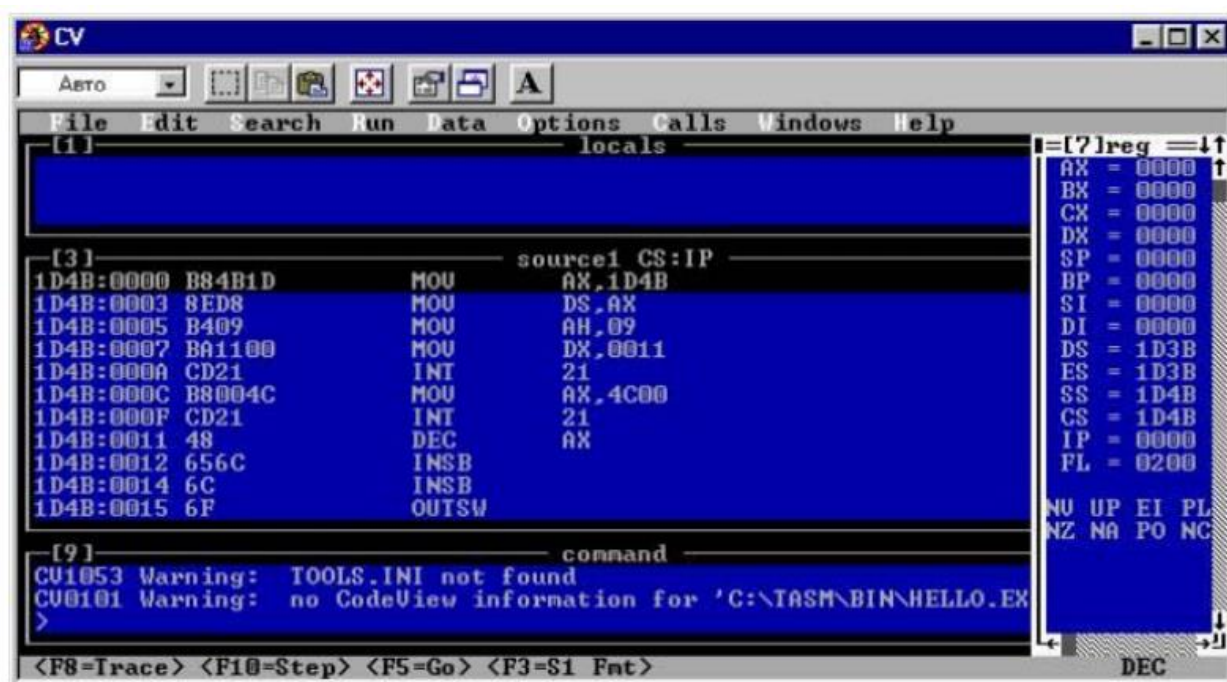


Рисунок 4 – Просмотр содержимого памяти

Для того чтобы увидеть содержимое памяти воспользуйтесь командой меню Windows -> Memory 1 или Windows -> Memory 2 или комбинацией "горячих" клавиш Alt+5 или Alt+6 соответственно.

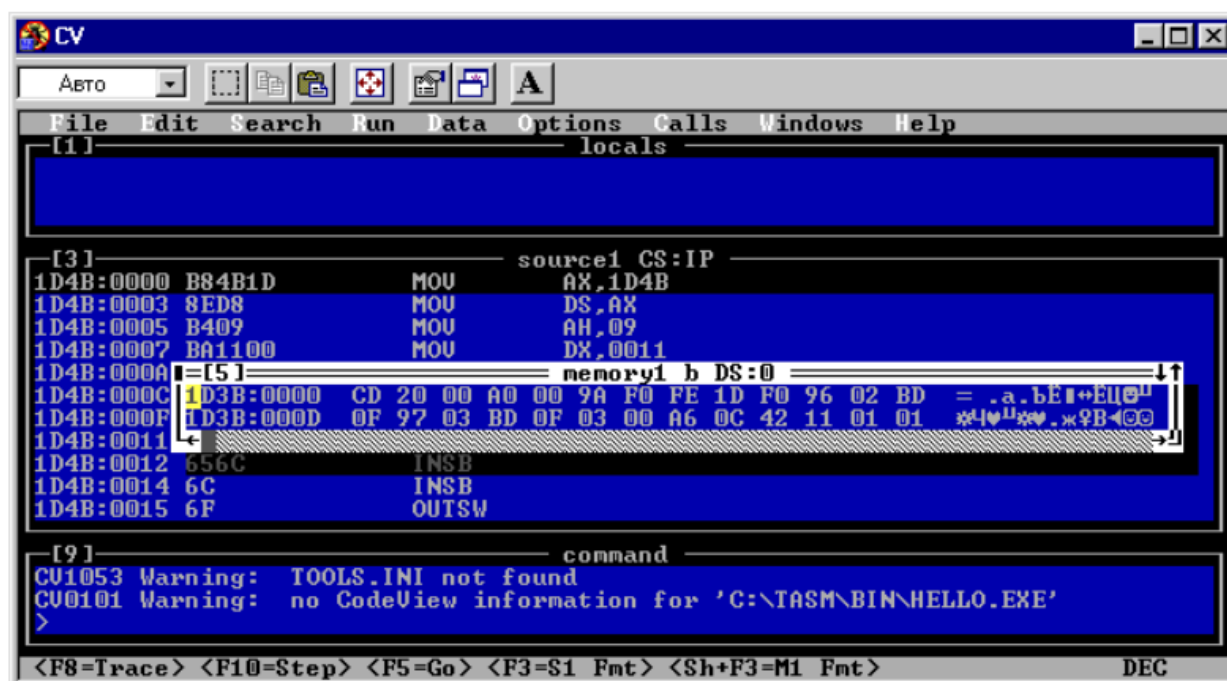


Рисунок 5 – Содержимое памяти

Далее для отладки используют трассировку (F8) - пошаговое выполнение программы, проверяя значения регистров, флагов и переменных.

При ассемблировании программы вы можете использовать в одной командной строке несколько опций, разделяя их пробелами, что не является обязательным. Ниже приводятся несколько примеров:

```
tasm /h
tasm -l -c ff
tasm /l /c ff
tasm -zi ff
tasm -l -iC:\INCLUDES ff
```

Вместо ассемблирования первая команда просто выводит список опций командной строки. Чтобы распечатать его, используйте команду `tasm /h > prn`. Вторая команда создает файл листинга с перекрестными номерами строк (#10, #25 и т.п.) в конце. Третья команда, делая то же самое и просто показывает, что при определении опций можно использовать косую черту вместо короткой черты. Четвертая команда добавляет в FF.OBJ информацию для Turbo Debugger. Последняя команда создает файл листинга и определяет путь включаемых (дополнительно используемых) файлов. Включаемые файлы - это отдельные текстовые файлы, которые по вашему желанию Turbo Assembler вставляет в программу. Программа 2.1 не использует включаемые файлы, следовательно, эта команда не приведет к каким-либо практическим результатам. Turbo Linker также имеет различные опции командной строки, определяемые аналогичным образом, за исключением того, что некоторые ранние версии TLINK требуют использования опций с косой чертой (/m), а не дефиса (-t). Некоторые версии компоновщиков допускают использование, как косой черты, так и дефиса, но при наборе нескольких опций с косой чертой в одной строке они должны разделяться пробелами. Ниже приводятся несколько примеров применения опций в Turbo Linker:

```
tlink -v ff
tlink /v ff
tlink -m -l ff tlink
/m /l ff tlink -x ff
tlink /x ff
```

Опции -v или /v в первых двух строках подготавливают FF.EXE для использования его с Turbo Debugger. Следующие строки определяют две опции, которые приводят к созданию расширенного файла отображений (FF.MAP) и добавляют в этот файл вспомогательную информацию о номерах строк (/l). Опции /x и -x запрещают Turbo Linker создавать файл отображений, что приводит к небольшой экономии дискового пространства и времени компоновки. Применяйте эти команды, если нет необходимости в файле отображений, который показывает организацию памяти программы и обычно используется отладчиками либо как часть программной документации.

Опции Tasm:

- /a Алфавитное упорядочение сегмента
- /b Не имеет никакого эффекта.
- /c Перекрестная ссылка в распечатке файла.
- /d Определяет символ.
- /e Генерирует команды эмулятора с плавающей точкой.
- /h,/? Отображает экран справки.
- /i Устанавливает путь для включаемого файла.
- /j Определяет директиву запуска ассемблера.
- /kh Максимальное число позволенных символов

ks Максимальный размер пространства под строку Турбо Ассемблера.
 /l Генерирует файл распечатки.
 /la Показывает код интерфейса высокого уровня в распечатке файла.
 /m # Позволить ☐ многократным проходам решать вперед ссылки
 /ml Обработывает все символы как чувствительные к регистру.
 /mu Преобразовывает символы в верхний регистр
 /mx Делает общие(public) и внешние(external) символы(глобальные) чувствительные к регистру .
 /mv# Установить максимальное значение длины для символов
 /n Подавляет таблицу символов в распечатке файла.
 /o Вывод tlink-совместимые объекты с допускаемой оверлейной поддержкой.
 /oi Вывод совместимых с IBM компоновщиком объектов IBM линейно - выполнимый компоновщик.
 /op Вывод Pharlap совместимые объекты
 /os Вывод tlink-совместимые объекты без оверлейных программ.
 /p Проверяет на чистоту код защищенного режима.
 /q Удаляет все записи из .OBJ, в которых нет необходимости для компонования.
 /r Генерирует действительные с плавающей запятой команды
 /s Последовательное упорядочение(заказывание) сегмента. (Значение по умолчанию)
 /t Подавляет сообщения об успешной трансляции.
 /v Опция совместимость.
 /w Генерация предупреждающих сообщений
 /x Включает неправильные условные выражения в распечатку.
 x/z Отображает исходные линии наряду с сообщениями об ошибках.
 /zd Допускает информацию номера строки в .OBJ
 /zi Допускает информацию отладки в объектном файле.

Листинг 1

```

assume CS:CodeSg, DS:CodeSg
CodeSg segment 'CODE'
begin: mov AX, CodeSg
mov DS, AX
;
mov AH, 09h
mov DX, offset message
int 21h
;
mov AX, 4C00h
int 21h
;
message db 'Hello World!', '$'
;
CodeSg ends
end begin
  
```

Организация программы

Сегменты

Каждая программа, написанная на любом языке программирования, состоит из одного или нескольких сегментов. Обычно область памяти, в которой находятся команды, называют

сегментом кода, область памяти с данными — сегментом данных и область памяти, отведенную под стек, — сегментом стека. Разумеется, ассемблер позволяет изменять устройство программы как угодно — помещать данные в сегмент кода, разносить код на множество сегментов, помещать стек в один сегмент с данными или вообще использовать один сегмент для всего.

Сегмент программы описывается директивами `SEGMENT` и `ENDS`.

имя_сегмента `segment readonly` выравнив. тип разряд 'класс'

...

имя_сегмента `ends`

Имя сегмента — метка, которая будет использоваться для получения сегментного адреса, а также для комбинирования сегментов в группы.

Все пять операндов директивы `SEGMENT` необязательны.

`READONLY`. Если этот операнд присутствует, `MASM` выдаст сообщение об ошибке на все команды, выполняющие запись в данный сегмент. Другие ассемблеры этот операнд игнорируют.

Выравнивание. Указывает ассемблеру и компоновщику, с какого адреса может начинаться сегмент. Значения этого операнда:

`BYTE` — с любого адреса;

`WORD` — с четного адреса;

`DWORD` — с адреса, кратного 4;

`PARA` — с адреса, кратного 16 (граница параграфа);

`PAGE` — с адреса, кратного 256.

По умолчанию используется выравнивание по границе параграфа.

Тип. Выбирает один из возможных типов комбинирования сегментов:

— тип `PUBLIC` (иногда используется синоним `MEMORY`) означает, что все такие сегменты с одинаковым именем, но разными классами будут объединены в один;

— тип `STACK` — то же самое, что и `PUBLIC`, но должен использоваться для сегментов стека, потому что при загрузке программы сегмент, полученный объединением всех сегментов типа `STACK`, будет использоваться как стек;

— сегменты типа `COMMON` с одинаковым именем также объединяются в один, но не последовательно, а по одному и тому же адресу, следовательно, длина суммарного сегмента будет равна не сумме длин объединяемых сегментов, как в случае `PUBLIC` и `STACK`, а длине максимального. Таким способом иногда можно формировать оверлейные программы;

— тип `AT` — выражение указывает, что сегмент должен располагаться по фиксированному абсолютному адресу в памяти. Результат выражения, использующегося в качестве операнда для `AT`, равен этому адресу, деленному на 16. Например: `segment at 40h` — сегмент, начинающийся по абсолютному адресу `0400h`. Такие сегменты обычно содержат только метки, указывающие на области памяти, которые могут потребоваться программе; `PRIVATE` (значение по умолчанию) — сегмент такого типа не объединяется с другими сегментами.

Разрядность. Этот операнд может принимать значения `USE16` и `USE32`. Размер сегмента, описанного как `USE16`, не может превышать 64 Кб, и все команды и адреса в этом сегменте считаются 16-битными. В этих сегментах все равно можно применять команды,

использующие 32-битные регистры или ссылающиеся на данные в 32-битных сегментах, но они будут использовать префикс изменения разрядности операнда или адреса и окажутся длиннее и медленнее. Сегменты USE32 могут занимать до 4 Гб, и все команды и адреса в них по умолчанию 32-битные. Если разрядность сегмента не указана, по умолчанию используется USE16 при условии, что перед директивой .MODEL не применялась директива задания допустимого набора команд .386 или старше.

Класс сегмента — это любая метка, взятая в одинарные кавычки. Все сегменты с одинаковым классом, даже сегменты типа PRIVATE, будут расположены в исполняемом файле непосредственно друг за другом.

Для обращения к любому сегменту следует сначала загрузить его сегментный адрес (или селектор в защищенном режиме) в какой-нибудь сегментный регистр. Если в программе определено много сегментов, удобно объединить несколько сегментов в группу, адресуемую с помощью одного сегментного регистра:

имя_группы group имя_сегмента...

Операнды этой директивы — список имен сегментов (или выражений, использующих оператор SEG), которые объединяются в группу. Имя группы теперь можно применять вместо имен сегментов для получения сегментного адреса и для директивы ASSUME.

assume регистр:связь...

Директива ASSUME указывает ассемблеру, с каким сегментом или группой сегментов связан тот или иной сегментный регистр. В качестве операнда «связь» могут использоваться имена сегментов, имена групп, выражения с оператором SEG или слово «NOTHING», означающее отмену действия предыдущей ASSUME для данного регистра. Эта директива не изменяет значений сегментных регистров, а только позволяет ассемблеру проверять допустимость ссылок и самостоятельно вставлять при необходимости префиксы переопределения сегментов, если они необходимы.

Перечисленные директивы удобны для создания больших программ на ассемблере, состоящих из разнообразных модулей и содержащих множество сегментов. В повседневном программировании обычно используется ограниченный набор простых вариантов организации программы, известных как модели памяти.

Листинг 2

```
CodeSg segment 'CODE'
assume CS:CodeSg, DS:DataSg, SS:StackSg
begin: mov AX, DataSg
mov DS, AX
;
; добавить код сюда
;
mov AX, 4C00h
int 21h
CodeSg ends
;-----
DataSg segment 'DATA'
;
; описать данные здесь
;
DataSg ends
;-----
StackSg segment stack 'STACK'
```

```
db 256 dup(0)
StackSg ends
;-----
end begin
```

Модели памяти и упрощенные директивы определения сегментов

Модели памяти задаются директивой `.MODEL`

`.model` модель, язык, модификатор, где модель — одно из следующих слов:

`TINY` — код, данные и стек размещаются в одном и том же сегменте размером до 64 Кб.

Эта модель памяти чаще всего используется при написании на ассемблере небольших программ;

`SMALL` — код размещается в одном сегменте, а данные и стек — в другом (для их описания могут применяться разные сегменты, но объединенные в одну группу). Эту модель памяти также удобно использовать для создания программ на ассемблере;

`COMPACT` — код размещается в одном сегменте, а для хранения данных могут использоваться несколько сегментов, так что для обращения к данным требуется указывать сегмент и смещение (данные дальнего типа);

`MEDIUM` — код размещается в нескольких сегментах, а все данные — в одном, поэтому для доступа к данным используется только смещение, а вызовы подпрограмм применяют команды дальнего вызова процедуры;

`LARGE` и `HUGE` — и код, и данные могут занимать несколько сегментов;

`FLAT` — то же, что и `TINY`, но используются 32-битные сегменты, так что максимальный размер сегмента, содержащего и данные, и код, и стек, — 4 Мб.

Язык — необязательный операнд, принимающий значения `C`, `PASCAL`, `BASIC`, `FORTRAN`, `SYSCALL` и `STDCALL`. Если он указан, подразумевается, что процедуры рассчитаны на вызов из программ на соответствующем языке высокого уровня, следовательно, если указан язык `C`, все имена ассемблерных процедур, объявленных как `PUBLIC`, будут изменены так, чтобы начинаться с символа подчеркивания, как это принято в `C`.

Модификатор — необязательный операнд, принимающий значения

`NEARSTACK` (по умолчанию) или `FARSTACK`. Во втором случае сегмент стека не будет объединяться в одну группу с сегментами данных.

После того как модель памяти установлена, вступают в силу упрощенные директивы определения сегментов, объединяющие действия директив `SEGMENT` и `ASSUME`. Кроме того, сегменты, объявленные упрощенными директивами, не требуется закрывать директивой

`ENDS` — они закрываются автоматически, как только ассемблер обнаруживает новую директиву определения сегмента или конец программы.

Директива `.CODE` описывает основной сегмент кода

`.code имя_сегмента`

эквивалентно

`_TEXT segment word public 'CODE'`

для моделей `TINY`, `SMALL` и `COMPACT` и

`name _TEXT segment word public 'CODE'`

для моделей `MEDIUM`, `HUGE` и `LARGE` (`name` — имя модуля, в котором описан данный сегмент). В этих моделях директива `.CODE` также допускает необязательный операнд — имя определяемого сегмента, но все сегменты кода, описанные так в одном и том же модуле, объединяются в один сегмент с именем `NAME_TEXT`.

`.stack` размер

Директива `.STACK` описывает сегмент стека и эквивалентна директиве `STACK segment para public 'stack'` Необязательный параметр указывает размер стека. По умолчанию он равен 1 Кб.

`.data`

Описывает обычный сегмент данных и соответствует директиве

`_DATA segment word public 'DATA'`

`.data?`

Описывает сегмент неинициализированных данных:

`_BSS segment word public 'BSS'`

Этот сегмент обычно не включается в программу, а располагается за концом памяти, так что все описанные в нем переменные на момент загрузки программы имеют неопределенные значения.

`.const`

Описывает сегмент неизменяемых данных:

`CONST segment word public 'CONST'`

В некоторых операционных системах этот сегмент будет загружен так, что попытка записи в него может привести к ошибке.

`.fardata` имя_сегмента

Сегмент дальних данных:

имя_сегмента `segment para private 'FAR_DATA'`

Доступ к данным, описанным в этом сегменте, потребует загрузки сегментного регистра.

Если не указан операнд, в качестве имени сегмента используется `FAR_DATA`.

`.fardata?` имя_сегмента

Сегмент дальних неинициализированных данных:

имя_сегмента `segment para private 'FAR_BSS'`

Как и в случае с `FARDATA`, доступ к данным из этого сегмента потребует загрузки сегментного регистра. Если имя сегмента не указано, используется `FAR_BSS`.

Во всех моделях памяти сегменты, представленные директивами `.DATA`, `.DATA?`, `.CONST`, `.FARDATA` и `.FARDATA?`, а также сегмент, описанный директивой `.STACK`, если не был указан модификатор `FARSTACK`, и сегмент `.CODE` в модели `TINY` автоматически объединяются в группу с именем `FLAT` — для модели памяти `FLAT` или `DGROUP` — для всех остальных моделей. При этом сегментный регистр `DS` (и `SS`, если не было `FARSTACK`, и `CS` в модели `TINY`) настраивается на всю эту группу, как если бы была выполнена команда `ASSUME`.

Способы адресации

Большинство команд процессора вызывается с аргументами, которые в ассемблере принято называть операндами. Например: команда сложения содержимого регистра с числом требует задания двух операндов — содержимого регистра и числа. Далее рассмотрены все существующие способы задания адреса хранения операндов — способы адресации.

Регистровая адресация

Операнды могут располагаться в любых регистрах общего назначения и сегментных регистрах. В этом случае в тексте программы указывается название соответствующего регистра, например команда, копирующая в регистр `AX` содержимое регистра `BX`,

записывается как

```
mov ax,bx
```

Непосредственная адресация

Некоторые команды (все арифметические команды, кроме деления) позволяют указывать один из операндов непосредственно в тексте программы, например команда `mov ax,2` помещает в регистр AX число 2.

Прямая адресация

Если известен адрес операнда, располагающегося в памяти, можно использовать этот адрес. Если операнд — слово, находящееся в сегменте, на который указывает ES, со смещением от начала сегмента 0001, то команда

```
mov ax,es:0001
```

поместит это слово в регистр AX. В реальных программах обычно для задания статических переменных используют директивы определения данных, которые позволяют ссылаться на статические переменные не по адресу, а по имени. Тогда, если в сегменте, указанном в ES, была описана переменная `word_var` размером в слово, можно записать ту же команду как `mov ax,es:word_var`

В таком случае ассемблер сам заменит слово «`word_var`» на соответствующий адрес. Если селектор сегмента данных находится в DS, имя сегментного регистра при прямой адресации можно не указывать, DS используется по умолчанию. Прямая адресация иногда называется адресацией по смещению.

Адресация отличается для реального и защищенного режимов. В реальном режиме (так же как и в режиме V86) смещение всегда 16-битное, это значит, что ни непосредственно указанное смещение, ни результат сложения содержимого разных регистров в более сложных методах адресации не могут превышать границ слова. При программировании для Windows, для DOS4G, PMODE и в других ситуациях, когда программа будет запускаться в защищенном режиме, смещение не может превышать границ двойного слова.

Косвенная адресация

По аналогии с регистровыми и непосредственными операндами адрес операнда в памяти также можно не указывать непосредственно, а хранить в любом регистре. До 80386 для этого можно было использовать только BX, SI, DI и BP, но потом эти ограничения были сняты и адрес операнда разрешили считывать также и из EAX, EBX, ECX, EDX, ESI, EDI, EBP и ESP (но не из AX, CX, DX или SP напрямую — надо использовать EAX, ECX, EDX, ESP соответственно или предварительно скопировать смещение в BX, SI, DI или BP). Например, следующая команда помещает в регистр AX слово из ячейки памяти, селектор сегмента которой находится в DS, а смещение — в BX:

```
mov ax,[bx]
```

Как и в случае прямой адресации, DS используется по умолчанию, но не во всех случаях: если смещение берут из регистров ESP, EBP или BP, то в качестве сегментного регистра используется SS. В реальном режиме можно свободно пользоваться всеми 32-битными регистрами, надо только следить, чтобы их содержимое не превышало границ 16-битного слова.

Адресация по базе со сдвигом

Теперь скомбинируем два предыдущих метода адресации: следующая команда

`mov ax,[bx+2]` помещает в регистр AX слово, находящееся в сегменте, указанном в DS, со смещением на 2 большим, чем число, находящееся в BX. Так как слово занимает ровно два

байта, эта команда поместила в AX слово, непосредственно следующее за тем, которое есть в предыдущем примере. Такая форма адресации используется в тех случаях, когда в регистре находится адрес начала структуры данных, а доступ надо осуществить к какому-нибудь элементу этой структуры. Другое важное применение адресации по базе со сдвигом — доступ из подпрограммы к параметрам, переданным в стеке, используя регистр BP (EBP) в качестве базы и номер параметра в качестве смещения, что детально разобрано в параграфе 5.2. Другие допустимые формы записи этого способа адресации:

```
mov ax,[bp]+2
```

```
mov ax,2[bp]
```

До 80386 в качестве базового регистра можно было использовать только BX, BP, SI или DI и сдвиг мог быть только байтом или словом (со знаком). Начиная с 80386 и старше, процессоры Intel позволяют дополнительно использовать EAX, EBX, ECX, EDX, EBP, ESP, ESI и EDI, так же как и для обычной косвенной адресации. С помощью этого метода можно организовывать доступ к одномерным массивам байт: смещение соответствует адресу начала массива, а число в регистре — индексу элемента массива, который надо считать. Очевидно, что, если массив состоит не из байт, а из слов, придется умножать базовый регистр на два, а если из двойных слов — на четыре. Для этого предусмотрен следующий специальный метод адресации.

Косвенная адресация с масштабированием

Этот метод адресации полностью идентичен предыдущему, за исключением того, что с его помощью можно прочесть элемент массива слов, двойных слов или учетверенных слов, просто поместив номер элемента в регистр:

```
mov ax,[esi*2]+2
```

Множитель, который может быть равен 1, 2, 4 или 8, соответствует размеру элемента массива — байту, слову, двойному слову, учетверенному слову соответственно. Из регистров в этом варианте адресации можно использовать только EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, но не SI, DI, BP или SP, которые можно было использовать в предыдущих вариантах.

Адресация по базе с индексированием

В этом методе адресации смещение операнда в памяти вычисляется как сумма чисел, содержащихся в двух регистрах, и смещения, если оно указано. Все следующие команды — это разные формы записи одного и того же действия:

```
mov ax,[bx+si+2]
```

```
mov ax,[bx][si]+2
```

```
mov ax,[bx+2][si]
```

```
mov ax,[bx][si+2]
```

```
mov ax,2[bx][si]
```

В регистр AX помещается слово из ячейки памяти со смещением, равным сумме чисел, содержащихся в BX и SI, и числа 2. Из шестнадцати-битных регистров так можно складывать только BX + SI, BX + DI, BP

+ SI и BP + DI, а из 32-битных — все восемь регистров общего назначения. Так же как и для прямой адресации, вместо непосредственного указания числа можно использовать имя переменной, заданной одной из директив определения данных. Так можно прочесть, например, число из двумерного массива: если задана таблица 10x10 байт, 2 — смещение ее начала от начала сегмента данных (на практике будет использоваться имя этой таблицы),

BX = 20, а SI = 7, приведенные команды прочитают слово, состоящее из седьмого и восьмого байт третьей строки. Если таблица состоит не из одиночных байт, а из слов или двойных слов, удобнее использовать следующую, наиболее полную форму адресации. Смещение может быть байтом или двойным словом. Если ESP или EBP используются в роли базового регистра, селектор сегмента операнда берется по умолчанию из регистра SS, во всех остальных случаях — из DS.

Примеры использования режимов адресации:

Пример 1 (регистровая адресация):

mov AX, BX ; копируются данные из BX в AX

mov DS, AX ; сегментный регистр DS инициализируется числом из AX

Пример 2 (непосредственная адресация):

; счетчик проинициализировать числом 10:

mov CX, 10

; нельзя инициализировать сегментный регистр непосредственным значением:

mov DS, 0000h ; недопустимо !

; можно сделать так:

mov AX, 0000h

mov DS, AX

Пример 3 (прямая адресация):

; скопировать в BX слово по адресу 0000:0000h:

mov AX, 0000h

mov ES, AX

mov BX, ES:0000h

; если в сегменте, на который указывает DS, есть переменная с именем X и

; размера слово, то значение хранящееся в DX можно скопировать в X так:

mov X, DX

Пример 4 (косвенная адресация):

; пусть в сегменте, на который указывает DS, первым объявлено слово X,

; а следом за ним - слово Y; тогда их значения можно скопировать так:

mov BX, 0000h

mov AX, [BX]

mov BX, 0002h

mov CX, [BX]

Пример 5 (адресация по базе со сдвигом):

; следующие команды из примера 4 можно заменить на одну:

; mov BX, 0002h

; mov CX, [BX]

mov CX, [BX]+2

Пример 6 (адресация по базе с индексированием):

; пусть объявлен массив слов X; тогда получить значения

; первого, третьего и пятого элемента можно так:

mov BX, offset X

mov SI, 0000h

mov AX, [BX][SI] ; 1

add SI, 4

mov AX, [BX][SI] ; 3

add SI, 4

mov AX, [BX][SI] ; 5

Ниже приведены примеры использования команд push/pop:

Пример 7: Временное хранение данных.

push eax ; сохраняем текущее значение eax в стеке

... ; выполняем какие нибудь действия,
... ; использующие еах
pop еах ; восстанавливаем прежнее значение еах

Пример 8: Копирование содержимого одного сегментного регистра в другой.

```
push ds  
pop es
```

Пример 9: Выделение старшей части расширенного регистра данных (с помещением значения в другой регистр).

```
push еах ; помещаем значение еах в стек (4 байта)  
pop ах ; извлекаем младшую часть (2 байта)  
pop bx ; извлекаем старшую часть (2 байта)
```

Структура программы

Программа на языке ассемблера состоит из строк, имеющих следующий вид:

метка команда/директива операнды ; комментарий

Причем все эти поля необязательны. Метка может быть любой комбинацией букв английского алфавита, цифр и символов `_`, `$`, `@`, `?`, но цифра не может быть первым символом метки, а символы `$` и `?` иногда имеют специальные значения и обычно не рекомендуются к использованию. Большие и маленькие буквы по умолчанию не различаются, но различие можно включить, задав ту или иную опцию в командной строке ассемблера. Во втором поле, поле команды, может располагаться команда процессора, которая транслируется в исполняемый код, или директива, которая не приводит к появлению нового кода, а управляет работой самого ассемблера. В поле операндов располагаются требуемые командой или директивой операнды (то есть нельзя указать операнды и не указать команду или директиву). И наконец, в поле комментариев, начало которого отмечается символом `;` (точка с запятой), можно написать все что угодно — текст от символа `<;>` до конца строки не анализируется ассемблером.

Для облегчения читаемости ассемблерных текстов принято, что метка начинается на первой позиции в строке, команда — на 17-й (две табуляции), операнды — на 25-й (три табуляции) и комментарии — на 41-й или 49-й. Если строка состоит только из комментария, его начинают с первой позиции.

Если метка располагается перед командой процессора, сразу после нее всегда ставится символ `<:>` (двоеточие), который указывает ассемблеру, что надо создать переменную с этим именем, содержащую адрес текущей команды:

```
some_loop:  
lodsw ; считать слово из строки,  
cmp ax,7 ; если это 7 - выйти из цикла  
loopne some_loop
```

Когда метка стоит перед директивой ассемблера, она обычно оказывается одним из операндов этой директивы и двоеточие не ставится. Рассмотрим директивы, работающие напрямую с метками и их значениями, — LABEL, EQU и =.

метка label тип

Директива LABEL определяет метку и задает ее тип. Тип может быть одним из: BYTE (байт), WORD (слово), DWORD (двойное слово), FWORD (6 байт), QWORD (учетверенное слово), TBYTE (10 байт), NEAR (ближняя метка), FAR (дальняя метка). Метка получает значение, равное адресу следующей команды или следующих данных, и тип, указанный явно. В зависимости от типа команда

```
mov метка,0
```

запишет в память байт (слово, двойное слово и т.д.), заполненный нулями, а команда `call` метка

выполнит ближний или дальний вызов подпрограммы.

С помощью директивы `LABEL` удобно организовывать доступ к одним и тем же данным, как к байтам, так и к словам, определив перед данными две метки с разными типами.

метка `equ` выражение

Директива `EQU` присваивает метке значение, которое определяется как результат целочисленного выражения в правой части. Результатом этого выражения может быть целое число, адрес или любая строка символов:

```
truth equ 1
message1 equ 'Try again$'
var2 equ 4[si]
cmp ax,truth ; cmp ax,1
db message1 ; db 'Try again$'
mov ax,var2 ; mov ax, 4[si]
```

Директива `EQU` чаще всего используется с целью введения параметров, общих для всей программы, аналогично команде `#define` препроцессора языка C.

метка = выражение

Директива = эквивалентна `EQU`, но определяемая ею метка может принимать только целочисленные значения. Кроме того, метка, указанная этой директивой, может быть переопределена.

Каждый ассемблер предлагает целый набор специальных предопределенных меток — это может быть текущая дата (`@date` или `??date`), тип процессора (`@cpu`) или имя того или иного сегмента программы, но единственная предопределенная метка, поддерживаемая всеми рассматриваемыми нами ассемблерами, — `$`. Она всегда соответствует текущему адресу. Например, команда

```
jmp $
```

выполняет безусловный переход на саму себя, так что создается вечный цикл из одной команды.

Директивы распределения памяти

Псевдокоманды определения переменных

Псевдокоманда — это директива ассемблера, которая приводит к включению данных или кода в программу, хотя сама она никакой команде процессора не соответствует. Псевдокоманды определения переменных указывают ассемблеру, что в соответствующем месте программы располагается переменная, определяют тип переменной (байт, слово, вещественное число и т.д.), задают ее начальное значение и ставят в соответствие переменной метку, которая будет использоваться для обращения к этим данным. Псевдокоманды определения данных записываются в общем виде следующим образом:

имя_переменной `d*` значение

где `D*` — одна из нижеприведенных псевдокоманд:

`DB` — определить байт;

`DW` — определить слово (2 байта);

`DD` — определить двойное слово (4 байта);

`DF` — определить 6 байт (адрес в формате 16-битный селектор: 32-битное смещение);

`DQ` — определить учетверенное слово (8 байт);

`DT` — определить 10 байт (80-битные типы данных, используемые FPU).

Поле значения может содержать одно или несколько чисел, строк символов (взятых в одиночные или двойные кавычки), операторов ? и DUP, разделенных запятыми. Все установленные таким образом данные окажутся в выходном файле, а имя переменной будет соответствовать адресу первого из указанных значений. Например, набор директив

```
text_string db 'Hello world!'
```

```
number dw 7
```

```
table db 1,2,3,4,5,6,7,8,9,0Ah,0Bh,0Ch,0Dh,0Eh,0Fh
```

```
float_number dd 3.5e7
```

заполняет данными 33 байта. Первые 12 байт содержат ASCII-коды символов строки «Hello world!», и переменная `text_string` указывает на первую букву в этой строке, так что команда `mov al,text_string`

считает в регистр AL число 48h (код латинской буквы H). Если вместо точного значения указан знак ?, переменная считается неинициализированной и ее значение на момент запуска программы может оказаться любым. Если нужно заполнить участок памяти повторяющимися данными, используется специальный оператор DUP, имеющий формат счетчик DUP (значение). Например, вот такое определение:

```
table_512w dw 512 dup(?)
```

создает массив из 512 неинициализированных слов, на первое из которых указывает переменная `table_512w`. В качестве аргумента в операторе DUP могут выступать несколько значений, разделенных запятыми, и даже дополнительные вложенные операторы DUP.

Задание для выполнения

1. Изучить теоретический материал.
2. Ознакомиться с пакетами MASM и TASM, воспользовавшись программой из работы.
3. Приведённые примеры скомпилировать и посмотреть в отладчике.

Контрольные вопросы

1. Что такое макроопределения и как их использовать в языке ассемблера?
2. Какие модели памяти используются при написании программ на языке Ассемблера, чем они отличаются?
3. Какова структура программы на языке Ассемблера?
4. Как происходит получение выполняемого кода программы?
5. Какую опцию необходимо применять при ассемблировании программы, для получения возможности ее отлаживать?
6. Как распечатать листинг программы?
7. Как правильно заканчивать программу на языке ассемблера?
8. Какие типы данных позволяет создать директива DB?

Практическое занятие 8 и 9

Тема: Процедурное программирование. Введение в язык программирования R.

Цель работы

Познакомиться основами языка программирования R.

Методические указания

Среда разработки R состоит из нескольких ключевых компонентов: интерпретатор языка, математические библиотеки, компиляторы Fortran/C/C++ и различные пользовательские пакеты.

Соответственно, установка R так или иначе включает в себя установку этих компонентов. Наиболее часто используемая схема — это установка языка с сайта проекта и одновременно с этим базового компилятора (GNU R) и стандартных библиотек для работы с алгоритмами линейной алгебры (BLAS/LAPACK).

Более подробно процесс установки и настройки R описан по ссылке ([Начало работы в R](#)).

Синтаксис языка

Операторы

Присваивание (<-, <<-, =)

Основной оператор присваивания в R — это оператор <-. Например, `x <- 5`. Если до этого объект не был создан, то таким образом мы создадим объект `x`, в котором содержится значение 5. Если до этого объект существовал, то ему будет присвоено новое значение (перезаписано).

Оператор `=` аналогичен оператору `<-` и может быть использован для левостороннего присваивания. Одновременно оператор `=` используется при указании значений аргументов функций. Большинство руководств по стилю рекомендуют именно такое использование операторов `<-` и `=`: первый для присваивания, а второй — при использовании функций.

Оператор `<<-` также является оператором левостороннего присваивания, как и `<-`, однако используется редко и в основном в функциях. Основное отличие: с помощью оператора `<<-` значения присваиваются в родительском окружении. Это используется, чтобы созданный во время выполнения функции промежуточный объект вывести за пределы области видимости функции и сохранить в родительском окружении. Родительским окружением может быть как глобальное окружение (чаще всего), так и какое-либо иное окружение, в котором вызывается функция (подробнее см. Выполнение выражений):

```
f1 <- function(x) {  
  tmp1 <<- paste("the result of <<- operator in f1()")  
  tmp2 <- x * 2  
  return(tmp2)  
}  
  
print(f1(2))  
## [1] 4  
print(tmp1)  
## [1] "the result of <<- operator in f1()"
```

Вместо оператора `<-` для изменения значений объектов в различных окружениях рекомендуется использовать функцию `assign()`. В этой функции с помощью аргумента `envir` можно указать, в каком окружении происходит присвоение. Окружением может быть как пользовательское окружение, так и родительское (`parent.env()`)/глобальное (`.GlobalEnv`):

```
f2 <- function(x) {
  assign(x = 'tmp1', value = paste('the result of assign() in f2()'), envir = .GlobalEnv)
  tmp2 <- x * 2
  return(tmp2)
}
```

```
print(f2(2))
## [1] 4
print(tmp1)
## [1] "the result of assign() in f2()"
```

Несмотря на то, что операторы присваивания позволяют множественное присваивание, а операторы `->` и `->>` — ещё и правостороннее присваивание, выражения вида `x = y = z = 5` или `5 <- x -> y` лучше не использовать, так как они уменьшают читабельность и прозрачность кода.

В редких случаях в левой части выражения присвоения при левостороннем присваивании может стоять не имя объекта, а вызов функции. Как правило, это касается функций, несущих определённую информацию об объекте. Наиболее часто встречающийся пример — создание или присвоение названий колонок в таблице:

Арифметические действия

Арифметические операторы аналогичны обычным математическим операторам:

- `+`, `-` — бинарные и унарные сложение и вычитание;
- `*`, `/` — умножение и деление;
- `^` или `**` — возведение в степень;
- `/%` — деление нацело;
- `%%` — остаток при делении нацело.

Сравнение и логические операторы

Операторы сравнения или логические операторы обычно используются при необходимости указать какие-то условия для выбора элементов вектора или таблицы, а также в конструкциях `if/else` (см. Условные операторы).

Операторы сравнения:

- `<`, `<=` — меньше, меньше или равно;
- `>`, `>=` — больше, больше или равно;
- `==`, `!=` — равно и не равно.

Логические операторы:

- `!` — отрицание, например, `!is.na(x)`;
- `&`, `&&` — И, бинарное И (оператор `&` поэлементно объединяет элементы двух множеств, оператор `&&` объединяет только первые слева элементы каждого множества), например, объединяем результаты двух логических проверок:

```
# объединяются попарно TRUE TRUE и FALSE FALSE
```

```
c(3, 5) < 6 & c(7, 3) < 6
```

```
## [1] FALSE TRUE
```

```
# объединяются только первые элементы множеств — TRUE и FALSE
```

```
c(3, 5) < 6 && c(7, 3, 6) < 6
```

```
## [1] FALSE
```

|, || — ИЛИ, бинарное ИЛИ (аналогично операторам & и &&, обычно бинарные И/ИЛИ используются разработчиками при написании функций, чтобы исключить возможность неоднозначности операции).

Доступ к данным

Как правило, объекты в R — это таблицы, списки или сложные объекты с большим количеством вложенных элементов. Для того чтобы извлечь какой-то элемент или переписать его значение, используются операторы доступа.

\$ и @

Оператор \$ используется для обращения к именованному элементу списка, колонке датафрейма и прочим элементам структуры объектов. Вызов выглядит следующим образом: `<object_name>$<element_name>`. В выражении можно использовать несколько операторов, особенно часто это необходимо при работе со сложными вложенными списками и объектами S3- или R6-классов. Оператор \$ обладает также полезной особенностью: название элемента можно вводить не полностью, но введенная часть должна однозначно определять требуемый элемент.

Оператор @ используется аналогично оператору \$ при работе с объектами S4-класса (доступ к слотам), однако требует полного названия требуемого элемента.

[и [[

Оператор [в определённой мере аналогичен оператору \$, однако обладает более гибким функционалом. Так, с помощью оператора можно выделять элементы по их полному названию, по номеру или вектору номеров в списке элементов, по логическому условию.

Для некоторых объектов оператор [переопределён и имеет дополнительные аргументы.

Оператор [[похож на операторы [и \$, однако на вход принимать может только одно значение — номер позиции или название элемента (по умолчанию название должно быть полным):

Доступ к объекту в пространстве имён (::, :::)

Операторы :: и ::: используются для доступа к объектам из какого-то определённого пространства имён (подробнее про пространства имён см. Окружения пакетов и пространство имён). Так, :: нередко используется для вызова функций любого установленного, но не подключённого пакета. Оператор ::: обеспечивает доступ вообще к любым неэкспортированным объектам пакетов, не только к функциям.

Чаще всего потребность в этих операторах возникает тогда, когда какая-то функция из специального пакета используется однократно и подключать весь пакет нет необходимости.

Например, установка пакета из github-репозитория с помощью пакета remotes: `remotes::install_github("ropensci/plotly")`. Также :: или ::: используются в ситуациях, когда в нескольких пакетах встречаются одинаковые названия функций или объектов. Например, `fromJSON()` встречается одновременно в пакетах jsonlite, RJSONIO и rjson.

Иерархия операторов

Операторы организованы в иерархию. Это позволяет сочетать их в разных выражениях.

Список операторов по убыванию приоритета выполнения:

Оператор	Значение
----------	----------

::, :::	доступ к значениям в пространстве имён
---------	--

\$, @	доступ к элементу объекта/слоту по названию
-------	---

[, [[доступ к элементу объекта/слоту по индексу
 ^ возведение в степень
 -, + унарные плюс и минус
 : оператор последовательности (короткая запись для seq())
 %any% специальные операторы (включая %% и %/%)
 *, / умножение и деление
 +, - сложение и вычитание
 <, >, <=, >=, ==, != логическое сравнение
 ! логическое отрицание
 &, && логическое И
 |, || логическое ИЛИ
 ~ знак формулы
 ->, ->> правое присваивание
 <-, <<- левое присваивание
 = левое присваивание
 ?, ?? справка

Конвейеры

%>%

Оператор конвейера %>% не входит в базовые библиотеки R и принадлежит пакету magrittr, однако весьма распространён, в частности, широко используется в пакетах экосистемы tidyverse. Оператор %>% используется для того, чтобы связать в последовательность несколько операций, когда в последующее выражение передаётся результат предыдущего. В результате сокращается количество кода и/или повышается читабельность. Например, три идентичных вычисления, первое использует оператор конвейера, второе — вложенные выражения, а третье просто каждую операцию производит в отдельном выражении:

```
library(magrittr)
```

```
x <- 12 ^ 2 %>% sqrt %>% -2
```

```
x <- sqrt(12^2) - 2
```

```
x <- 12
x <- x ^ 2
x <- sqrt(x)
x <- x - 2
```

|>

Оператор конвейера |> совсем недавно вошел в base R. Операторы %>% и |> одинаковы по смыслу, но всё же неравнозначны. Собственно, поэтому команда R Core и не стала использовать %>% для своего конвейера.

Основное различие заключается в том, что они выполняются в разное время: %>% применяется при выполнении выражений, в рантайме; оператор |> обрабатывается интерпретатором на этапе чтения кода, и только потом происходит выполнение конвейера выражений. Такая организация делает применение |> надёжным и не вносящим дополнительных расходов.

Служебные символы

Конец выражения

Для маркировки конца выражения в R используется `;`. Однако парсер интерпретатора читает код построчно. Это значит, что если не писать более одного выражения на одной строке, то указывать `;` нет необходимости, достаточно переноса строки. В результате вот такие два выражения идентичны:

```
# два выражения на одной строке
x <- 5; y <- 3
x + y
## [1] 8

# два выражения отдельно
x <- 5
y <- 3
x + y
## [1] 8
```

Фигурные скобки

Фигурные скобки используются в тех ситуациях, когда необходимо несколько выражений объединить в один блок. Чаще всего это необходимо при объявлении функций, в циклах и в некоторых случаях при использовании условных операторов (`if ... else/ifelse/switch`, см. Условные операторы). Результатом выполнения блока будет результат последнего выражения:

```
result <- {
  y <- 2 * 3
  z <- y ^ 2
  x <- y + z
}
print(result)
## [1] 42
```

В том случае, когда тело функции или цикла составляет одно выражение, фигурные скобки можно опустить. В таком случае выражение после `function()/for` на той же строке или одна следующая строка считается как относящиеся к функции/циклу:

```
for (i in 1:3)
  print(i)
## [1] 1
## [1] 2
## [1] 3
```

Комментарии

В R для комментирования используется символ `#`. Если в какое-то место на строке поставить знак комментария, то вся оставшаяся часть строки будет считаться закоментированной. Исключение — если знак комментария находится внутри выражения, окруженного кавычками:

Обычно строку комментария ставят перед комментируемым выражением. Однако если комментарий короткий, то можно, писать комментарий после комментируемого выражения, на той же строке.

Кавычки

В R используются три вида кавычек: кавычки (“”), апостроф (') и обратный апостроф (`). Простые кавычки и апостроф взаимозаменяемы, в коде можно использовать и то, и другое, в том числе и комбинированно. Единственное, следует выдерживать стилистическую однородность и использовать какой-нибудь тип как основной, а второй — только в тех ситуациях, когда надо написать сложное выражение.

Зарезервированные слова

Как и во многих других языках, в R часть слов является зарезервированными, то есть за ними закреплено определённое значение (функционал), и пользователь не имеет возможности использовать их в качестве имён объектов. Основные зарезервированные слова (согласно документации R Foundation) можно посмотреть с помощью ?Reserved:

```
if else repeat while function for in next break
TRUE FALSE NULL Inf NaN
NA NA_integer_ NA_real_ NA_complex_ NA_character_
... ..1 ..2 etc
```

Слова, которые уже обозначают функции используемых пакетов или глобальных переменных, не рекомендуется использовать в качестве названий объектов.

Основные типы и структуры данных

Векторы

Атомарные типы

Вектор — простой набор элементов одного типа данных. В R для создания векторов используется шесть базовых (их иногда ещё называют атомарными) типов.

`character` — используется для текстовых значений. Любой набор знаков, заключённый в кавычки, в R воспринимается как текстовое значение. Два нюанса — знак `#`, заключённый в кавычки, перестаёт быть символом, обозначающим комментарий. А обратный слэш `\` всё так же остается знаком экранирования, что позволяет использовать в текстовых строках такие специальные символы, как, например, `\n`.

`complex` — используется при работе с комплексными числами.

`numeric` (или `double`) — используется для числовых значений (с плавающей точкой).

`integer` — целочисленные значения (по форме хранения, а не математическое множество целых).

`logical` — логические значения TRUE/FALSE.

`raw` — достаточно редко используемый тип данных, побайтовая запись в шестнадцатеричном формате.

Особенности числовых типов

Тип `numeric` в R хранится в виде чисел с плавающей запятой и двойной точностью. Числа с одинарной точностью можно симулировать с помощью функции `single()` или `as.single()`, приписав `numeric`-объекту атрибут `Csingle`. Однако это не настоящий хранимый формат, и его использование имеет смысл только при обращении к некоторым функциям, написанным на C или Fortran. Хранение чисел в формате с плавающей запятой может приводить к некоторым казусам, когда требуется каким-то образом обрабатывать числовые значения, которые не умещаются в 32 бита. Максимально возможное целочисленное значение можно определить с помощью `.Machine$integer.max`, всё, что больше него, будет преобразовано в `double` с плавающей запятой.

Создание векторов

Векторы создаются с помощью функции `c()` (`c` — от `combine`). Например:

```
x <- c(1, 3, 5)
print(x)
## [1] 1 3 5
```

В редких случаях используются векторы с именованными элементами (аналоги словарей в некоторых других языках программирования). В повседневной практике встречается достаточно редко. Как правило, именованные векторы можно получить в результате функции `sapply()` и некоторых подобных:

```
x <- c(var1 = 12, var2 = 23, var3 = 34)
print(x)
## var1 var2 var3
## 12 23 34
```

Простую неименованную последовательность атомарного типа можно задать также другими командами, наиболее часто используемые из них:

`seq()` — создание последовательности значений в указанном интервале с заданным шагом (`seq` от `sequence`). Например:

```
x <- seq(from = 10, to = 2, by = -2)
print(x)
## [1] 10 8 6 4 2
```

Выражение `:` — аналог функции `seq()`, используется для создания последовательности целых чисел, где последующее отличается от предыдущего на 1. Так, выражение `5:1` тождественно выражению `seq(5, 1, -1)`.

`rep()` — повтор какого-либо элемента заданное число раз. В качестве элемента может выступать практически любой объект R (в зависимости от осмысленности в целом подобного выражения). Аргументы `times` или `each` задают, какое количество раз повторять весь объект или каждый элемент объекта соответственно:

```
x <- c('a', 'b', 'c')
rep(x, times = 2)
## [1] "a" "b" "c" "a" "b" "c"
rep(x, each = 2)
## [1] "a" "a" "b" "b" "c" "c"
```

Если значения аргументов задавать позиционно, то первый аргумент — это итерируемый объект, а второй аргумент — аргумент `times`:

```
rep(x, 2)
## [1] "a" "b" "c" "a" "b" "c"
```

`sample()` — создание последовательности случайно выбранных значений из какого-то заданного вектора значений. Первый аргумент задаёт выборку, из которой необходимо извлечь подвыборку, второй аргумент (`size`) определяет объём извлекаемой выборки. Аргумент `replace` определяет, выборка извлекается с возвращением элементов или нет:

```
sample(x = 1:5, size = 5, replace = FALSE)
## [1] 1 2 5 4 3
sample(x = 1:5, size = 5, replace = TRUE)
## [1] 3 5 1 3 3
```

`vector()` — функция создания вектора значений определённого типа (обычно атомарного, но можно создавать и более сложные объекты, например, списки), где первый аргумент задаёт тип, а второй — длину вектора (количество элементов в последовательности). По умолчанию созданная последовательность заполнена значениями `FALSE`, если вектор логический, `0` и `"`, если вектор числовой или текстовый соответственно.

`raw()`, `logical()`, `integer()`, `numeric()`, `complex()`, `character()` — функции создания векторов, аналогичные `vector()`, для каждого из атомарных типов отдельно. В качестве аргумента эти функции принимают требуемую длину создаваемого вектора и создают вектор значений по умолчанию (`00` для `raw`-векторов, `FALSE` для логических векторов, `0` — для числовых и `"` — для текстовых):

```
character(length = 5)
## [1] "" "" "" "" ""
```

Вектор единичной длины

Простое присвоение `x <- 'alpha'` также будет созданием вектора. В R в принципе отсутствуют скалярные значения и любое единичное значение/константа — это всё тот же вектор, просто единичной длины.

```
x <- 'alpha'
length(x)
## [1] 1
```

Неявные преобразования

В R используется достаточно простая система автоматического преобразования атомарных типов по логике от самого строгого к самому гибкому, то есть по цепочке `logical -> integer -> numeric -> complex -> character` (тип `raw` не включается в эту систему). Если попытаться создать вектор из последовательности элементов разных атомарных типов, то все элементы будут преобразованы к самому общему типу в заданной последовательности:

```
print(c(TRUE, 1, 2, FALSE, FALSE))
## [1] 1 1 2 0 0
print(c(1, 3.6, 2, 5.1))
## [1] 1.0 3.6 2.0 5.1
print(c('a', 3.6, 'b', 5.1))
## [1] "a" "3.6" "b" "5.1"
print(c(TRUE, 3.6, 'b', 2))
## [1] "TRUE" "3.6" "b" "2"
```

Второе следствие неявных преобразований — возможность использовать конструкции вида `sum(5 == 5)`, потому что логический результат сравнения (`TRUE`) будет преобразован в целое числовое значение (`1`, так как `TRUE` преобразовывается в `1`, а `FALSE` — в `0`). Следовательно, можно вычислить количество верных утверждений. Подобные процедуры крайне часто используются в ситуациях, когда надо посчитать количество определённых значений в векторе или в колонке таблицы. Вообще во избежание неожиданных результатов рекомендуется остерегаться неявных преобразований и контролировать типы самостоятельно.

Массивы

В R, помимо подобных одномерных массивов (векторов), также можно создавать и многомерные массивы — **двумерные массивы** называются **матрицами**, **многомерные**

соответственно **многомерными массивами** или **многомерными матрицами**. Все элементы массивов также должны быть элементами одного типа.

Матрицы создаются с помощью функции `matrix()`, где первым аргументом задаётся набор значений, а вторым и третьим аргументами — количество строк и столбцов в матрице. Заполнение ячеек массива по умолчанию происходит по колонкам: сначала заполняется построчно первая колонка, потом вторая (если есть) и т. д. Название строк и колонок при желании можно задать с помощью аргумента `dimnames` в виде списка векторов названий:

```
x <- matrix(data = 1:4,
            nrow = 2, ncol = 2,
            dimnames = list(c('r1', 'r2'), c('c1', 'c2')))
print(x)
##   c1 c2
## r1 1 3
## r2 2 4
```

Многомерные массивы задаются аналогичным образом, только вместо аргументов количества строк и колонок используется аргумент `dim` (от *dimension*), задающий размерность массива. При `dim = c(2, 2)` будет создан двумерный массив 2×2 , матрица, как и в примере выше.

```
x <- array(data = 1:4,
           dim = c(2, 2),
           dimnames = list(c('r1', 'r2'), c('c1', 'c2')))
```

Если создать и вывести на печать трёхмерный массив, который можно назвать стопкой двумерных массивов, то будут выведены матрицы для каждого элемента третьего измерения, для *n*-мерных массивов аналогично. Например, ниже мы создаём и выводим трёхмерный массив размером $2 \times 2 \times 2$. В результате мы получаем две матрицы — `[, , v1]` и `[, , v2]`:

```
array(data = 1:8,
      dim = c(2, 2, 2),
      dimnames = list(c('r1', 'r2'), c('c1', 'c2'), c('v1', 'v2')))
## , , v1
##
##   c1 c2
## r1 1 3
## r2 2 4
##
## , , v2
##
##   c1 c2
## r1 5 7
## r2 6 8
```

Следует помнить, что массивы — это всё те же векторы (наборы элементов атомарных типов), просто с атрибутом размерности.

Списки

Списки — это коллекции объектов любых типов (в том числе и другие списки) любого уровня вложенности. По сути, списки представляют собой наборы ссылок на другие

объекты. Для создания списка используется команда `list()`, где в аргументах через запятую перечисляются все элементы, которые необходимо включить в список. Например:

```
x <- seq(from = 13, to = 0, by = -3)
y <- rep(x = 'c', times = 3)
z <- TRUE
```

```
my_list <- list(x, y, z)
print(my_list)
## [[1]]
## [1] 13 10 7 4 1
##
## [[2]]
## [1] "c" "c" "c"
##
## [[3]]
## [1] TRUE
```

Таблицы (датафреймы)

`data.frame` — базовый для R формат, который используется для представления таблиц. В R поддерживается классическая концепция таблиц, характерная для социальных наук и *data science*, когда по строкам расположены наблюдения, а колонки образуют пространство признаков этих наблюдений. На уровне структуры `data.frame` — это всё те же списки, в которых могут храниться разные по типу объекты, однако с требованием равенства длины объектов. Важно: все значения одной колонки могут быть только одного типа, а не как в Excel, OpenOffice или другом табличном процессоре.

Таблицы можно создать с помощью функции `data.frame()`, где аналогично созданию списка через запятую перечисляются векторы, значения которых будут представлены в виде колонок. Так как все колонки должны быть одной длины, то и длины векторов соответственно должны быть одинаковы. Возможны и другие варианты создания `data.frame`-объектов путем преобразования из других объектов с помощью функции `as.data.frame()`.

```
# создаем data.frame-объект
x <- seq(from = 10, to = 0, by = -2)
y <- rep(x = 'a', times = 6)
z <- sample(x = 1:20, size = 6, replace = FALSE)
```

```
dataset <- data.frame(column1 = x,
                      column2 = y,
                      column3 = z)
```

```
str(dataset)
## 'data.frame': 6 obs. of 3 variables:
## $ column1: num 10 8 6 4 2 0
## $ column2: chr "a" "a" "a" "a" ...
## $ column3: int 12 14 5 19 3 11
```

`data.frame` — это базовый класс для таблиц в R, однако в настоящее время используется, как правило, один из двух других подходов — `tibble` (из экосистемы *tidyverse*) или `data.table`. Оба эти класса наследуются от `data.frame`, поэтому в целом поддерживают базовые принципы и идеи `data.frame`.

Факторы

Факторы (factor) — специфичный для R формат, в других языках аналогом будет тип `enum` (enumerated type). Это упорядоченный вектор номинальных или интервальных значений, аналог категориальных шкал. При этом объект класса `factor` содержит как собственно значения, так и уровни, по которым эти значения упорядочены. Обычно факторы используют для упорядочивания номинальных, категориальных или текстовых данных, а также в различных статистических моделях и некоторых графиках.

Создадим вектор значений `low`, `medium`, `high` и преобразуем его в фактор, где `high` — значение с самым низким рангом, а `low` — с самым высоким:

```
quality <- c("low", "high", "medium", "high", "low", "medium", "high")
quality_f <- factor(quality, levels = c("high", "medium", "low"))
print(quality_f)
## [1] low  high  medium high  low  medium high
## Levels: high medium low
```

Дата и время

Класс Date используется при работе с датами. `Date`-объект не содержит информацию о времени или временной зоне и представляет собой количество дней с 1 января 1970 года (1970-01-01), представленное в человекочитаемом формате дат. В базовом R получить объект класса `Date` можно преобразованием других объектов с помощью функции `as.Date()`:

```
# преобразование текстового значения
x <- as.Date('2018-04-01')
class(x)
## [1] "Date"
# преобразование количества дней
x <- as.Date(17622, origin = '1970-01-01')
class(x)
## [1] "Date"
# преобразование POSIXct-значения
x <- as.POSIXct(1522540800, origin = '1970-01-01')
x <- as.Date(x)
class(x)
## [1] "Date"
```

Origin — это дата и время, от которых отсчитывается количество дней или секунд до настоящего момента. По умолчанию это таймстамп 1970-01-01 00:00:00, который служит точкой отсчета в Unix-системе (и прочих POSIX-совместимых операционных системах) описания моментов времени:

```
# снимаем атрибут класса Date, в результате получаем количество дней
unclass(as.Date('2018-04-01'))
## [1] 17622
# вычитаем полученное количество из исходной даты
as.Date('2018-04-01') - 17622
## [1] "1970-01-01"
```

Наиболее часто используемый формат представления дат — это формат `уууу-mm-dd`, однако нередко встречаются иные варианты, особенно если импортируются данные из

табличных процессоров типа Excel или Google Spreadsheet, например, mm/dd/yy или dd.mm.yyyy. В таких случаях в функции as.Date() необходимо указывать используемый формат представления дат:

```
x <- as.Date('4/1/18', format = '%m/%d/%y')
str(x)
## Date[1:1], format: "2018-04-01"
```

Для работы с данными даты и времени используют функции as.POSIXct() и as.POSIXlt() (cl от calendar time, lt от local time). Обе функции работают с данными в Unix-формате (количество секунд с 1970-01-01 00:00:00) и в качестве дополнительного аргумента требуют указать временную зону:

```
# создаём as.POSIXct-объект
x_ct <- as.POSIXct(1522571987, origin = '1970-01-01', tz = 'UTC')
str(x_ct)
## POSIXct[1:1], format: "2018-04-01 08:39:47"
# создаём as.POSIXlt-объект
x_lt <- as.POSIXlt(1522571987, origin = '1970-01-01', tz = 'UTC')
str(x_lt)
## POSIXlt[1:1], format: "2018-04-01 08:39:47"
```

Основное различие между функциями as.POSIXct() и as.POSIXlt() — первая аналогично Date принимает информацию о временной метке в числовом виде и представляет результат в текстовом виде, а также содержит метку временной зоны в атрибуте tz. Вторая возвращает именованный список параметров даты (день, месяц, год, час и т. д.), который также представляет собой таймстамп в указанной временной зоне:

Формулы

На данный момент формулы — специальный объект, который описывает схему взаимодействия зависимой переменной (таргета) и независимых переменных (предикторов), а также взаимодействие предикторов между собой. Формульная нотация для записи регрессионной модели предложена Роджерсом и Вилкинсоном в 1973 году (G. Wilkinson и Rogers 1973), позднее её включили в язык S (Chambers и Hastie 1991) и, следовательно, она попала в R. Простой пример формулы, описывающий одну зависимую переменную и две независимые:

$$y \sim x + z$$

Слева от знака тильды находится зависимая переменная (LHS, left-hand side). В стандартном использовании слева только одна переменная, если же необходимо указать несколько зависимых и/или их взаимодействие, то следует воспользоваться соответствующими пакетами. Справа от тильды, right-hand side (RHS), — две независимые переменные. Знак + в формулах означает независимое влияние переменных. По сути, формульная нотация — это своего рода DSL внутри R, так как в нотации привычные знаки и операторы имеют собственное значение:

- + используется для того, чтобы перечислить переменные-предикторы;
- используется, чтобы указать, влияние какого фактора необходимо исключить из модели;

: передаёт в качестве предиктора модели только взаимодействие факторов (аналогично $x_1 * x_2 - x_1 - x_2$);

* задаёт как изолированное влияние каждого фактора, так и их взаимодействие (аналогично $x_1 + x_2 + x_1:x_2$);

| используется в тех ситуациях, когда необходимо ввести в модель случайные эффекты;

-1 или 0 в формуле указывает, что из модели следует исключить случайный член (intercept);

. позволяет указывать в качестве предикторов все колонки в таблице, помимо зависимой переменной.

Формулу можно создать несколькими путями: просто создать текстовую переменную либо же воспользоваться функцией `formula()`, в конце концов, можно просто написать `my_formula = y ~ x`. Результаты потом можно использовать в соответствующих функциях. Текстовое описание модели в формуле позволяет создавать генераторы моделей (например, при переборе комбинаций предикторов в машинном обучении). Пример случайной генерации моделей, сочетающих два независимых предиктора и взаимодействие ещё двух предикторов:

```
# создаём вектор текстовых значений имён предикторов
predictors <- c('x1', 'x2', 'x3', 'x4')
for (i in 1:5) {
  tmp <- paste(
    # выбираем независимые факторы
    paste(sample(predictors, 2), collapse = ' + '),
    # выбираем взаимодействующие факторы
    paste(sample(predictors, 2), collapse = ':'),
    sep = ' + ')
  tmp <- paste('y ~', tmp)
  print(tmp)
}
## [1] "y ~ x2 + x3 + x1:x2"
## [1] "y ~ x4 + x3 + x2:x1"
## [1] "y ~ x2 + x4 + x4:x3"
## [1] "y ~ x4 + x3 + x1:x2"
## [1] "y ~ x3 + x4 + x4:x2"
```

Формулы обычно используются в линейных моделях, случайных лесах и многих других функциях, в которых требуется описание модели взаимосвязи нескольких переменных. Так как тильда в формулах обозначает, что ничего не надо вычислять и надо просто запомнить выражение и его окружение, возможны и другие варианты использования формул. Например, помимо соотношения таргета и предикторов, тильда и формульная нотация может использоваться для определения формы взаимодействия колонок при решейпинге таблиц.

Выражения и вызовы функции

Выражения в узком смысле объект класса `expression`, который содержит запись операции (операций) на R, чтобы можно было выполнить записанную операцию не сразу, а в необходимом месте (так называемое `unevaluated expression`, подробнее см. Выполнение выражений). В базовом R выражения можно создать с помощью функции `expression()` по аналогии с созданием вектора. При использовании таких выражений необходимо их явно выполнять с помощью функции `eval()`:

```
# создаём выражение из двух операций
my_exp <- expression(x <- 6, x ^ 2)
print(my_exp)
## expression(x <- 6, x^2)
# выполняем выражение
eval(my_exp)
## [1] 36
```

Второй вариант создания выражения — это парсинг текстовой строки, в котором также записана операция на языке R. Для подобного парсинга используется функция `parse()`, где строковая запись передаётся в аргумент `text` (подробнее см. Парсинг выражений из строки). Также в функции `parse()` можно указывать кодировку записи, что может быть полезно, когда скрипт был написан и сохранен в одной кодировке, а вызывается в рабочей среде с другой кодировкой. В примере ниже мы хотим записать две операции в одну строку, поэтому их необходимо разделить `;`:

```
# записываем операции в строковый объект
my_exp <- "x <- 6; x ^ 2"
class(my_exp)
## [1] "character"
# парсим строковую запись операций
my_exp <- parse(text = my_exp)
class(my_exp)
## [1] "expression"
# выполняем результат парсинга
eval(my_exp)
## [1] 36
```

Функция `call()` создаёт объект, несколько похожий на объект класса `expression`, только вместо записи какой-либо операции содержит запись вызова функции. Первым аргументом в `call()` передаётся название функции, последующими аргументами — значения, которые должны быть переданы в аргументы этой функции:

```
# создаём вызов функции возведения в степень
my_call <- call('^', 3, 2)
class(my_call)
## [1] "call"
# выполняем объект вызова функции
eval(my_call)
## [1] 9
```

На практике нередко встречается другой вариант работы с вызовами функции, когда с помощью `do.call()` вызывают функцию со списком аргументов и сразу получают результат. Самый наглядный пример — это вызов функции `merge()`:

```
df1 <- data.frame(var1 = letters[1:3], df1_col = sample(5, 3))
df2 <- data.frame(var1 = letters[1:3], df2_col = sample(month.abb, 3))
df3 <- data.frame(var1 = letters[1:3], df3_col = rnorm(3))
list_df <- list(df1, df2, df3)

do.call('merge', c(list_df[1:2], list(by = 'var1', all.x = TRUE)))
## var1 df1_col df2_col
## 1 a 3 May
```

```
## 2 b 1 Dec
## 3 c 2 Sep
NULL, NA, Inf, NaN
```

Ряд специальных значений или объектов используется в ситуациях, когда значения или объекта нет, значение недоступно или некорректно:

NA — not available. Ситуация, когда значение пропущено или не определено. Например, для какой-то строки в колонке таблицы нет значения или текстовое значение пытаются преобразовать в числовое или логическое. Следует учитывать, что NA — это в первую очередь логическое значение, маркирующее, что значение пропущено и будет ошибкой приравнять его к нулю, например, или к NULL. Когда известно, какого типа значения вектора, можно использовать специализированные вариации NA — NA_character_, NA_integer_, NA_real_ или NA_complex_.

NULL — пустой (null) объект с собственным одноимённым типом. Объект NULL не может быть использован одновременно с другими объектами, например, при создании последовательности, а присвоение какому-то элементу объекта значения NULL удаляет этот элемент. Чаще всего это встречается при работе со списками или таблицами, так как присвоение элементу значения NULL удаляет из списка ссылку на заданный объект.

NaN, Inf — not a number и infinite соответственно. Используются при работе с числовыми значениями и появляются при некорректных с математической точки зрения операциях — 0/0 и 1/0 соответственно. Значение Inf может быть как положительным, так и отрицательным.

Окружения

Окружения — это хеш-таблицы, то есть пары ключ-значение. Значениями обычно являются объявленные функции, данные, другие окружения. Окружения похожи на именованные списки, но сходство только внешнее, так как окружения, в отличие от списков, не имеют порядок объектов (соответственно, объекты окружения нельзя вызвать по их номеру, как в листе) и ансамбль окружений может быть представлен в виде дерева — практически у любого окружения есть родительское окружение.

За пределами функций использование окружений достаточно ограничено. Как правило, это либо ситуативные модификации функций пакетов, либо эзотерические практики организации объектов в интерактивном рабочем пространстве. Однако у окружений есть одна нечасто используемая характеристика: можно использовать окружения именно как хеш-таблицы, в которых операции поиска, вставки или удаления по ключам значений (названию) осуществляются в среднем быстрее, чем по обычным таблицам.

Например, создадим два объекта — таблицу, в которой в одной колонке будет вектор имён — текстовых значений, а во второй — числовое значение (0L). Второй объект — хешированное окружение с таким же количеством объектов, как строк в таблице. Каждый объект этого окружения имеет название, соответствующее значениям в первой колонке таблицы, а значения объектов — значениям из второй колонки таблицы (0L для всех объектов, как и для всех строк таблицы).

```
# создаём вектор уникальных "имён"
vector_letters <- paste(
  sample(letters, 10e4, replace = TRUE),
```

```

sample(10e4, replace = FALSE),
sep = ' _')

# создаём таблицу
simple_df <- data.frame(var1 = vector_letters, var2 = 0L)

# создаём хешированное окружение
hash_env <- new.env(hash = TRUE)
for (i in vector_letters) hash_env[[i]] <- 0L
ls(hash_env)[1:3]
## [1] "a_10012" "a_10031" "a_1004"
Как можно увидеть по бенчмаркам, поиск значения определённого объекта в
хешированном окружении существенно быстрее поиска соответствующих значений в
таблице:

item <- vector_letters[999]
microbenchmark::microbenchmark(
  df_search = simple_df[simple_df$var1 == item, 'var2'],
  hash_env_search = hash_env[[item]],
  times = 1000,
  unit = 'ms'
)
## Unit: milliseconds
##      expr   min    lq   mean  median    uq   max
##  df_search 1.137430 1.319790 1.664632644 1.4144280 1.578067 13.486095
## hash_env_search 0.000156 0.000264 0.001897494 0.0017455 0.003175 0.013404
## neval cld
## 1000 b
## 1000 a

```

Задания для выполнения

Задание 1. Установите и настройте среду R и RStudio. Создайте первый скрипт, который выводит на экран сообщение "Здравствуйтесь, R!".

- 1) Скачайте и установите R с официального сайта (<https://cran.r-project.org/>).
- 2) Скачайте и установите RStudio (<https://posit.co/download/rstudio/>).
- 3) Откройте RStudio, создайте новый скрипт (File → New File → R Script).
- 4) Введите команду `print("Здравствуйтесь, R!")` и выполните её (клавиша Ctrl+Enter).

Задание 2. Выполните задание из Таблицы 1 в соответствии с вариантом. Для выполнения задания используйте материал из методических указаний и, при необходимости, из дополнительной литературы.

Таблица 1. Варианты заданий.

Вариант	Задание
1	1) Создайте переменные различных типов (числовые, строковые, логические). Выведите их значения и типы с помощью функции <code>class()</code> . • Создайте переменную <code>a</code> со значением 10 (число). • Создайте переменную <code>b</code> со строкой "Привет".

	- Создайте переменную с со значением TRUE. - Выведите каждую переменную и её тип: <code>print(a)</code>; <code>print(class(a))</code>, и так далее.
2	2) Напишите скрипт, который создает график нескольких линий на одном поле (например, $y=x$ и $y=x^2$), используя базовые средства R или пакет <code>ggplot2</code>
3	1) Создайте вектор чисел от 1 до 10 с помощью функции <code>c()</code> и оператора <code>:</code> . Выполните арифметические операции с этим вектором. - Создайте вектор: <code>vec1 <- c(1:10)</code> или <code>vec2 <- 1:10</code>. - Вычислите сумму элементов: <code>sum(vec1)</code>. - Умножьте все элементы на 2: <code>vec1 * 2</code>. - Найдите квадрат каждого элемента: <code>vec1^2</code>. 2) Создайте серию случайных чисел с помощью функции <code>rnorm()</code> . Постройте гистограмму полученных данных и посчитайте их среднее и стандартное отклонение.
4	1) Создайте матрицу 3x3 из чисел 1-9. Выведите её и выполните транспонирование с помощью функции <code>t()</code> . - Используйте команду: <code>mat <- matrix(1:9, nrow=3, ncol=3)</code>. - Посмотрите матрицу: <code>print(mat)</code>. - Транспонируйте её: <code>t(mat)</code>. 2) Используя функцию <code>apply()</code> , найдите среднее значение по строкам и по столбцам матрицы, созданной в пункте 1.
5	1) Создайте датафрейм с данными о студентах: имена, возраст, оценки по предметам. Выведите его на экран и выполните выборку студентов старше 20 лет. - Используйте команду: <pre>students <- data.frame(name = c("Иван", "Мария", "Петр", "Анна"), age = c(19, 22, 20, 23), grade_math = c(4.5, 4.7, 4.0, 4.8), grade_history = c(4.0, 4.5, 3.8, 4.9)</pre> - Выведите весь датафрейм: <code>print(students)</code>. - Выберите студентов старше 20 лет 2) Создайте факторные переменные для категорий: "низкий", "средний", "высокий". Выполните их кодирование в числовой формат (например, <code>низкий=1</code>).
6	1) Используя встроенные функции, найдите сумму, среднее и медиану для набора чисел: 5, 10, 15, 20, 25. - Создайте вектор: <code>numbers <- c(5,10,15,20,25)</code> - Найдите сумму: <code>sum(numbers)</code> - Среднее значение: <code>mean(numbers)</code> - Медиана: <code>median(numbers)</code> 2) Используя пакет <code>ggplot2</code> , постройте точечную диаграмму (scatter plot) зависимости расхода топлива от веса автомобиля (<code>wt</code>).

7	1) Постройте график функции $y = x^2$ для x от -10 до 10 с помощью функции <code>plot()</code>. - Создайте последовательность x: <code>x <- -10:10</code> - Вычислите y: <code>y <- x^2</code> - Постройте график: <code>plot(x, y, type='l', main='График $y = x^2$', xlab='x', ylab='y')</code> 3) Напишите цикл <code>for</code>, который выводит квадраты чисел от 1 до 10.
8	1) Создайте функцию в R, которая принимает число и возвращает его квадрат. Проверьте работу функции на нескольких значениях. 2) Напишите скрипт для чтения данных из CSV-файла (создайте примерный файл вручную). В файле должны быть столбцы «Имя», «Возраст», «Оценка». Выведите первые строки файла после загрузки - Создайте файл <code>example.csv</code> со следующим содержимым: Имя;Возраст;Оценка Иван;19;4.5 Мария;22;4.7 Петр;20;4.0 Анна;23;4.8 - Загрузите его
9	1) Загрузите встроенный набор данных <code>mtcars</code>. Выполните описание данных (<code>summary()</code>, <code>str()</code>) и постройте гистограмму расхода топлива (<code>mpg</code>). 2) Используя функцию <code>ifelse()</code>, создайте новый столбец в датафрейме <code>mtcars</code>, который будет содержать "бензиновый" или "дизельный" в зависимости от значения переменной <code>am</code> (0 — бензин, 1 — дизель).

Литература

1. Язык программирования R: что делает его таким важным для анализа данных <https://practicum.yandex.ru/blog/chto-takoe-yazyk-r/>
2. Императивное (процедурное) программирование <https://0861.ru/paradigm19/lab-1.pdf>
3. Введение в язык программирования R <https://textbook.rintro.ru/index.html>

Практическое занятие 10 и 11

Тема: Процедурное программирование

Цель работы

Познакомиться с особенностями процедурного программирования. Решить задания в процедурном стиле.

Методические указания

Выполнение программы в процедурном стиле сводится к последовательному выполнению операторов с целью преобразования исходного состояния памяти, т.е. значений исходных данных, в заключительное, т.е. в результаты. Таким образом, с точки зрения программиста имеются программа и память, причем первая последовательно обновляет содержимое последней.

Процедурный стиль предоставляет возможность программисту определять каждый шаг в процессе решения задачи. Особенность таких языков программирования состоит в том, что задачи разбиваются на шаги и решаются шаг за шагом. Используя процедурный язык, программист определяет языковые конструкции для выполнения последовательности алгоритмических шагов.

Переменная состоит из имени и выделенной области памяти, которая соответствует ей. Для объявления или, другими словами, создания переменной используются директивы (ключевые слова, конструкции). В разных языках программирования создание переменных отличается.

Функция – это подпрограмма специального вида, которая может принимать на вход параметры, выполнять различные действия и передавать результаты работы. Вызов функции является, с точки зрения языка программирования, выражением, он может использоваться в других выражениях или в качестве правой части присваивания.

Процедура – это независимая именованная часть программы, которую после однократного описания можно многократно вызвать по имени из последующих частей программы для выполнения определенных действий.

Универсальные языки программирования подходят почти для любых задач, а специализированные — для конкретных нужд разработчиков: например, как R, язык для науки и анализа данных.

Создание переменных в языке R:

```
chr_var <- "Привет, Мир!" #Создание переменной строкового типа (character)
num_var <- 24.3 #Создание переменной вещественного типа (numeric)
int_var <- 124L #Создание переменной целочисленного типа (integer)
cplx_var <- 56 + 4i #Создание переменной комплексного типа (complex)
logi_var <- TRUE #Создание переменной логического типа (logical)
assign("name_var", "Какое-то значение") #Создание переменной с помощью assign()
```

Функции. Для создания функций используются служебные слова: `func`, `def`, `function` и др. Синтаксис создания функций отличается в разных языках программирования.

Создание и использование функций в языке R:

```
#Создание функции, возвращающей большее значение
```

```

bigger <- function(a, b) {
  if (a > b) {
    return(a)
  } else {
    return(b)
  }
}
bigger(5, 8) #Корректное использование функции
bigger(5) #Не корректное использование функции
bigger(7, 6, 9) #Не корректное использование функции

```

Написание программ

С развитием языков программирования развивались и технологии, используемые при написании программного кода. Первые программы писались сплошным текстом. Это была простая последовательность команд. Все это выглядело следующим образом:

Начало программы

Команда 1

Команда 2

...

Команда N

Конец программы

Используя такой подход к программированию, можно было сделать очень мало. Единственное, что было доступно программисту для создания логики в данном случае – это переходы. Под переходом понимается переход на какую-то команду при определенных условиях, которые сложились в процессе обработки данных на процессоре.

Образец части программы, построенной с использованием безусловного перехода:

Начало программы

Команда 1

Команда 2

Если выполнено условие, то вернуться к команде 1

Команда 3

...

Команда N

Конец программы

Безусловный переход (*goto, jmp*).

Как правило, оператор безусловного перехода состоит из двух частей: собственно оператора и метки, указывающей целевую точку перехода в программе (например: *goto метка*). Метка, в зависимости от правил языка, может быть либо числом (как, например, в классическом BASIC), либо идентификатором используемого языка программирования. Для меток-идентификаторов метка, как правило, ставится перед оператором, на который должен осуществляться переход, и отделяется от него двоеточием (метка:).

Действие оператора перехода состоит в том, что после его исполнения следующими будут исполняться операторы программы, идущие в тексте непосредственно после метки (до следующего оператора перехода, ветвления или цикла). Для машинных языков инструкция перехода копирует в регистр процессора, содержащий адрес следующей выполняемой команды, адрес команды, помеченной меткой.

Пример программ, печатающих числа от 1 до 100 и квадраты этих чисел, выполненные в разных стилях на языке программирования BASIC:

Процедурный стиль	Структурное программирование
1 i=0	1 FOR i=1 TO 100

2 i+=1 3 PRINT i; "squared="; i*i 4 IF i>100 THEN GOTO 6 5 GOTO 2 6 PRINT "Program Completed." 7 END	2 PRINT i; "squared="; i*i 3 NEXT i 4 PRINT "Program Completed." 5 END
---	---

В процедурном подходе какой-то код программы мог объединяться в отдельные блоки (процедуры). После этого такой блок команд можно вызывать из любой части программы. Структура программы, выполненной в процедурном стиле:

начало процедуры 1

команда 1

команда 2

конец процедуры 1

начало программы

команда 1

команда 2

если выполнено условие, то выполнить код процедуры 1.

команда 3

конец программы.

Задания для выполнения

Задание 1. Написать программу, выполненную в процедурном стиле. Программа должна быть выполнена в виде псевдокода, в виде блок-схемы и на языке высокого уровня (ЯВУ) (здесь и далее, если не оговорено иное, при отсылке к ЯВУ необходимо выполнять код на языке R).

- Напишите программу, рассчитывающую площадь трех фигур: квадрат, прямоугольник и круг. На входе программа запрашивает введение данных о фигурах (для квадрата – сторона, круг – радиус, прямоугольник – две стороны). На выходе программа указывает площади трех фигур и общую площадь.
- Напишите программу, рассчитывающую сумму расходов за месяц. На входе программа запрашивает сведения о расходах по нескольким пунктам (минимум 3 статьи расходов). На выходе программа указывает суммарные расходы и максимальную статью расходов.
- Напишите программу, подсчитывающую среднее количество занятий в неделю. Программа запрашивает информацию о количестве занятий в день (по дням недели). На выходе программа указывает среднее количество занятий в неделю с округлением до ближайшего целого числа.

Задание 2. Опишите, представленный код в виде псевдокода и ответьте на вопрос, что будет получено при передаче функции числа 7? Также реализуйте данный алгоритм на ЯВУ.

Функция на ассемблере с синтаксисом AT&T:

foo:

cmp \$0, %edi

jg calc

mov \$1, %eax

jmp exit

calc:

```
push %edi
sub $1, %edi
call foo
pop %edi
imul %edi, %eax
exit:
ret
```

Это функция с одним входным параметром, для которой ABI (двоичный интерфейс приложений) предписывает передачу одного параметра через регистр %edi, а передачу возвращаемого значения через регистр %eax.

Замечания:

- 1) Команда 'imul src, dest' умножает src на dest и кладет результат в dest.
- 2) Не нужно думать о переполнении. Его здесь не будет.

Литература

4. Язык программирования R: что делает его таким важным для анализа данных <https://practicum.yandex.ru/blog/chto-takoe-yazyk-r/>
5. Императивное (процедурное) программирование <https://0861.ru/paradigm19/lab-1.pdf>
6. Введение в язык программирования R <https://textbook.rintro.ru/index.html>

Практическое занятие 12 и 13

Тема: Функциональное программирование. Описание и вызов функций в языке muLISP.

Цель работы

Целью работы является изучение базовых функций организации и обработки списков, а также способов описания и вызова нерекурсивных функций в языке программирования Лисп.

Методические указания

Работа с интерпретатором Microsoft muLISP.

Исполняемым файлом интерпретатора Лиспа является mulisp.com (или mulisp_2.com - для русифицированной версии). Русифицированную версию интерпретатора рекомендуется использовать в комплекте с программой-русификатором для MS DOS, например, keyrus.com или rk.com.

Для работы с интерпретатором необходимо после запуска mulisp.com и появления приглашения \$ набирать либо описание функции, либо ее вызов.

Примеры :

\$(* 4 5)- вызов функции умножения для двух аргументов;

\$(* 5 6 7 8)- тоже для четырех аргументов;

\$(- 12 3)- вызов функции вычитания 12-3

Для выхода из среды Microsoft muLISP необходимо набрать в командной строке интерпретатора :

\$(SYSTEM).

Рекомендуемая последовательность действий при работе с интерпретатором.

- разработать текст программы;
- записать разработанный текст в файл с расширением .lsp;
- вызвать интерпретатор с указанием выполняемого файла.

Для сокращения объема программы в первой лабораторной работе разрешается использовать псевдофункцию, связывающую имена и значения (SETQ имя значение). К примеру, нужно выполнять многократную обработку списка '(a c d (e f)). Обозначим этот список l1 и сделаем это так (SETQ l1 '(a c d (e f))). После такого связывания там, где используется значение списка, можно использовать имя l1.

Рекомендуемая структура программы.

- связывание имен и значений;
- вызовы функций, либо описание функций, т.е. собственно текст программы;
- функция переключения входного потока (RDS).

Базовые функции Лиспа.

Примитивные (простейшие) базовые функции можно сравнить с основными арифметическими действиями: они просты и их мало.

Назначение этих функций состоит в:

- расчленении S-выражений (функции-селекторы);
- составлении S-выражений (функции-конструкторы);

– анализ S-выражений (функции-предикаты).

Таблица 1. Примитивные базовые функции.

Назначение	Вызов	Результат
Селектор	(CAR список)	S-выражение
Селектор	(CDR список)	список
Конструктор	(CONS S-выраж.,список)	список
Конструктор	(ATOM S-выражение)	T, либо NIL
Предикат	(EQ атом атом)	T, либо NIL

Функция CAR.

Значением функции является голова списка-аргумента.

Примеры :

\$ (CAR '(P Q R))-результат P;

\$ (CAR '(A B))- результат A;

\$ (CAR '(DOG CAT))-результат DOG;

\$ (CAR '((A B) C))-результат (A B);

\$ (CAR (A B))-результат не определен, т.к. (A B) интерпретатором воспринимается как вызов функции A с аргументом B.

Функция CDR.

Значением функции является хвост списка-аргумента.

Примеры :

\$ (CDR '(P Q R))-результат (Q R);

\$ (CDR '(B))-результат NIL.

В muLISP наряду с простейшими базовыми функциями-селекторами CAR и CDR определены более сложные функции, выделяющие любой из десяти первых элементов :

FIRST - выделяет первый элемент списка, SECOND- второй и т.д.

Функция CONS.

Функция CONS имеет два аргумента - первый S-выражение, вторым аргументом обязательно должен быть список. Назначение функции CONS конструирование нового списка. Новый список получаем путем добавления первого аргумента к списку- второму аргументу.

Для получения списка, содержащего один элемент, нужно в качестве второго списка указать пустой список NIL.

Примеры :

\$ (CONS 'A '(B C))-результат есть список (A B C);

\$ (CONS '(FIRST SECOND)'(THIRD FOURTH FIFTH)) – результат есть ((FIRST SECOND) THIRD FOURTH FIFTH);

\$ (CONS NIL NIL) – результат есть одноэлементный список (NIL);

\$ (CONS 12 NIL) - результат есть список (12);

\$ (CONS '(A B C) NIL) – результат есть список ((A B C)).

Связь между функциями CAR,CDR,CONS.

Список, разделенный с помощью CAR и CDR на голову и хвост, можно объединить с помощью функции-конструктора CONS.

Пример.

\$ (CAR '(FIRST SECOND THIRD)) – результатом будет FIRST;

\$ (CDR '(FIRST SECOND THIRD)) – результатом будет список (SECOND THIRD);

\$ (CONS (CAR '(FIRST SECOND THIRD)) (CDR '(FIRST SECOND THIRD))) – результатом будет исходный список (FIRST SECOND THIRD).

Предикат ATOM.

Предикат- это логическая функция. Предикат ATOM возвращает значение Т (эквивалент TRUE), если его аргументом является атом и NIL(эквивалент FALSE), иначе.

Примеры использования ATOM :

\$ (ATOM 'X)-результатом является Т, т.к. аргумент – атом;

\$ (ATOM '(1 2 3))-результатом является NIL, т.к. аргумент – список;

\$ (ATOM (CDR '(1 2 3))) – результат NIL;

\$ (ATOM (CAR '(A B C))) – результат Т.

Предикат EQUAL.

Предикат EQUAL проверяет эквивалентность любых S-выражений.

Примеры :

\$ (EQUAL '(3 3) (4 3)) – результат есть NIL, так как списки не совпадают;

\$ (EQUAL 'Y 'Y) – результат есть Т, так как атомы совпадают;

\$ (EQL 5.0 5.0) – результат есть Т.

Расширение базовых функций в muLISPe.

Кроме базовых функций в muLISPe есть целый набор встроенных функций, выполняющих преобразования S-выражений и дополняющих базовые функции.

Вложенные вызовы функций CAR и CDR.

Путем использования функций CAR и CDR можно выделить любой элемент списка.

Например, для выделения второго элемента второго подсписка нужно записать :

\$ (CAR (CDR (CAR (CDR '((A B C) (D E) (F H))))))

В muLISP для таких комбинаций CDR и CAR можно использовать более короткую запись в виде одного вызова функции : (C...R список), где вместо многоточия записывается нужная комбинация букв A (CDR) и D (CDR). Но максимум – 4 буквы !

Например, записанная выше конструкция может быть представлена как (CADADR '((A B C) (D E) (F H))).

Для выделения из списка какого-либо элемента : первого, второго, ..., десятого можно использовать специальные функции muLISP : FIRST, SECOND, THIRD, FOURTH, FIFTH, SEVENTH, EIGHTH, NINTH, TENTH. Все эти функции имеют один аргумент – список.

Более того, в muLISP существуют две дополнительные функции для выделения элемента из списка :

– (NTH n список) – выделяет n-ый элемент из списка – второго аргумента функции NTH;

– (LAST список) – возвращает список из одного последнего элемента списка – аргумента.

Примеры использования LAST и NTH :

\$ (NTH 4 '(F D C C F H)) – выделяет элемент F, т.к. номера элементов начинается с нуля;

\$ (LAST '(A B C D E F)) – возвращает список (F).

Дополнительные предикаты сравнения аргументов.

Предикат EQ сравнивает два атома и принимает значение Т, если атомы идентичны, NIL – в противном случае. EQ сравнивает только атомы и константы Т, NIL.

Примеры :

\$ (EQ 'X 'Y) – атомы неэквивалентны, значение функции равно NIL;

\$ (EQ () NIL) – значением функции будет T;

\$ (EQ T (ATOM 'CAT)) – значением функции будет T.

Предикат "=" применяется для сравнения чисел различных типов.

Предикат принимает значение T, если значение чисел совпадают вне зависимости от типа, иначе NIL.

Примеры :

\$ (= 3 3) – результат T;

\$ (= 3 7) - результат NIL.

Предикат NUMBERP имеет один аргумент, который является S выражением. Предикат принимает значение T, если аргумент является числом любого типа, и NIL – в любом другом случае.

Примеры :

\$ (NUMBERP 3.066) – значением является T;

\$ (NUMBERP T) – значением является NIL, т.к. T – не число.

Предикат NULL имеет один аргумент-список. Если список пустой, то NULL принимает значение T, иначе NIL.

Функция LIST. Эта функция может иметь любое количество аргументов. Значением функции является список из заданных аргументов.

Примеры :

\$ (LIST '1 '2) – результатом является список (1 2);

\$ (LIST 'A 'B 'C 'D (+ 3 4)) – результатом является список (A B C D 7).

Объявление функций.

В функциональных языках программирования различают описания именованных и неименованных функций. Для описания неименованных функций в LISPe используются две конструкции: LAMBDA и NLAMBDA.

Неименованные функции.

Неименованные функции используются для выполнения однократных преобразований или вычислений, такие действия носят локальный характер и используются только одной функцией. Наиболее часто неименованные функции используются в функционалах и макросах.

Для описания неименованных функций используются lambda-вызовы, имеющие следующий вид :

((lambda (<список формальных параметров>)

< тело функции>)

< список фактических параметров>)

Пример. Для функции f1(x,y), которая возвращает в качестве результата y в случае атомарного x и список '(x,y) – в противном случае, примером реализации в виде lambda-вызова может послужить :

((lambda (x y)

((atom (car x)) y)

(cons x y))

(car '((f) g h)) '(r t y))

Описание именованных функций.

Для объявления именованных функций используются конструкции :

(DEFUN <имя функции> (список формальных параметров)

<lambda -вызов>)

(DEFUN <имя функции> (список формальных параметров)

<тело функции>)

DEFUN - служебное слово (DEfine FUNction - определение функции), с которого обязательно должно начинаться описание именованной функции. Имя функции может быть любым символьным атомом, рекомендуется давать такие имена функциям, которые отражают смысловое содержание функции. Список формальных параметров представляет собой список символьных атомов, записанных через пробел. Тип параметров не указывается. Тело функции может содержать либо ветвления, либо представлять собой суперпозицию вызовов функций.

Результат, полученный при выполнении функций, связывается с именем функции.

Для примера рассмотрим описание и вызов функции, которая из списка выделяет второй и четвертый элементы и конструирует из них список.

Задачу конструирования нового списка из элементов старого можно решить различными способами, используя базовые функции CONS, CDR, CAR и функции muLISP: SECOND, FOURTH, LIST.

Описание функции непосредственно в интерпретаторе будет выглядеть следующим образом :

```
$ (DEFUN SFLIST (LST) (CONS (SECOND LST) (CONS (FOURTH LST) NIL)))
```

При использовании lambda-вызова описание имеет вид :

```
$ (DEFUN SFL (LST)
  ((lambda (X Y)
    (CONS X(CONS Y NIL)))
   (SECOND LST)(FOURTH LST)))
```

Эта же функция может быть описана с использованием других функций muLISP: LIST, CADR, CADDDR :

```
$ (DEFUN SFILST1 (LST)
  (LIST (CADR LST)(CADDDR LST)))
```

Вызов функции.

Вызов функции записывается следующим образом :

(имя функции <список фактических параметров>)

Например, вызов функции SFLIST :

```
$ (SFLIST '(DOG CAT COW PIG))
```

дает в качестве результата список (CAT PIG).

Задание для выполнения

1. Описать неименованную функцию для объединения голов трех списков в один список, исходные данные взять из Таблицы 2.
2. Описать именованную функцию для создания нового списка из элементов нескольких исходных списков. В качестве исходных списков использовать списки Таблицы 2. Номера элементов списков взять в Таблице 3.
3. Описать именованную функцию в соответствии с вариантом индивидуального задания в Таблице 4.

Таблица 2. Исходные списки.

Вариант т	Исходные списки		
	1	2	3
1	(Y U I)	(G1 G2 G3)	(KK LL MM JJJ)
2	(G55 G66 G777)	(9 (F G) I)	(N I L T D J (II JJ))
3	((PI) V (H J K))	(R YU (H KJ KL)	(U II OO LL PP (3 4 5))
4	(T (U U1 U2) (U4 U6 U8))	(4 6 (7 8 9))	(78 89 90 67 45)
5	(9 (I) 8 88 888))	(H (J K L) (UJN))	(C B (N M I) (T Y U))
6	(T Y D E F (NL KM LM) JL)	(+ 2 3)	(* (+ 6 8) (- 70 8))
7	(TYPE PRINT DEL)	(H (H J O) (UJ N))	(READ SAVE LOAD (TXT))
8	(GOAL FUNCTOR CLAUSE (DATA BASE))	(2 5(5 4 6)8)	(L (K (K I)U))
9	(DOG (CAT) FOX (I))	(RET GET PUT OUT IN)	(MOV ADD (MUL DEV))
10	(FIR SED (1 2 3) (5) (I))	(H J U K (L M N) (D E L))	(4 5(6 7))

Таблица 3. Номера элементов.

Вариант	Список 1	Список 2	Список 3
1	2	2	3
2	3	2	6
3	1	3	6
4	2	3	4
5	2	3	3
6	6	2	2
7	3	2	3
8	4	3	2
9	3	4	3
10	4	4	3

Таблица 4. Варианты индивидуального задания.

Вариант	Задание
1	Есть два списка. Если первый элемент списка есть натуральное число, то вернуть второй список, иначе вернуть список из головы второго и хвоста первого.
2	Написать функцию, которая возвращает квадратный корень из аргумента, если аргумент является числом, последний элемент аргумента-списка, если аргумент – список, аргумент – иначе.
3	Есть пять чисел. Написать функцию, формирующую список из максимального и минимального по модулю чисел, если минимальное и максимальное числа – целые, среднее арифметическое минимального и максимального чисел - иначе.

4	Написать функцию, которая для аргумента-числа проверяет, является ли оно степенью двойки.
5	Написать функцию, которая для заданных координат X1,Y1 и X2,Y2 возвращает расстояние между ними. Координаты могут иметь отрицательное значение.
6	Написать функцию, которая по заданному вещественному числу формирует список из трех элементов. Первый элемент – знак числа, второй – модуль числа, третий – ближайшее к нему целое число.
7	Написать функцию, которая для трех аргументов-чисел проверяет, является ли третье число результатом возведения в степень первого числа с показателем, равным второму числу.
8	Есть список. Найти сумму первого, третьего и седьмого элементов списка, если указанные элементы – числа, вернуть последний элемент списка – иначе.
9	Написать функцию вычисления дискриминанта квадратного уравнения.
10	Написать функцию, которая для аргумента-списка формирует список-результат по правилу : если первый и последний элементы списка-аргумента есть четные натуральные числа, то включить в список-результат первым элементом – квадрат последнего элемента исходного списка, вторым – четвертую степень первого; в противном случае сформировать список из первого и последнего элементов.

Литература

1. Функциональное и логическое программирование. Лабораторный практикум. Часть 1. Функциональное программирование / Авт.-сост. Д. В. Михайлов, Г.М. Емельянов; НовГУ им. Ярослава Мудрого.– Великий Новгород, 2007. 80 с.
http://www.machinelearning.ru/wiki/images/0/0b/Method_fp.pdf

Практическое занятие 14

Тема: Функциональное программирование. Описание простейших рекурсивных функций в языке muLISP.

Цель работы

Целью работы является изучение основных правил написания рекурсивных функций в функциональном языке.

Методические указания

Функция называется **рекурсивной**, если в ее теле содержится вызов самой этой функции.

Принято различать рекурсию **по значению** и рекурсию **по аргументам**.

В случае рекурсии по значению сам рекурсивный вызов непосредственно вычисляет значение функции. Пример - принадлежность объекта списку.

В случае рекурсии по аргументам в качестве значения функции возвращается значение некоторой другой функции и рекурсивный вызов участвует в вычислении аргументов этой функции. Пример – нахождение суммы элементов списка.

Рекурсия называется простой, если вызов функции встречается в некоторой ветви лишь один раз. Простой рекурсии в процедурном программировании соответствует обыкновенный цикл.

В ЛИСПе рекурсия - основной метод программирования. Как и в других языках программирования, описание рекурсивных функций требует особой тщательности. Во-первых, функция обязательно должна содержать хотя бы одно условие окончания рекурсии, во-вторых, значение

аргумента при рекурсивных вызовах должно изменяться.

Часто условием окончания рекурсии является пустой список аргумент. В процессе работы все элементы исходного списка обрабатываются последовательно до тех пор, пока не будет достигнут конец списка, т.е. пока список не пуст. Указанное типовое условие окончания рекурсивной обработки можно записать как :

((NULL lst) <действие>)

Второе требование к рекурсивной функции, касающееся изменения аргумента при рекурсивных вызовах, чаще всего обеспечивается рекурсивным вызовом с аргументом-хвостом списка.

Можно выделить следующие (основные) шаги при построении рекурсивной функции :

- 1) Определить количество и вид аргументов.
- 2) Определить характер результата.
- 3) Задать как минимум одно условие окончания рекурсии.
- 4) Определить формирование результата вызова функции.
- 5) Описать формирование новых значений аргументов для рекурсивного вызова.

Рассмотрим примеры описания рекурсивных функций.

Пример 1. Описать функцию, которая проверяла бы поэлементную эквивалентность двух списков.

Из задачи ясно, что формальных параметров у функции должно быть два и оба – списки. Назовем параметры lst1 и lst2.

Функция должна возвращать значение Т, если длина списков одинакова и совпадают все элементы: первые, вторые, и т. д.

Определим, сколько и каких условий окончания рекурсии должно присутствовать в функции. Благополучный исход, при котором функция возвращает Т, достигается при поэлементном совпадении lst1 и lst2. При этом должно выполняться условие :

```
(and (null lst1)(null lst2))
```

Два условия завершения рекурсивного вызова надо предусмотреть для неблагоприятного исхода, когда функция возвращает NIL. Это произойдет, во-первых, если один список короче другого:

```
(or (and (null lst1)(not (null lst2)))
```

```
    (and (not (null lst1))(null lst2)))
```

И, во-вторых, если найдены несовпадающие элементы :

```
(NOT (EQUAL (CAR lst1)(CAR lst2)))
```

Если ни одно из условий окончания рекурсии не выполняется, то функция вызывает себя же уже для хвостов списка.

Описание функции на muLISPe :

```
(defun eqlists (lst1 lst2)
```

```
  ((and (null lst1)(null lst2)) T)
```

```
  ((or (and (null lst1)(not (null lst2)))
```

```
      (and (not (null lst1))(null lst2))
```

```
      (not (equal (car lst1)(car lst2)))) nil)
```

```
  (eqlists (cdr lst1)(cdr lst2)))
```

Пример 2. Суммирование элементов списка.

Количество аргументов : 1 – список.

Результат : число.

Условие окончания рекурсии – пустой список :

```
((null list) 0).
```

По достижению этого условия в качестве значения функции возвращается 0.

Формирование значения функции :

```
(+ (car list) сумма хвоста)
```

Сумма хвоста есть результат рекурсивного вызова проектируемой функции с хвостом списка в качестве аргумента.

Описание функции :

```
(defun sum (list)
```

```
  ((null list) 0)
```

```
  (+ (car list)(sum (cdr list))))
```

Результат вызова (sum '(9 7 5 6 4)) есть 31.

Пример 3. Принадлежность объекта заданному списку.

Количество аргументов : 2 – произвольное s-выражение (obj) и список (lst).

Результат : значение Т (истина), либо NIL (ложь).

Условий окончания рекурсии два :

```
((null lst) nil) - пустой список, искомый объект не найден,
```

```
((equal obj (car lst)) T) - объект найден.
```

В случае несовпадения искомого объекта и головы списка результирующее значение вычисляется путем вызова проектируемой функции с хвостом списка в качестве второго аргумента. Полное описание функции:

```
(defun member (obj lst)
```

`((null lst) nil)`

`((equal obj (car lst)) T)`

`(member obj (cdr lst)))`

Результатом вызова `(member 7 '(1 2 3 4))` есть `nil`.

Результатом вызова `(member 3 '(1 2 3 4))` есть `T`.

Пример 4. Объединение списков.

Рассмотри три варианта решения данной задачи.

Вариант 1.

Количество аргументов : 2 – списки `lst1` и `lst2`.

Результат : список.

Условий окончания рекурсии три :

`((and (null lst1)(null lst2)) nil)` - оба списка пустые,

`((null lst1) lst2)` - первый список пустой,

`((null lst2) lst1)` - второй список пустой.

Значение функции вычисляется путем присоединения головы первого списка в качестве головы к результату вызова проектируемой функции с хвостом первого списка в качестве первого аргумента и вторым списком в качестве второго аргумента.

Полное описание первого варианта функции объединения списков:

`(defun append (lst1 lst2)`

`((and (null lst1)(null lst2)) nil)`

`((null lst1) lst2)`

`((null lst2) lst1)`

`(cons (car lst1)(append (cdr lst1) lst2)))`

Результат вызова `(append '(1 2 3 4) '(5 6 7 8))` есть список `(1 2 3 4 5 6 7 8)`.

Рассмотрим варианты более компактной записи условия окончания рекурсии.

Вариант 2.

Количество аргументов : 2 – списки `lst1` и `lst2`.

Результат : список.

Условий окончания рекурсии два:

`((null lst1) lst2)` - первый список пустой,

`((null lst2) lst1)` - второй список пустой.

Результат функции формируется аналогично варианту 1.

Полное описание функции :

`(defun append (lst1 lst2)`

`((null lst1) lst2)`

`((null lst2) lst1)`

`(cons (car lst1)(append (cdr lst1) lst2)))`

Вариант 3.

Количество аргументов : 2 - списки `lst1` и `lst2`.

Результат : список.

Условие окончания рекурсии :

`((null lst1) lst2).`

Результат функции формируется аналогично варианту 1.

Полное описание функции :

`(defun append (lst1 lst2)`

```
((null lst1) lst2)
(cons (car lst1)(append (cdr lst1) lst2)))
```

Задание для выполнения

1. Выполнить Примеры 1-4.
2. Описать функцию в соответствии со своим вариантом задания из Таблицы 1.

Таблица 1. Варианты заданий.

Вариант	Задание
1	Описать функцию, которая для заданного списка lst формирует список-результат путем объединения результата реверсирования lst, результата реверсирования хвоста lst, результата реверсирования хвоста хвоста lst и так далее. Пример : для списка '(1 2 3 4 5 6) результатом будет : '(6 5 4 3 2 1 6 5 4 3 2 6 5 4 3 6 5 4 6 5 6).
2	Есть список lst и два произвольных лисповских объекта obj1 и obj2. Описать функцию, которая в списке lst заменяет все вхождения объекта obj1 объектом obj2.
3	Описать функцию, которая находила бы сумму всех числовых элементов списка с учетом наличия подсписков. Пример: для списка '(1 ((2 3) 4) 5 6) результатом будет 21.
4	Описать функцию, которая на основе списка чисел формирует список-результат следующим образом: первый элемент есть произведение элементов списка, второй – произведение элементов хвоста, третий – произведение элементов хвоста хвоста и так далее. Пример: для списка '(1 2 3 4 5 6) результатом будет : '(720 720 360 120 30 6).
5	Реализовать функцию включения объекта на заданное место в списке (нумерация элементов – от начала списка).
6	Реализовать функцию, которая в исходном списке заменяет все элементы-списки результатами их реверсирования. Реверсирование производить на всех уровнях вложения. Пример : для списка '(1 ((2 3) 4) 5 6) результатом будет : '(1 (4 (3 2)) 5 6).
7	Реализовать функцию, которая выдавала бы элемент списка по заданному номеру с конца.
8	Дан список lst и число n. Реализовать функцию, которая удаляет все i+n – е элементы списка.
9	Даны списки lst1 и lst2. Реализовать функцию, которая удаляет из lst1 все элементы-списки, которые соответствуют тому же множеству, что и lst2. Пример : для списков : lst1='(1 (2 2 3) 4 (3 2 3) 5), lst2='(3 2 3 2) результатом будет '(1 4 5).
10	Реализовать функцию, возвращающую Т при идентичности порядка расположения одинаковых атомов в исходных списках.

Литература

1. Функциональное и логическое программирование. Лабораторный практикум. Часть 1. Функциональное программирование / Авт.-сост. Д. В. Михайлов, Г.М. Емельянов;

НовГУ им. Ярослава Мудрого.– Великий Новгород, 2007. 80 с.
http://www.machinelearning.ru/wiki/images/0/0b/Method_fp.pdf

Практическое занятие 15 и 16

Тема: Функциональное программирование. Методы разработки функциональных программ.

Цель работы

Целью работы является изучение основных методов разработки функциональных программ с позиций Строго Функционального Языка.

Методические указания

Использование вспомогательных функций и применение накапливающих параметров являются основными методами разработки функциональных программ. Понятия основной и вспомогательной функции являются относительными. Одна и та же функция может использоваться для вычисления своего значения другие функции из числа описанных в программе, но в то же время и ее могут вызывать как вспомогательную.

Нисходящее и восходящее проектирование.

Различают нисходящее и восходящее проектирование функциональных программ. Основным отличием нисходящего проектирования является решение задачи с использованием разработанных ранее функций в качестве вспомогательных.

Пример 1. Использование нисходящего проектирования для задачи преобразования списка в множество.

Рассмотрим вначале взаимные отличия списков и множеств.

Список :

- Упорядоченная последовательность : $(1\ 2\ 3) \neq (3\ 2\ 1)$;
- Один и тот же элемент может встречаться дважды.

Множество :

- Наличие отношения порядка не является обязательным;
- Каждый элемент встречается ровно один раз.

Аргумент : список.

Результат : множество.

Условие выхода из рекурсии : $((\text{null lst}) \text{ nil})$.

Вариант 1.

Генерация результата :

Включить (car lst) в множество, полученное из (cdr lst) , из которого удалены все вхождения (car lst) .

Предположим, что мы имеем функцию удаления всех вхождений объекта в список :

```
(defun delete (obj list)
  ((null list) nil)
  ((equal obj (car list))(delete obj (cdr list)))
  (cons (car list)(delete obj (cdr list))))
```

Будем использовать функцию delete для удаления из исходного списка вхождений повторяющихся элементов:

```
(defun list_set1 (list)
  ((null list) nil)
  (cons (car list)(list_set1 (delete (car list)(cdr list)))))
```

Вариант 2.

Генерация результата:

Если (car list) содержится в (cdr list), то вернуть в качестве результата множество, полученное из (cdr list). Иначе включить (car list) в множество, полученное из (cdr list). Для определения принадлежности объекта списку будем использовать разработанную нами ранее функцию member :

```
(defun member (obj list)
  ((null list) nil)
  ((equal obj (car list)) T)
  (member obj (cdr list)))
(defun list_set2 (list)
  ((null list) nil)
  ((member (car list)(cdr list))(list_set2 (cdr list)))
  (cons (car list)(list_set2 (cdr list))))
```

Использование накапливающих параметров.

Пример 2. Функция реверсирования списка.

Аргумент: список lst.

Результат: тот же список, переписанный в обратном порядке.

Условие окончания рекурсии : пустой список.

Генерация результата : (append (reverse (cdr lst))(cons (car lst) nil)).

Полное описание функции :

```
(defun reverse (lst)
  ((null lst) nil)
  (append (reverse (cdr lst))(cons (car lst) nil)))
```

Рассмотрим возможность снижения вычислительной сложности задачи путем уменьшения числа вызовов простейших базовых функций. Функция выполняется тем эффективнее, чем меньше число вызовов cons она предполагает. Исследуем зависимость числа N вызовов функции cons функцией reverse от числа n элементов реверсируемого списка.

Пусть дан список '(1 2 3 4 5). Вызываем (reverse '(1 2 3 4 5))

Шаг 1. 1-й аргумент append : '(2 3 4 5). Результат вызова cons : '(1) - 2-й аргумент append.

Шаг 2. 1-й аргумент append : '(3 4 5). Результат вызова cons : '(2) - 2-й аргумент append.

Шаг 3. 1-й аргумент append : '(4 5). Результат вызова cons : '(3) - 2-й аргумент append.

Шаг 4. 1-й аргумент append : '(5). Результат вызова cons : '(4) - 2-й аргумент append.

Шаг 5. 1-й аргумент append : '(). Результат вызова cons : '(5) - 2-й аргумент append.

Достигнуто условие окончания рекурсии (пустой список).

Итого на этапе развертывания рекурсии получилось n, то есть 5 вызовов cons. Рассмотрим теперь свертывание рекурсии.

Посредством функции append конструируем список-результат из извлекаемых из стека результатов рекурсивных вызовов :

(append nil '(5)) → '(5) – 0 вызовов cons.

(append '(5) '(4)) → '(5 4) – 1 вызов cons.

(append '(5 4) '(3)) → '(5 4 3) – 2 вызова cons.

(append '(5 4 3) '(2)) → '(5 4 3 2) – 3 вызова cons.

(append '(5 4 3 2) '(1)) → '(5 4 3 2 1) – 4 вызова cons.

Число вызовов cons на этапе свертывания рекурсии соответствует сумме $n - 1$ первых членов арифметической прогрессии:

$$\frac{(1+(n-1)) \cdot (n-1)}{2} = \frac{n \cdot (n-1)}{2}.$$

Общее число вызовов cons при работе функции reverse составляет

$$N = n + \frac{n \cdot (n-1)}{2} = \frac{n \cdot (n+1)}{2}.$$

Рассмотрим теперь вариант функции реверсирования, который предполагает $N = n$ вызовов cons. Данный вариант функции реверсирования использует вспомогательную функцию с накапливающим параметром, который передается в качестве результата в основную функцию по завершению рекурсии.

```
(defun reverse (lst)
  (rev lst nil))
(defun rev (lst1 lst2)
  ((null lst1) lst2)
  (rev (cdr lst1) (cons (car lst1) lst2)))
```

lst2 – накапливающий параметр.

Функция rev работает следующим образом :

lst1	lst2
(1 2 3 4)	()
(2 3 4)	(1)
(3 4)	(2 1)
(4)	(3 2 1)
()	(4 3 2 1) Возвращается в качестве результата

Задание для выполнения

ВНИМАНИЕ! В лабораторной работе можно использовать только средства строго функционального языка программирования. Иными словами, при описании функций нельзя использовать функции SET, SETQ, SETF и циклы. Все функции должны быть разработаны самостоятельно.

1. Написать программу сортировки списка методом Шелла. Вычисление последовательности шагов сортировки производится в соответствии с вариантом в Таблице 1.

Таблица 1. Вычисление шага сортировки Шелла.

Вариант	Задание
1	Методом Р. Седжвика. Длина очередного S -го шага: $inc[s] = \begin{cases} 9 \cdot 2^s - 9 \cdot 2^{\frac{s}{2}} + 1, & \text{если } s - \text{четно;} \\ 8 \cdot 2^s - 6 \cdot 2^{\frac{s+1}{2}} + 1, & \text{если } s - \text{нечетно.} \end{cases}$
2	Методом, предложенным Дональдом Кнутом: $h_{k-1} = 3 \cdot h_k + 1$, $h_t = 1$, где h_i – шаг сортировки, $t = [n] - 1$ – число шагов сортировки, n – длина списка.

Методом, предложенным Дональдом Кнутом: $h_{k-1} = 2 \cdot h_k + 1$, $h_t = 1$, где h_i – шаг сортировки, $t = \lceil n \rceil - 1$ – число шагов сортировки, n – длина списка.

2. Написать программу сортировки списка в соответствии с вариантом:

- 1) Сортировка простыми включениями.
- 2) Сортировка бинарными включениями.
- 3) Сортировка методом прямого выбора.
- 4) Сортировка методом пузырька.
- 5) Шейкер-сортировка.
- 6) Сортировка Хоара.

Сравнить эффективность реализованной сортировки и реализованного в Задании 1 варианта сортировки Шелла.

3. Написать программу объединения двух отсортированных списков в один. При этом порядок сортировки в списке-результате должен сохраняться.

Литература

1. Функциональное и логическое программирование. Лабораторный практикум. Часть 1. Функциональное программирование / Авт.-сост. Д. В. Михайлов, Г.М. Емельянов; НовГУ им. Ярослава Мудрого.– Великий Новгород, 2007. 80 с.
http://www.machinelearning.ru/wiki/images/0/0b/Method_fp.pdf

Практическое занятие 17 и 18

Тема: Логическое программирование. Структура программы на Прологе.

Цель работы

Целью работы является изучение структуры программы на Прологе и интегрированной Пролог-среды (Турбо-Пролога либо Visual Prolog'a).

Методические указания

Концепция языка Пролог.

Пролог является языком программирования, который обеспечивает решение задач, выраженных в терминах объектов и отношений между ними.

Отличительные особенности языка Пролог состоят в следующем:

- Для представления знаний используются фразы Хорна.
- Программа описывает логическую модель Предметной Области в виде фактов относительно свойств Предметной Области и отношений между этими свойствами + правила вывода новых свойств и отношений из уже заданных.
- В качестве единообразной структуры данных используется терм.
- Отсутствуют операторы присваивания, ветвлений, безусловных переходов, а также указатели, которые используются при определении динамических структур данных в традиционных процедурно-ориентированных языках.

Написание программы на языке Пролог включает следующие этапы:

- 1) Объявление некоторых фактов об объектах и отношениях между ними.
- 2) Определение некоторых правил, касающихся объектов и отношений между ними.
- 3) Формулировка вопросов об объектах и отношениях между ними.

Для объяснения смысла программы применяются три семантические модели: декларативная, процедурная модели и модель в виде абстрактной машины.

Процедурная трактовка Пролог-программы.

При процедурной трактовке Пролог-программы подчеркивается последовательность шагов, которые выполняет интерпретатор при обработке запроса. Здесь имеет значение порядок следования подцелей в правиле.

Множество фраз, имеющих одно и то же имя и одинаковое количество аргументов, рассматривается как процедура. При этом запрос (или подцель правила) является вызовом процедуры.

Достоинство: позволяет адекватно представлять фразы, в которых важен порядок следования подцелей. Пример: вывод сообщений на экран в определенной последовательности.

Недостаток: невозможность разъяснения смысла фраз, вызывающих побочные эффекты управления. Примеры: остановка выполнения запроса, удаление фразы из программы.

Декларативная трактовка Пролог-программы.

При декларативной трактовке Пролог-программы специфицируются истинностные значения конкретных случаев отношений.

Для декларативной модели фразы Пролога являются формулами логики предикатов 1-го порядка, записанными в форме фраз Хорна.

Достоинство: эффективность представления знаний ввиду близости к семантике логики

предикатов.

Недостаток: невозможность адекватного представления фраз, в которых важен порядок следования подцелей.

Модель в виде абстрактной машины.

С позиций декларативной модели Пролог-программа есть описание логической структуры. При выполнении запроса интерпретатор применяет по отношению к множеству фраз Пролога некоторую стратегию решения задачи. С точки зрения вычислений эта стратегия может быть описана при помощи некоторой абстрактной машины.

В языке Пролог запрос и множество фраз программы имеет вычислительный смысл. Это проявляется в том, что они вызывают определенное поведение интерпретатора Пролога. Модель в виде абстрактной машины описывает смысл запроса и множества фраз через действия этой машины.

Действия такой абстрактной машины можно рассматривать как применение правила резолюции.

Достоинство: модель в виде абстрактной машины наиболее точная из трех рассматриваемых моделей.

Недостаток: большая зависимость от реализации языка.

Факты и правила.

Факт понимается как запись того отношения, значение которого истинно.

Форма записи факта:

имя_предиката(аргумент {, аргумент}).

При этом :

- 1) Все имена предикатов и аргументов должны начинаться со строчной латинской буквы.
- 2) Перечисление аргументов — через запятую.
- 3) Каждый факт должен заканчиваться точкой.
- 4) Количество аргументов и вид отношений (направления отношений) определяются программистом и не меняются при выполнении программы.

К примеру, факт:

who_likes_what(ivan, programming)

не эквивалентен факту:

who_likes_what(programming, ivan),

поскольку имеет место изменение направления отношения «Иван увлекается программированием» на «Программирование увлекается Иваном».

Сам тип отношений в Турбо-Прологе и Visual Prolog'e описывается в разделе predicates. К примеру, для рассматриваемого who_likes_what это будет who_likes_what(symbol, symbol).

При описании фактов и правил в разделе clauses программы на Турбо-Прологе (Visual Prolog'e) одноименные предикаты должны быть сгруппированы:

clauses

who_likes_what(ivan, programming).

who_likes_what(ivan, reading).

who_likes_what(mary, reading).

В терминологии Пролога любая совокупность фактов (и правил) называется **Базой Данных** (БД).

Под **правилами** в Прологе понимаются наиболее общие утверждения об объектах и отношениях между ними.

Пролог-правило имеет вид фразовой формы:

закключение: – усл1, усл2, . . . , услN.

Данное выражение считается основным в Прологе. Языки, подобные Прологу, считаются языками типа «если-то»: заключение истинно, если истинными являются все условия, перечисленные в правой части.

Пример правила для рассмотренной выше БД:

```
common_interests(X, Y): –  
who_likes_what(X, Z),  
who_likes_what(Y, Z),  
X<>Y.
```

Термы и операторы.

Пролог-программа состоит из термов.

Терм – это либо константа, либо переменная, либо структура (=составной объект). Терм записывается как последовательность литер.

Константы представляют собой имена конкретных (не абстрактных!) объектов, либо конкретных отношений. Существует два вида констант: **атомарные символы** (в том числе **специальные символы**) и **целые числа**.

Переменная служит для представления объекта, на который нельзя сослаться по имени.

В Прологе **переменная** обозначает объект, значение которого может быть найдено. При этом имя переменной не связывается с какой-то конкретной областью памяти, как это делается в процедурных языках.

В отличие от атомов, имя переменной начинается с прописной буквы или знака подчеркивания. Если обозначаемый переменной объект в рассматриваемом контексте не является существенным, то переменная считается анонимной и обозначается символом «_».

Переменная называется **конкретизированной**, если существует объект, который она обозначает. Если не известно, что именно обозначает переменная, то считается, что переменная **не конкретизирована**.

Под **структурой** (составным объектом) в Прологе понимается единый объект, который состоит из совокупности других объектов, называемых в этом случае компонентами:

имя_функтора(компонента {, компонента}).

Иногда удобно записывать некоторые функторы как **операторы**. В Прологе операторы не вызывают выполнения каких-либо арифметических операций.

Так, терм 3+4 – другой способ записи структуры +(3,4).

При построении арифметических выражений необходимо знание трех основных свойств каждого оператора:

- Позицию;
- Приоритет;
- Ассоциативность.

Операторы +, -, *, / записываются между своими аргументами и называются **инфиксными**.

Операторы «+» и «-» могут быть записанными перед своими аргументами. Пример: изменение знака: -x+y. В данной ситуации оператор «-» является **префиксным**.

Оператор, записываемый после своего аргумента, называется **постфиксным**. Пример: математическая запись факториала: n!

Приоритет оператора определяет, какая операция выполняется первой.

В Прологе каждый оператор связан со своим классом приоритета.

Класс приоритета есть целое число, величина которого зависит от конкретной версии Пролога. Однако в любой версии оператор с большим приоритетом имеет класс приоритета,

более близкий к 1.

От свойства ассоциативности операторов зависит порядок выполнения операторов с одинаковым приоритетом. Пример: $Y=8/2/2$.

В подобных случаях необходимо знать, является ли данный оператор **левоассоциативным** или **правоассоциативным**.

Левоассоциативный оператор должен иметь слева операции одинакового или более высокого приоритета, а справа — операции низшего приоритета.

В Прологе арифметические операции (+, -, *, /) являются левоассоциативными. Это означает, что выражения, подобные $Y=8/2/2$ интерпретируются как $Y=(8/2)/2$.

Вопросы.

Запись вопросов в Прологе сходна с записью фактов и отличается местоположением. Для формулировки вопросов в программах на ТурбоПрологе и Visual Prolog'е существует раздел goal.

Обращение к Прологу с вопросом инициализирует процедуру поиска в базе данных, ранее введенной в систему. Пролог просматривает БД в поисках предиката, сопоставимого с вопросом.

Предикаты считаются **сопоставимыми**, если они совпадают посимвольно и их соответствующие аргументы попарно совпадают.

Если предикат вопроса сопоставим с предикатом одного из фактов/правил в БД, то считается, что **вопрос согласуется с БД**. При этом ответом на вопрос будет либо Yes, либо No.

Переменные в вопросах используются для того, чтобы найти какой-либо объект. Пример: для вопроса `who_likes_what(ivan, X)` ответом будет :

X=programming

X=reading 2 Solutions

Yes

В сложных вопросах, содержащих конъюнкцию, и в правилах Пролог при согласовании каждому целевому утверждению приписывает его **собственный маркер**. Если в базе данных есть факт, соответствующий целевому высказыванию, то Пролог отмечает найденное место и пытается согласовать оставшиеся целевые высказывания. Для каждой подцели просмотр БД начинается с начала.

Согласование целевых утверждений.

Для доказательства целевых утверждений Пролог использует известные утверждения. Если **подцель** представляет собой факт, то согласование заканчивается, как только факт будет найден в базе. Если подцель есть правило, то задача сводится к конъюнкции предикатов подцелей. *При согласовании Пролог руководствуется следующими правилами:*

- Константа равна только самой себе.
- Переменная может быть конкретизирована любым объектом Пролога.
- Структуры совпадают, если совпадают их функторы, положение и количество аргументов.

Для каждой подцели Пролог генерирует свой маркер. Если целевое утверждение не доказано, возбуждается процесс возврата.

Как в теле правила, так и в вопросе Пролог пытается согласовать входящие в конъюнкцию целевые утверждения в том порядке, в каком они написаны (слева направо). Доказательство согласованности целевого утверждения с базой данных включает в себя поиск

соответствующих (сопоставимых) утверждений, пометку этого места базы данных и затем доказательство возникших подцелей. В ходе доказательства согласованности целевых утверждений с базой данных происходит конкретизация переменных. Показанный процесс получил название **поиска с возвратом**.

Когда целевое утверждение недоказуемо, осуществляется возврат по «цепочке доказательств» в место выбора утверждения для согласования заново соответствующих целевых утверждений. При этом Пролог пытается найти альтернативное утверждение, соответствующее данной цели. Вначале происходит расконкретизация всех переменных, конкретизированных в ходе доказательства данного целевого утверждения. Затем возобновляется поиск в базе данных, начиная с помеченного маркером места. Если будет найдено другое утверждение, соответствующее целевому, Пролог помещает в это место маркер, поиск в базе данных продолжается, начиная с помеченного места.



Рис. 1. Диаграмма согласования ЦУ, общий случай

Согласование целевых утверждений при наличии правил.

В качестве примера рассмотрим последовательность действий по согласованию целевого утверждения

`common_interests(ivan, mary)`

при наличии правила `common_interests` в рассмотренной выше БД. Факт `common_interests(ivan, mary)` в БД отсутствует, но есть соответствующее правило. Пролог отмечает это место в БД. При этом переменная `X` конкретизируется значением `ivan`, `Y` — значением `mary`.

Вначале Пролог ищет соответствие для предиката

`who_likes_what(ivan, Z)`.

Факт, с которым происходит сопоставление, есть `who_likes_what(ivan, programming)`, и тем

самым первая цель достигнута.

Пролог отмечает маркером соответствующее место в БД и записывает, что *Z* присвоено значение *programming*.

Затем Пролог пытается найти соответствие для следующего предиката в правиле, для чего ищет в базе данных факт *who_likes_what(mary, programming)* и, не находя его, расконкретизирует переменную *Z*. Теперь Пролог продолжает поиск соответствия для предиката *who_likes_what(ivan, Z)* начиная от того места в БД, где находится маркер, и находит факт *who_likes_what(ivan, reading)*. При этом *Z* присваивается значение *reading*. Утверждение *who_likes_what(mary, reading)* успешно согласуется с БД. Последняя цель в конъюнкции также успешно достигается и тем самым доказывается, что факт *common_interests(ivan, mary)* является истинным, Пролог отвечает *Yes*.

Структура программы на Турбо-Прологе.

Программа на Турбо-Прологе имеет следующие разделы:

- *domains* — для описания типов данных;
- *predicates* — для описания предикатов, которые используются в программе;
- *database* — для описания предикатов динамической базы данных;
- *clauses* — для описания фактов и правил (предикатов), составляющих программу;
- *goal* — для целевых утверждений (вопросов).

Разделы *predicates* и *clauses* должны присутствовать в любой программе, раздел *domains* используется для описания пользовательских типов и списков. Если в программе используются только стандартные типы данных: *symbol*, *string*, *char*, *integer*, *real*, *file*, то этот раздел может быть опущен.

Раздел *database* должен присутствовать только в тех программах, которые работают с динамической базой данных, т. е. изменяют программу в процессе ее работы, в противном случае этот раздел опускается.

Раздел *goal* должен содержать сформулированное на Прологе назначение программы и часто этот раздел также опускается. Комментарии могут помещаться в любом месте программы, на их длину нет ограничений, обрамляются комментарии символами */* ... */*.

Основные разделы Visual Prolog-программ.

Синтаксис Visual Prolog-программ в общих чертах сходен с синтаксисом Турбо-Пролога.

Обычно программа на Visual Prolog'е состоит из четырех основных программных разделов:

- раздела *clauses* (предложений);
- раздела *predicates* (предикатов);
- раздела *domains* (доменов);
- раздела *goal* (целей).

Помимо основных разделов, в программах на Visual Prolog'е часто используются разделы фактов (*facts*), констант (*constants*) и различные глобальные (*global*) разделы.

Раздел фактов.

В ряде случаев в процессе работы программы необходимо модифицировать (изменить, удалить или добавить) некоторые из фактов, с которыми она работает. При этом факты рассматриваются как динамическая или внутренняя база данных, которая при выполнении программы может изменяться. Для объявления тех фактов программы, которые рассматриваются как части динамической (или изменяющейся) базы данных, в Visual Prolog-программу включают специальный раздел — *facts*. Ключевое слово *facts* в Visual Prolog'е является синонимом ключевого слова *database* в Турбо-Прологе.

Раздел констант.

В Visual Prolog-программах используется следующий синтаксис объявления констант:

<Id>=<Макроопределение>,

где <Id> – имя символической константы, <Макроопределение> – присваиваемое ей значение.

Перед компиляцией программы Visual Prolog заменяет каждую константу на соответствующую ей строку.

Глобальные разделы.

Visual Prolog позволяет объявить некоторые разделы domains, predicates, clauses глобальными, для этого перед названием раздела помещается ключевое слово global: global domains, global predicates, global facts.

Директивы компилятора.

Visual Prolog поддерживает несколько директив компилятора, которые можно добавлять в программу для сообщения компилятору специальных инструкций по обработке программы во время компиляции.

Так, наиболее часто используемые в программе процедуры описываются в отдельном файле, а для вызова их из других файлов программы применяется директива include:

include «myfile.pro»

Большинство директив компилятора можно устанавливать с помощью меню среды визуальной разработки Visual Prolog: Options | Project | Compiler Options.

Задание для выполнения

Для Турбо-Пролога

Запустите Турбо-Пролог (файл prolog.exe). Изучите интегрированную среду Турбо-Пролога. Выполните Задания 1-6. Выясните назначение и возможности каждой программы.

Для Visual Prolog'a

Запустите Visual Prolog. Изучите визуальную среду разработки (VDE, Visual Development Environment). При этом обратить внимание на работу эксперта приложений, эксперта кода, браузера исходного кода, интерактивной справочной системы.

Создайте тестовый проект и выполните программу «Hello, world!»:

- 1) Создайте новый проект. Для этого выберите команду Project | New Project, активизируется диалоговое окно Application Expert.
- 2) Определите базовый каталог и имя проекта. Имя в поле Project Name следует определить как «TestGoal». Установите флажок Multiprogrammer Mode и щелкните мышью внутри поля Name of . PRJ File. При этом внутри данного поля появится имя файла проекта TestGoal.prj (рис.2).

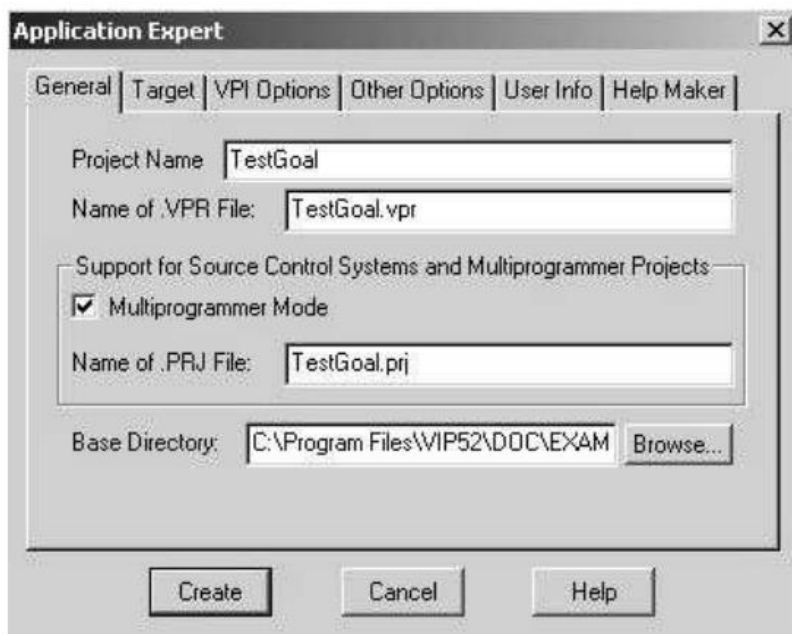


Рис. 2. Общие установки проекта в окне Application Expert

3) Определите цель проекта, для чего на вкладке Target для нашего тестового проекта следует выбрать параметры, отмеченные на рис.3. После этого нажмите кнопку Create, чтобы создать файлы проекта по умолчанию.

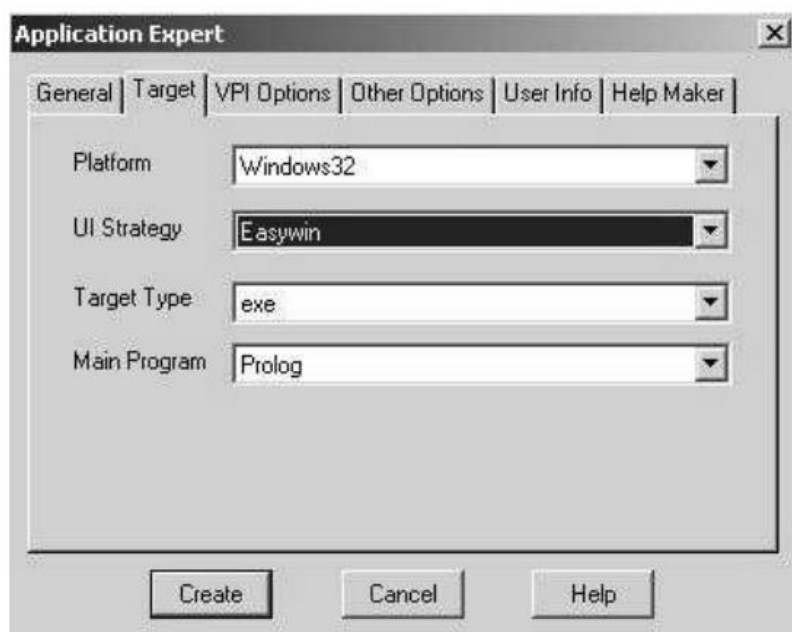


Рис. 3. Параметры цели проекта

4) Установите требуемые опции компилятора для созданного TestGoalпроекта, для чего в главном меню среды визуальной разработки следует выбрать команду Options | Project | Compiler Options. При этом активизируется окно Compiler Options. В этом окне откройте вкладку Warnings (рис.4).

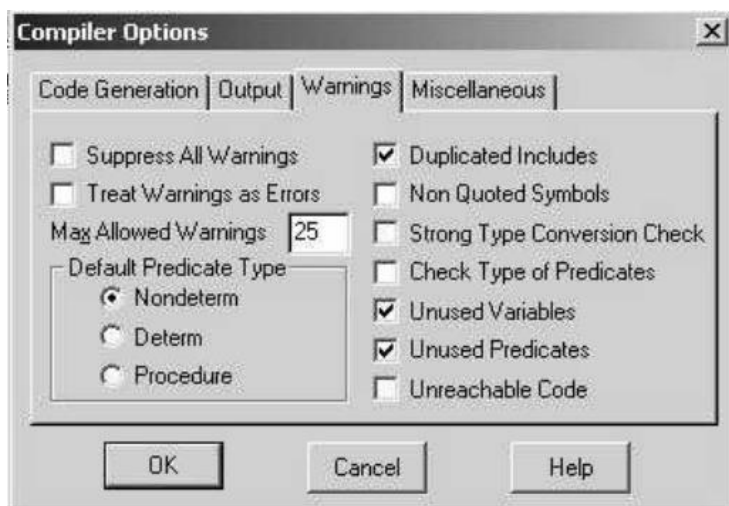


Рис. 4. Установки опций компилятора

5) Выполните следующие действия:

- Установите переключатель Nondeterm. При этом компилятором Visual Prolog'a по умолчанию принимается, что все определяемые в программе предикаты являются недетерминированными, то есть могут породить более одного решения.
- Снимите флажки Non Quoted Symbols, Strong Type Conversion Check и Check Type of Predicates. Это будет подавлять некоторые возможные предупреждения компилятора, которые не важны для понимания выполняемого примера.
- Нажмите кнопку ОК, чтобы сохранить установки опций компилятора.

6) Создайте новый файл программы, для чего в главном меню среды визуальной разработки выберите команду File | New. При этом на экране появляется новое окно редактирования с именем noname.pro (рис.5).

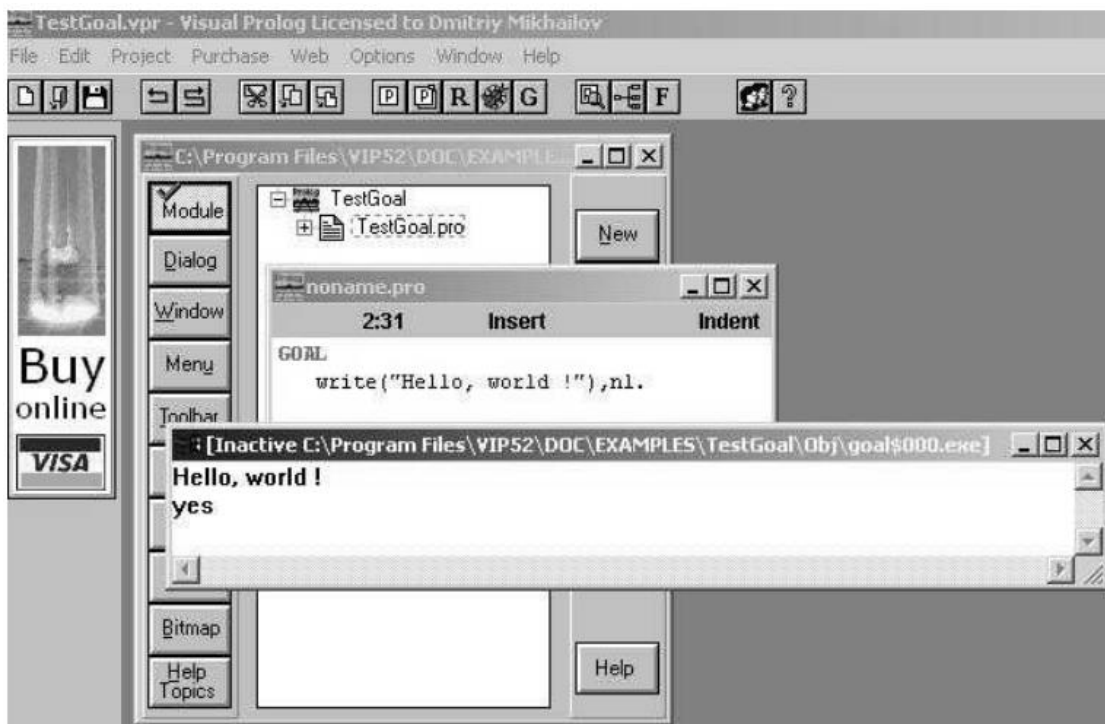


Рис. 5. Тестовая программа «Hello, word!»

7) В появившемся окне напечатайте следующий программный код:

Goal write(«Hello, world !»),nl.

Для выполнения введенной программы необходимо в главном меню среды визуальной разработки выбрать команду Project | Test Goal. Результат ее работы представлен на рис.5.

После создания и апробации тестового примера «Hello, word!» выполните Задания 2–6.

При этом, находясь в проекте TestGoal, следует использовать команду File | Open главного меню для открытия файлов упражнений и команду Project | Test Goal (или комбинацию клавиш <Ctrl>+<G>) для тестирования загружаемых примеров.

Задания

1. Запустите приведенную ниже программу и выясните, как она выполняется :

predicates

```
male(symbol).
parent(symbol, symbol, symbol).
son(symbol, symbol).
```

clauses

```
male(ivan).
male(tom).
male(john).
parent(tom, ivan, mary).
parent(john, ivan, mary).
parent(anna, ivan, mary).
parent(olga, ivan, mary).
son(X, Y): — male(X), parent(X, Y, _). /* X — сын, Y — отец*/
son(X, Y): — male(X), parent(X, _, Y). /* X — сын, Y — мать*/
```

2. Дана программа, не содержащая разделов domains и goal :

predicates

```
common_interests(symbol, symbol)
who_likes_what(symbol, symbol)
likes(symbol, symbol)
```

clauses

```
common_interests(X, Y): —
    who_likes_what(X, Z), Z<>X,
    who_likes_what(Y, Z), X<>Y.
who_likes_what(ivan, reading).
who_likes_what(peter, programming).
who_likes_what(peter, reading).
who_likes_what(mike, tennis).
who_likes_what(mike, programming).
likes(X, Y): —
    who_likes_what(X, Y);
    common_interests(X, Y).
```

Запустите программу и выясните :

- что нравится каждому из упомянутых людей;
- кто и с кем имеет общие интересы;
- кто увлекается программированием.

3. Дана программа:

domains

```
title, author = symbol
pages = integer
```

predicates

```

book(title, pages)
written_by(author, title)
long_novel(title)
clauses
  written_by(fleming, «DR NO»).
  written_by(melville, «MOBY DICK»).
  book(«MOBY DICK», 250).
  book(«DR NO», 310).
  long_novel(Title): —
    written_by(_, Title),
    book(Title, Length),
    Length > 300.

```

Выясните назначение этой программы. Дополните программу информацией о книгах, изданных на русском языке. Добавьте в программу правила:

- отыскивающее самую тонкую книгу в БД;
- отыскивающее самую толстую книгу в БД;
- отыскивающее все книги в БД, не являющиеся самой тонкой и самой толстой;
- определяющее наличие самой тонкой и самой толстой книги в Базе Данных среди произведений заданного автора;
- определяющее самую тонкую и самую толстую книгу для заданного автора;
- отыскивающее в БД для заданного автора все книги, которые он не писал.

4. «Новый магазин».

Имеется множество названий магазинов, среди которых есть существующие в Вашем городе. Имеется также множество товаров, среди которых имеются экзотические, которых нет на прилавках магазинов Вашего города. Предположим, что Вы разрабатываете проект нового магазина, торгующего экзотикой. Требуется:

- из множества названий магазинов выбрать для нового магазина то название, которым не назван ни один из существующих в Вашем городе;
- из множества всех возможных товаров выбрать экзотические, которые будут продаваться в новом магазине.

Написать программу, которая на основе фактов, описывающих магазины и товары (включая бинарное отношение «имеется в продаже в магазинах города»), находила бы название для нового магазина и определяла бы те товары, которые будут в нем продаваться.

5. Решить задачу из Таблицы 1.

Таблица 1. Варианты заданий.

Вариант	Задание
1	Четыре девочки: Надя, Вера, Галя и Люба, нарисовали по одному животному. Получились две лошади, одна овца и одна корова. Что нарисовала каждая девочка, если : — Надя и Вера не рисовали овцу; — Галя не рисовала корову; — Люба не рисовала лошадь; — Вера и Галя, Галя и Люба, Надя и Люба нарисовали разных животных.
2	«Грибники и ягоdniки».

	Три девочки: Оля, Лена и Маша, пошли в лес. Двое из них собирали грибы, одна — ягоды. Что собирала каждая девочка, если Оля с Леной и Маша с Леной не собирали одно и то же.
3	Четверо студентов: Иван, Петр, Михаил и Сергей, участвовали в олимпиадах по математике, физике и программированию. Определить вид дисциплины, в которой участвовал каждый студент, если: — Сергей и Петр не участвовали в олимпиаде по программированию; — Михаил не участвовал в олимпиаде по математике; — Иван не участвовал в олимпиаде по физике; — Сергей и Иван выступали в одной дисциплине; — Каждый студент участвовал ровно в одной дисциплине.
4	Четверо друзей: Иван, Петр, Михаил и Сергей, проводили свободное время по-разному: двое играли в шахматы, один читал книги, один смотрел телевизор. Определить, кто каким образом проводил время, если Сергей не играл в шахматы, а Петр не смотрел телевизор.
5	Четверо друзей: Иван, Петр, Михаил и Сергей, проводили летние каникулы по-разному: один был в деревне, один оставался дома, один был на море, один — в оздоровительном лагере. Определить, кто каким образом отдыхал, если: — Иван, Петр и Михаил не были в оздоровительном лагере; — Иван и Михаил не были в деревне; — Михаил не оставался дома.
6	Иван, Петр, Михаил и Сергей посвящали свое свободное время чтению книг. Один читал о путешествиях, другой — о войне, третий — о спорте, четвертый — детектив. Кто о чем читал, если: — Иван и Петр не читали о войне; — Михаил и Сергей не читали о спорте; — Петр и Михаил не читали детектив; — Иван и Сергей не читали о путешествиях.
7	Четыре девочки: Надя, Вера, Галя и Люба, сажали плодовые деревья. Одна сажала яблони, другая груши, третья сливы, четвертая — вишни. Что сажала каждая девочка, если: — Вера не сажала яблони и груши; — Галя не сажала яблони и сливы; — Люба не сажала сливы и вишни; — Надя не сажала груши и вишни.

2. Решить задачу из Таблицы 2.

Таблица 2. Варианты заданий.

Вариант	Задание
1	Можно ли расставить на гранях куба целые числа от 1 до 6 так, чтобы каждое число являлось делителем суммы своих соседей ?

2	Расшифруйте числовой ребус : T>P>A>H<C<П<O<P<T>И>P>O<B<K<A (разные буквы обозначают разные цифры)
3	Расшифруйте числовой ребус: БЕЕЕ - М = МУУУ (одинаковым буквам соответствуют одинаковые цифры, разным — разные).
4	Расшифруйте числовой ребус: т р и - д в а = я р д (одинаковым буквам соответствуют одинаковые цифры, разным — разные).
5	Расшифруйте числовой ребус: $\{MA * MA = МИР AM * AM = РИМ$ (одинаковым буквам соответствуют одинаковые цифры, разным — разные).

Литература

1. Сергиевский Г.М. Функциональное и логическое программирование: учеб. пособие для студ. высш. учеб. заведений / Г. М. Сергиевский, Н.Г. Волчёнков. — М.: Издательский центр «Академия», 2010. — 320 с.
2. Игонина Е.В. Основы логического программирования и реализация программ на языке Пролог: учебное пособие. — Елец: ЕГУ им. И.А. Бунина, 2018. — 92 с.
3. Попов С.Г. Программирование на языке ПРОЛОГ: Лабораторный практикум. 2014. 49 с.
4. Михайлов Д. В. Функциональное и логическое программирование. Ч. 2. Логическое программирование: Лабораторный практикум / Сост. Д.В. Михайлов, Г.М. Емельянов; НовГУ им. Ярослава Мудрого. — Великий Новгород, 2007. — 88 с.
5. Адаменко А. Н., Кучуков А. М. Логическое программирование и Visual Prolog. — СПб. : БХВ-Петербург, 2003. — 992 с.

Практическое занятие 19 и 20

Тема: Логическое программирование. Экспертные системы на Прологе.

Цель работы

Целью работы является изучение принципов построения и организации экспертных систем, базирующихся на логике и правилах.

Методические указания

Назначение и общая структура экспертных систем.

Под **Экспертной Системой (ЭС)** понимается программа (комплекс программ), моделирующая в некоторой степени работу человека-эксперта в конкретной предметной области. Причем эта область строго ограничена.

Основное назначение ЭС — проведение консультаций в той предметной области, для которой спроектирована данная ЭС.

В составе ЭС выделяется три компонента (рис.1) :

- 1) **База Знаний (БЗ).** БЗ — центральная часть Экспертной Системы. Она содержит совокупность фактов и знаний (правил) для вывода других знаний. Содержащаяся в БЗ информация используется и спользуется ЭС для вывода экспертного заключения во время консультации. Обычно БЗ располагают отдельно от программы, на диске или другом носителе.
- 2) **Механизм Вывода (МВ).** МВ содержит описания способов применения содержащихся в БЗ знаний. Во время консультации МВ запускает ЭС в работу, выполняет правила, определяет приемлемость найденного решения и передает результаты в СПИ.
- 3) **Система Пользовательского Интерфейса (СПИ).** СПИ есть та часть ЭС, которая взаимодействует с пользователем. В функции СПИ входит: прием информации от пользователя, передача результатов пользователю в наиболее удобной для него форме, объяснение полученных ЭС результатов (выдача справочной информации по выводу результатов).

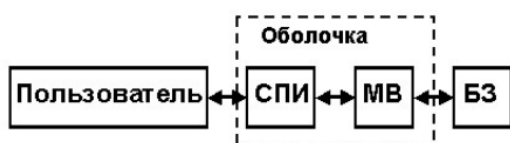


Рис. 1. Общая структура ЭС

В зависимости от способов классификации и размещения информации Базы Знаний различают: **продукционную, сетевую и фреймовую** модели представления знаний.

Сетевая модель основана на представлении знаний в виде сети, вершины которой соответствуют понятиям, а дуги — отношениям между ними.

В основе **фреймовой модели** лежит логическая группировка атрибутов объекта, при этом для хранения и обработки логические группы описываются во фреймах.

Продукционная модель основывается на правилах вида «если–то» и позволяет помещать фрагменты фактического знания в правила языка Пролог. Именно так строятся ЭС, базирующиеся на правилах. При реализации ЭС, базирующейся на логике, БЗ представляется совокупностью утверждений в виде фактов. Вывод экспертного заключения

при этом строится на основе стандартных средств работы со списками.

Вывод экспертного заключения.

Под выводом в ЭС понимается доказательство того, что из множества предположений следует некоторое заключение. Принятая логика получения заключения специфицируется правилами вывода. Вывод осуществляется посредством поиска и сопоставления по образцу.

В ЭС, базирующейся на правилах, запросы пользователя трансформируются в форму, сопоставимую с формой правил БЗ.

Механизм вывода инициализирует процесс сопоставления, начиная с «верхнего» правила. Обращение к правилу называется **«вызовом»**. Вызов соответствующих правил в процессе сопоставления продолжается до тех пор, пока не произошло сопоставление или не исчерпана вся БЗ, а значение не найдено. Если МВ обнаруживает, что можно вызвать более одного правила, то запускается процесс разрешения конфликта. При разрешении конфликта приоритет отдается обычно тем правилам, которые более конкретны, либо правилам, которые учитывают больше текущих данных.

В ЭС, базирующейся на логике, трансформированные запросы являются значениями, которые сопоставляются со списками значений, находящихся в утверждениях БЗ.

Внутренние программы унификации Турбо-Пролога и Visual Prolog'a позволяют решать задачи поиска и сопоставления по образцу без написания дополнительных правил. Как в системе, базирующейся на правилах, так и в системе, базирующейся на логике, пользователь получает ответы на свои запросы в соответствии с заложенной в ЭС логикой. Для программной реализации механизма вывода экспертного заключения достаточно только написать необходимые спецификации.

ЭС, базирующаяся на правилах, и ее реализация.

ЭС, базирующаяся на правилах, позволяет проектировщику строить правила, которые естественным образом объединяют в группы связанные фрагменты знаний. При этом взаимная независимость продукционных правил делает базу правил семантически модульной и способной к развитию.

Реализованная на Турбо-Прологе (или Visual Prolog'e) ЭС на правилах содержит множество правил, которые вызываются посредством входных данных в момент сопоставления. Наряду с множеством правил МВ ЭС имеет в своем составе интерпретатор, который выбирает и активизирует различные модули системы. *Работа этого интерпретатора описывается последовательностью трех шагов:*

- 1) Интерпретатор сопоставляет образец правила с элементами данных в БЗ.
- 2) Если можно вызвать более одного правила, то для выбора правила используется механизм разрешения конфликта.
- 3) Выбранное правило применяется для поиска ответа на поставленный вопрос.

Этот трехшаговый процесс является циклическим и называется **циклом распознавание-действие**. Утвердительный ответ ЭС является результатом выполнения одного из продукционных правил, выбор правила производится в соответствии с входными данными. Написание на Турбо-Прологе (или Visual Prolog'e) ЭС, базирующейся на правилах, начинается с декларации БД. БД хранит ответы пользователя на вопросы СПИ. Эти данные являются утвердительными или отрицательными ответами. Далее строятся продукционные правила, описывающие фрагменты фактического знания.

Пример (для ЭС идентификации и выбора породы собак)

```

dog_is(«английский бульдог»): —
    it_is(«короткошерстная»),
    positive(«имеет», «рост меньше 22 дюймов»),
    positive(«имеет», «свисающий хвост»),
    positive(«имеет», «хороший характер»), !.

```

/* Аналогично описываются знания о других породах собак */

...

```

dog_is(«коккер-спаниэль»): —
    it_is(«длинношерстная»),
    positive(«имеет», «рост меньше 22 дюймов»),
    positive(«имеет», «свисающий хвост»),
    positive(«имеет», «длинные уши»),
    positive(«имеет», «хороший характер»), !.

```

Причем с целью ограничения пространства поиска описывающие связанные фрагменты знаний правила могут быть сгруппированы посредством введения в базу вспомогательных правил для идентификации подкатегорий. К примеру, в ЭС выбора породы собаки это будет правило `it_is`, которое идентифицирует породу собаки по признаку принадлежности к группе длинношерстных или короткошерстных:

```

it_is(«короткошерстная»): —
    positive(«собака имеет», «короткую шерсть»), !.
it_is(«длинношерстная»): —
    positive(«собака имеет», «длинную шерсть»), !.

```

В рассматриваемом здесь примере основные данные для выбора породы собаки и являются общеизвестными, поскольку выбраны распространенные породы собак. Набор признаков для классификации пород выбран на основании следующих соображений:

— Все используемые атрибуты являются необходимыми для идентификации породы.

— Ни один из атрибутов не характерен для всех пород одновременно.

Правила МВ сопоставляют данные пользователя с данными в продукционных правилах (правила `positive` и `negative` в рассматриваемой ЭС), а также сохраняют «трассу» утвердительных и неутвердительных (отрицательных) ответов (правило `remember` для добавления в БД утверждений с ответами 1 (да) и 2 (нет)), которые используются при сопоставлении с образцом:

/* Механизм вывода экспертного заключения.

`xpositive(X, Y)` и `xnegative(X, Y)` — предикаты динамической БД, хранят, соответственно, утвердительные и неутвердительные ответы пользователя на вопросы, которые СПИ задает на основе значений аргументов предиката `positive` в теле очередного варианта

правила `dog_is` */

```

positive(X, Y): —
    xpositive(X, Y), !.
positive(X, Y): —
    not(negative(X, Y)), !, ask(X, Y).
negative(X, Y): —
    xnegative(X, Y), !.

```

/* Консультация. Предикат `dlg_Ask` создает стандартное диалоговое окно получения ответа пользователя на вопрос Да/Нет, */

```

ask(X, Y): —
    concat(«Вопрос : », X, Temp),
    concat(Temp, « », Temp1),
    concat(Temp1, Y, Temp2),
    concat(Temp2, «?», Quest),

```

```

Reply1=dlg_Ask(«Консультация», Quest, [«Да», «Нет»]),
Reply=Reply1+1,
remember(X, Y, Reply).
/* Занесение ответов пользователя в динамическую БД */
remember(X, Y, 1): — !,
    assertz(xpositive(X, Y)).
remember(X, Y, 2): — !,
    assertz(xnegative(X, Y)), fail.

```

ЭС, базирующаяся на логике, и ее реализация.

Здесь БЗ состоит из утверждений в виде предложений логики предикатов. При этом часть утверждений описывают объекты, другая часть — условия или атрибуты, которые характеризуют различные объекты.

Количество признаков определяет степень точности классификации.

Интерпретатор внутри системы выполняет свои функции на основе следующей схемы :

- 1) Система содержит в БЗ предложения, которые управляют поиском и сопоставлением. Интерпретатор сопоставляет эти предложения с элементами данных в БД.
- 2) Если существует возможность вызова более одного правила, то для разрешения конфликта система использует возможности механизма внутренней унификации Пролога.
- 3) Система получает результаты унификационного процесса автоматически, поэтому они направляются на нужное (логическое) устройство вывода информации.

Так же, как и в ЭС, базирующейся на правилах, данный циклический процесс является процессом распознавание-действие.

Основное отличие структуры ЭС, базирующейся на логике, состоит в описании объектов и атрибутов в виде фактов :

```

/* Условия-характеристики различных пород.*/
cond(1, «короткошерстная порода»).
cond(2, «длинношерстная порода»).
cond(3, «рост меньше 22 дюймов»).
cond(4, «рост больше 30 дюймов»).
cond(5, «свисающий хвост»).
cond(6, «длинные уши»).
cond(7, «хороший характер»).
cond(8, «вес более 100 фунтов»).

/* Данные о типах породы */
topic(«короткошерстная»).
topic(«длинношерстная»).

/* Данные о конкретных породах */
rule(1, «собака», «короткошерстная», [1]).
rule(2, «собака», «длинношерстная», [2]).
rule(3, «короткошерстная», «английский бульдог», [3,5,7]).
rule(4, «короткошерстная», «гончая», [3,6,7]).
rule(5, «короткошерстная», «датский дог», [5,6,7,8]).
rule(6, «короткошерстная», «американский фокстерьер», [4,6,7]).
rule(7, «длинношерстная», «коккер-спаниэль», [3,5,6,7]).
rule(8, «длинношерстная», «ирландский сеттер», [4,6]).
rule(9, «длинношерстная», «колли», [4,5,7]).
rule(10, «длинношерстная», «сенбернар», [5,7,8]).

```

Третий аргумент предиката rule — список целых чисел-номеров условий из предложений типа cond. Каждое предложение типа cond задает свое условие-признак из используемых здесь для классификации пород собак. В ЭС, базирующейся на логике, интерпретатор

использует эти номера условий, чтобы делать соответствующий выбор.

МВ содержит правила обработки списков атрибутов в описаниях объектов. Посредством применения правила go МВ просматривает утверждения БЗ rule и cond для выяснения с помощью правила check существования или отсутствия подходящих значений данных :

```
/* Начальное правило механизма вывода */
```

```
go(_, Mygoal): —
    not(rule(_, Mygoal, _, _)), !,
    concat(«Рекомендуемая порода : », Mygoal, Temp),
    concat(Temp, «.», Result),
    dlg_Note(«Экспертное заключение : », Result).
go(History, Mygoal): —
    rule(Rule_number, Mygoal, Type_of_breed, Conditions),
    check(Rule_number, History, Conditions),
    go([Rule_number|History], Type_of_breed).
```

```
/* Сопоставление входных данных пользователя со списками атрибутов отдельных пород собак */
```

```
check(Rule_number, History, [Breed_cond|Rest_breed_cond_list]): —
    yes(Breed_cond), !,
    check(Rule_number, History, Rest_breed_cond_list).
check(_, _, [Breed_cond|_]): —
    no(Breed_cond), !, fail.
check(Rule_number, History, [Breed_cond|Rest_breed_cond_list]): —
    cond(Breed_cond, Text),
    ask_question(Breed_cond, Text),
    check(Rule_number, History, Rest_breed_cond_list).
check(_, _, [_]).
do_answer(Cond_number, 1): — !,
    assertz(yes(Cond_number)).
do_answer(Cond_number, 2): — !,
    assertz(no(Cond_number)), fail.
```

```
/* Запрос и получение ответов yes и no от пользователя */
```

```
ask_question(Breed_cond, Text): —
    concat(«Вопрос : », Text, Temp),
    concat(Temp, « », Temp1),
    concat(Temp1, «?», Quest),
    Response1=dlg_Ask(«Консультация», Quest, [«Да»,«Нет»]),
    Response=Response1+1,
    do_answer(Breed_cond, Response).
```

Правило go пытается сопоставить объекты, классифицированные при помощи номеров условий. Если сопоставление происходит, то данный модуль программы должен добавить в БЗ сопоставленные значения и продолжить процесс с новыми данными, полученными от пользователя. Если сопоставление не происходит, механизм останавливает текущий процесс и выбирает для сопоставления другую трассу. Поиск и сопоставление продолжается до тех пор, пока не исчерпаны все возможности.

Достоинством ЭС, базирующейся на логике, является возможность хранения фактов Базы Знаний в утверждениях динамической Базы Данных. При этом База Знаний может быть размещена в файле на диске, что позволяет сделать ее не зависимой от программного кода.

Задание для выполнения

Реализовать экспертную систему по заданной предметной области в соответствии с номером задания в Таблице 1. При этом количество описываемых объектов должно быть не менее 12, а характеризующих их атрибутов — не менее 8. Рассмотреть реализацию экспертной системы как базирующуюся на логике и как базирующуюся на правилах.

Таблица 1. Предметная область для экспертной системы.

Вариант	Предметная область
1	Микропроцессоры
2	Мобильные устройства
3	Операционные системы
4	Языки программирования
5	Компьютерные игры
6	Компьютерные вирусы
7	Компьютерные сети
8	Алгоритмы сортировки
9	Алгоритмы поиска

Литература

6. Сергиевский Г.М. Функциональное и логическое программирование: учеб. пособие для студ. высш. учеб. заведений / Г. М. Сергиевский, Н.Г. Волчёнков. – М.: Издательский центр «Академия», 2010. – 320 с.
7. Игонина Е.В. Основы логического программирования и реализация программ на языке Пролог: учебное пособие. – Елец: ЕГУ им. И.А. Бунина, 2018. – 92 с.
8. Попов С.Г. Программирование на языке ПРОЛОГ: Лабораторный практикум. 2014. 49 с.
9. Михайлов Д. В. Функциональное и логическое программирование. Ч. 2. Логическое программирование: Лабораторный практикум / Сост. Д.В. Михайлов, Г.М. Емельянов; НовГУ им. Ярослава Мудрого. — Великий Новгород, 2007. — 88 с.
10. Адаменко А. Н., Кучуков А. М. Логическое программирование и Visual Prolog. — СПб. : БХВ-Петербург, 2003. — 992 с.
11. Математическая логика и логическое программирование [https://mk.cs.msu.ru/index.php/%D0%9C%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D0%BB%D0%BE%D0%B3%D0%B8%D0%BA%D0%B0_%D0%B8_%D0%BB%D0%BE%D0%B3%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_\(3-%D0%B9_%D0%BF%D0%BE%D1%82%D0%BE%D0%BA\)](https://mk.cs.msu.ru/index.php/%D0%9C%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D0%BB%D0%BE%D0%B3%D0%B8%D0%BA%D0%B0_%D0%B8_%D0%BB%D0%BE%D0%B3%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_(3-%D0%B9_%D0%BF%D0%BE%D1%82%D0%BE%D0%BA))

Практическое занятие 21 и 22

Тема: Объектно-ориентированное программирование. Введение в язык программирования Python.

Цель работы

Познакомиться со средой разработки Python. Изучить основные типы данных, команды ввода и вывода данных.

Методические указания

Типы данных

В Python вам не нужно объявлять тип переменной вручную (как, например в C++). Объявление происходит автоматически (это называется динамическая типизация), когда вы присваиваете значение переменной. Знак равенства (=) используется для присвоения значения переменной. Операнд по левую сторону от знака равно (=) это имя переменной, операнд по правую сторону - значение присвоенное этой переменной.

Встроенные типы данных:

Имя	Тип	Описание и пример
Целые Числа	int	Целые положительные или отрицательные числа -35, 0, 24, 123467890033373747428
Числа с плавающей точкой	float	Дробные числа 3.14, 2.5, -2.33333, 0.12334
Строки	str	Строки «asdf», «Hello world», «123456»
Списки	list	последовательность элементов [«hello», -123, 0.34, «345»]
Словарь	dict	Последовательность пар элементов содержащих ключ-значение (key-value) {«Language»: «Python», «Version»: «3.8»}
Кортеж (Tuple)	tup	Неизменяемая упорядоченная последовательность элементов («hostname», 1234, -0.45, -32)
Множество	set	Изменяемая неупорядоченная последовательность элементов {10, «Name», -30, 4.02, 100}
Булевы значения	bool	Тип данных принимающий одно из двух значений true - истина false - ложь

Строки

Строки - это неизменяемая упорядоченная последовательность символов, заключенная в кавычки. Они являются первым примером того, что в Python называется **последовательностью** — позиционно упорядоченной коллекцией других объектов. Для содержащихся элементов последовательности поддерживают порядок слева направо: элементы сохраняются и извлекаются по своим относительным позициям. Строго говоря, строки представляют собой последовательности односимвольных строк.

Строки можно суммировать. Тогда они объединяются в одну строку, такая операция называется «Конкатенацией строк»

Пример.

```
firts_string = "asdfgh"
second_string = "oiuytr"
print(firts_string + second_string)
```

Извлечь компоненты можно по индексу

Пример.

```
simple_string = 'Spam'
len(simple_string)
# 4
simple_string[0]
# 'S'
simple_string[1]
# 'p'
```

Нумерация всех символов в строке идет с нуля. Но, если нужно обратиться к какому-то по счету символу, начиная с конца, то можно указывать отрицательные значения (на этот раз с единицы).

Пример.

```
simple_string = "StringBody"
simple_string[1]
# t

simple_string[-1]
# y
```

Кроме обращения к конкретному символу, можно делать срезы строк, указав диапазон номеров (срез выполняется по второе число, не включая его):

Пример.

```
example_string = "Lorem Ipsum is simply dummy text of the printing and typesetting"
example_string[0:9]
# 'Lorem Ips'

example_string[10:22]
# 'm is simply '

# Если не указывается второе число, то срез будет до конца строки:
example_string[-3:]
# 'ing'
```

Также в срезе можно указывать шаг:

Пример.

```
# Так можно получить нечетные числа
a = '0123456789'
a[1::2]
# '13579'
# А таким образом можно получить все четные числа строки a:
a[::2]
# '02468'
# Срезы также можно использовать для получения строки в обратном порядке:
a[::-1]
# '9876543210'
```

Операции над последовательностями (строками)

len() – возвращает длину строки

count() используется для подсчета того, сколько раз символ или подстрока встречаются в строке

Методу find() можно передать подстроку или символ, и он покажет, на какой позиции находится первый символ подстроки (для первого совпадения)

Проверка на то, начинается или заканчивается ли строка на определенные символы (методы startswith(), endswith()).

replace() – замена последовательности символов в строке на другую последовательность

split() разбивает строку на части, используя как разделитель какой-то символ (или символы) и возвращает список строк

join() позволяет объединить список, кортеж или словарь в строку разделяя ее элементы другой строкой

format() позволяет подставлять в отмеченные в строке области символами «{}» значения из списка аргументов

Списки

Списки – это изменяемые упорядоченные последовательности произвольных объектов. Списки создаются посредством заключения элементов списка в квадратные скобки. Списки могут содержать объекты любого типа, числа, строки, другие списки. Списки так же как и строки можно конкатинировать между собой.

Пример.

```
names = [ "Dave", "Mark", "Ann", "Phil" ]
```

Элементы списка индексируются целыми числами, первый элемент списка имеет индекс, равный нулю. Для доступа к отдельным элементам списка используется оператор индексирования.

Пример.

```
a = names[2] # Вернет третий элемент списка, "Ann"
names[0] = "Jeff" # Запишет имя "Jeff" в первый элемент списка
b = names[0:2] # Вернет ["Jeff", "Mark"]
c = names[2:] # Вернет ["Thomas", "Ann", "Phil", "Paula"]
names[1] = "Jeff" # Во второй элемент запишет имя "Jeff"
names[0:2] = ["Dave", "Mark", "Jeff"] # Заменит первые два элемента списком справа.
```

Методы для работы со списками

`list()` – создаст пустой список, либо преобразует аргументы в список
`join()` – собирает список строк в одну строку с разделителем, который указан перед `join`
`append()` – добавляет в конец списка указанный элемент
`extend()` – объединяет два списка (`extend` меняет список, к которому применен метод, а суммирование возвращает новый список, который состоит из двух)
`pop()` – удаляет элемент, который соответствует указанному номеру, при этом метод возвращает этот элемент
`remove()` – удаляет указанный элемент и не возвращает удаленный элемент
`insert()` – позволяет вставить элемент на определенное место в списке
`sort()` – сортирует список на месте

Словари

Словари - это изменяемый упорядоченный тип данных:

- данные в словаре - это пары ключ: значение
- доступ к значениям осуществляется по ключу, а не по номеру, как в списках
- данные в словаре упорядочены по порядку добавления элементов
- так как словари изменяемы, то элементы словаря можно менять, добавлять, удалять
- ключ должен быть объектом неизменяемого типа: число, строка, кортеж
- значение может быть данными любого типа.

Пример

```
london = {  
'id': 1,  
'name': 'London',  
'it_vlan': 320,  
'user_vlan': 1010,  
'mngmt_vlan': 99,  
'to_name': None,  
'to_id': None,  
'port': 'G1/0/11'  
}
```

Для того, чтобы получить значение из словаря, надо обратиться по ключу, таким же образом, как это было в списках, только вместо номера будет использоваться ключ

Пример

```
london = {'name': 'London1', 'location': 'London Str'}  
print(london['name'], london['location'])  
#London1 London Str
```

Методы для работы со словарями

`clear()` – позволяет очистить словарь
`copy()` – создает полную копию словаря
`get()` – запрашивает ключ, и если его нет, вместо ошибки возвращает `None`
`setdefault()` – ищет ключ, и если его нет, вместо ошибки создает ключ со значением `None`, если ключ есть, `setdefault` возвращает значение, которое ему соответствует
`keys()`, `values()`, `items()` – все три метода возвращают специальные объекты `view`, которые отображают ключи, значения и пары ключ-значение словаря соответственно.
`del()` – удаляет ключ и значение
`update()` – позволяет добавлять в словарь содержимое другого словаря

Кортежи

Объект кортежа примерно похож на список, который нельзя изменять. Если вы передаете коллекцию объектов внутри своей программы в виде списка, тогда он может быть модифицирован где угодно; если вы используете кортеж, то изменить его не удастся. То есть кортежи обеспечивают своего рода ограничение целостности.

Пример.

```
T = (1, 2, 3, 4) # Кортеж из 4 элементов
len(T)          # Длина 4
T + (5, 6)       # Конкатенация (1, 2, 3, 4, 5, 6)
T[0]            # Индексация, нарезание и т.д.
```

Множество

Множество — это изменяемый неупорядоченный тип данных. В множестве всегда содержатся только уникальные элементы.

Множество в Python — это последовательность элементов, которые разделены между собой запятой и заключены в фигурные скобки. С помощью множества можно легко убрать повторяющиеся элементы.

Методы для работы с множествами

add() — добавляет элемент во множество

discard() — позволяет удалять элементы, не выдавая ошибку, если элемента в множестве нет

clear() — очищает множество

Булевы значения

Булевы значения в Python это две константы True и False.

В Python истинными и ложными значениями считаются не только True и False.

Истинное значение:

- любое ненулевое число
- любая непустая строка
- любой непустой объект

Ложное значение:

- 0
- None
- пустая строка
- пустой объект

Остальные истинные и ложные значения, как правило, логически следуют из условия.

Для проверки булевого значения объекта, можно воспользоваться bool:

Пример.

```
items = [1, 2, 3]
empty_list = []
bool(empty_list) #False
bool(items) #True
bool(0) #False
bool(1) #True
```

Преобразование типов

В Python есть несколько полезных встроенных функций, которые позволяют преобразовать данные из одного типа в другой.

`int()` – преобразует строку в `int`;
`bin()` – преобразует десятичное число в двоичный формат;
`hex()` – преобразует десятичное число в шестнадцатеричный формат;
`list()` – преобразует аргумент в список;
`set()` – преобразует аргумент в множество;
`tuple()` – преобразует аргумент в кортеж;
`str()` – преобразует аргумент в строку.

Проверка типов

`isdigit()` – проверяет, состоит ли строка из одних только цифр;
`isalpha()` – проверяет, состоит ли строка из одних букв;
`isalnum()` – позволяет проверить, состоит ли строка из букв или цифр;
`type()` – возвращает тип аргумента.

Файлы

Объекты файлов являются главным интерфейсом к внешним файлам на компьютере. Файлы относятся к основным типам, но они кое в чем своеобразны – специфический литеральный синтаксис для их создания отсутствует. Чтобы создать объект файла, необходимо вызвать встроенную функцию `open`, передав ей в виде строк имя внешнего файла и необязательный режим обработки.

Например, для создания выходного текстового файла понадобится передать его имя и строку режима обработки `'w'`, чтобы записывать данные:

```
f = open('data.txt', 'w') # Создать новый файл в режиме записи ('w')
f.write('Hello\n') # Записать в него строки символов
f.write('world\n') # Возвратить количество записанных элементов
f.close() # Закрыть для сбрасывания буферов вывода на диск
```

Код создает файл в текущем каталоге и записывает в него текст. Чтобы прочитать то, что было записано, необходимо повторно открыть файл в режиме обработки `'r'` для чтения текстового ввода (он выбирается по умолчанию, если в вызове строка режима не указана). Затем следует прочитать содержимое файла в строку и отобразить ее. В сценарии содержимое файла всегда будет строкой независимо от типа находящихся в нем данных:

```
f = open('data.txt') # 'r' (чтение) - стандартный режим обработки
text = f.read() # Прочитать все содержимое файла в строку
text # 'Hello\nworld\n'
print(text) # print интерпретирует управляющие символы
#Hello
#world
text.split() # Содержимое файла - всегда строка
#['Hello', 'world']
```

Вывод информации на экран

`print()` — встроенная в Python функция, с помощью которой можно вывести текст в консоль. У функции `print()` есть дополнительные аргументы — значения, с помощью которых можно модифицировать вывод данных. Их следует передавать во время вызова. С ними синтаксис `print()` выглядит так:

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

В ней:

`objects` — один или несколько объектов, которые нужно вывести;

sep — разделитель между объектами (по умолчанию пробел);
end — символ завершения строки (по умолчанию перевод строки);
file — объект файла, в который направлен вывод;
flush — принудительный сброс буфера вывода.

Базовые операторы языка Python

Оператор	Название	Объяснение
„+“	Сложение	Суммирует два объекта
„-“	Вычитание	Даёт разность двух чисел; если первый операнд отсутствует, он считается равным нулю
„*“	Умножение	Даёт произведение двух чисел или возвращает строку, повторённую заданное число раз.
„**“	Возведение в степень	Возвращает число x, возведённое в степень y
/	Деление	Возвращает частное от деления x на y
//	Целочисленное деление	Возвращает неполное частное от деления
%	Деление по модулю	Возвращает остаток от деления
<<	Сдвиг влево	Сдвигает биты числа влево на заданное количество позиций. (Любое число в памяти компьютера представлено в виде битов - или двоичных чисел, т.е. 0 и 1)
>>	Сдвиг вправо	Сдвигает биты числа вправо на заданное число позиций.
&	Побитовое И	Побитовая операция И над числами
„ “	Побитовое ИЛИ	Побитовая операция ИЛИ над числами
^	Побитовое ИСКЛЮЧИТЕЛЬНО ИЛИ	Побитовая операция ИСКЛЮЧИТЕЛЬНО ИЛИ
~	Побитовое НЕ	Побитовая операция НЕ для числа x соответствует -(x+1)

<	Меньше	Определяет, верно ли, что x меньше y. Все операторы сравнения возвращают True или False [1]. Обратите внимание на заглавные буквы в этих словах.
>	Больше	Определяет, верно ли, что x больше y
<=	Меньше или равно	Определяет, верно ли, что x меньше или равно y
>=	Больше или равно	Определяет, верно ли, что x больше или равно y
==	Равно	Проверяет, одинаковы ли объекты
!=	Не равно	Проверяет, верно ли, что объекты не равны
not	Логическое НЕ	Если x равно True, оператор вернёт False. Если же x равно False, получим True.
and	Логическое И	x and y даёт False, если x равно False , в противном случае возвращает значение y
or	Логическое ИЛИ	Если x равно True, в результате получим True, в противном случае получим значение y

Управляющие операторы

Условные операторы (if/else)

Оператор if используется для проверки условий: если условие верно, выполняется блок выражений (называемый “if-блок”), иначе выполняется другой блок выражений (называемый “else-блок”). Блок “else” является необязательным.

Пример.

```
number = 23
guess = int(input('Введите целое число : '))
if guess == number:
    print('Поздравляю, вы угадали,') # Здесь начинается новый блок
    print('(хотя и не выиграли никакого приза!))' # Здесь заканчивается новый блок
elif guess < number:
    print('Нет, загаданное число немного больше этого.') # Ещё один блок
    # Внутри блока вы можете выполнять всё, что угодно ...
else:
    print('Нет, загаданное число немного меньше этого.')
    # чтобы попасть сюда, guess должно быть больше, чем number
print('Завершено')
# Это последнее выражение выполняется всегда после выполнения оператора if
```

Цикл for

Оператор `for..in` также является оператором цикла, который осуществляет итерацию по последовательности объектов, т.е. проходит через каждый элемент в последовательности. Мы узнаем больше о последовательностях в дальнейших главах, а пока просто запомните, что последовательность – это упорядоченный набор элементов.

Пример.

```
for x in ["spam", "eggs", "ham"]:  
    print(x, end=' ')  
#результатом этого цикла будет строка spam eggs ham
```

Оператор break

Оператор `break` служит для прерывания цикла, т.е. остановки выполнения команд даже если условие выполнения цикла ещё не приняло значения `False` или последовательность элементов не закончилась.

Важно отметить, что если циклы `for` или `while` прервать оператором `break`, соответствующие им блоки `else` выполняться не будут.

Пример.

```
while True:  
    s = input('Введите что-нибудь : ')  
    if s == 'выход':  
        break  
    print('Длина строки:', len(s))  
print('Завершение')
```

Оператор continue

Оператор `continue` используется для указания Python, что необходимо пропустить все оставшиеся команды в текущем блоке цикла и продолжить со следующей итерации цикла.

Пример.

```
while True:  
    s = input('Введите что-нибудь : ')  
    if s == 'выход':  
        break  
    if len(s) < 3:  
        print('Слишком мало')  
        continue  
    print('Введённая строка достаточной длины')
```

Функции языка Python

Функции – это многократно используемые фрагменты программы. Они позволяют дать имя определённому блоку команд с тем, чтобы впоследствии запускать этот блок по указанному имени в любом месте программы и сколь угодно много раз. Это называется вызовом функции.

Для объявления функции используют ключевое слово `def` (от англ. `define` — определить, обозначить). В общем виде объявление выглядит следующим образом:

```
def имя_функции (аргументы):  
    тело_функции  
    return результат  
Здесь:
```

имя_функции — название, с помощью которого можно вызывать функцию в коде;
аргументы — значения, которые функция принимает на вход. Это поле может быть пустым;

тело_функции — набор инструкций, которые выполняются при вызове;

результат — значения, которые функция возвращает при завершении работы.

Чтобы вызвать функцию, надо ввести её название и передать аргументы в скобках. В общем виде синтаксис вызова выглядит так:

имя_функции(аргументы)

Область видимости функций

В Python область видимости определяет, где и какие переменные можно использовать внутри функции. Есть нескольких видов переменных.

Локальная область (local scope)

Внутри функции можно объявить временные переменные, которые помогают в промежуточных вычислениях. Они существуют только внутри тела функции, их нельзя использовать в других местах проекта. Это и есть локальная область видимости.

Область объемлющей функции (enclosing function scope)

Функции бывают вложенными, когда одна находится внутри другой как матрёшка. В таком случае у внутренней функции есть доступ к переменным, определённым во внешней. Наоборот, это правило не будет работать.

Глобальная область (global scope)

Переменные, определённые вне функций, находятся в глобальной области видимости. Это значит, что они видны во всей программе и доступны всем функциям. Если надо изменить значение глобальной функции внутри функции, то необходимо использовать ключевое слово `global`.

Аргументы функций

В Python функции могут работать с разными типами аргументов. Это делает язык гибким и позволяет решать разные задачи.

Позиционные

Этот тип встречается чаще всего. Значения передаются в функцию в том порядке, в каком указано в функции. Иначе возникнет ошибка. Если функция принимает несколько аргументов, нужно разделить их запятой:

Именованные

Если использовать имена параметров при вызове функции, то можно передавать аргументы в произвольном порядке.

*Аргументы переменной длины (*args и **kwargs)*

Python позволяет определять функции, даже если мы не знаем, сколько аргументов она должна принимать:

`*args` используют, когда неясно, сколько позиционных аргументов у нас есть. Звёздочка `*` перед `args` указывает на то, что все позиционные аргументы, переданные при вызове функции, должны быть собраны в кортеж `tuple` и присвоены `args`.

`**kwargs` используют, чтобы передать именованные аргументы в виде словаря (`dictionary`), когда мы не знаем, сколько их у нас. Две звёздочки (`**`) перед `kwargs` означают, что все именованные аргументы должны быть собраны в словарь и присвоены `kwargs`.

Необязательные аргументы (параметры по умолчанию)

Иногда в функциях задают значения по умолчанию. Функция будет использовать их, если

не указано, что нужно использовать другой аргумент.

Например, если бы книжный магазин был интернет-магазином, каждый посетитель получал бы одинаковое приветственное сообщение. Но авторизированные пользователи, то есть те, кто передал свои данные магазину, получают персональное приветствие.

Возвращаемые значения (return)

В Python ключевое слово `return` используется для возврата значения из функции. Полученное значение можно дальше использовать в программе.

Если не использовать в функции ключевое слово `return`, она по умолчанию вернёт `None`.

Lambda-функции

Лямбда-функции (lambda-функции) – это безымянные функции, которые могут быть определены в одной строке кода. Лямбда-функцию нельзя переиспользовать – у неё нет имени, по которому её можно вызвать. Обычно их используют там, где требуется передать небольшую функцию в качестве аргумента.

Для определения лямбда-функций используют ключевое слово `lambda`. После него перечисляют параметры, а после двоеточия указывают выражение, результат которого функция вернёт. В общем плане это выглядит так:

`lambda` аргументы: выражение

Пример.

В качестве примера рассмотрим лямбда-функцию, которая проверяет чётность числа и выводит `True` или `False`. В качестве параметра передаётся переменная `num`, а после двоеточия указано выражение `num % 2 == 0`:

```
is_even = lambda num: num % 2 == 0
print(is_even(4)) # Вывод: True
print(is_even(7)) # Вывод: False
```

Задание для выполнения

1. Написать программу на языке Python в зависимости от варианта. Теме «Ввод и вывод данных в Python.»

Таблица 1. Варианты заданий.

Вариант	Задание
1	Запишите 10 целых чисел в файл <code>numbers.txt</code> по одному в строке. Прочитайте все числа из файла и выведите их сумму.
2	Предварительно подготовьте файл <code>input.txt</code> с следующим содержимым: “Привет, Мир!” Считайте содержимое файла <code>input.txt</code> и запишите его в новый файл <code>output.txt</code> .
3	Предварительно подготовьте файл <code>log.txt</code> с следующим содержимым: «У лукоморья дуб зелёный; Златая цепь на дубе том: И днём и ночью кот учёный Всё ходит по цепи кругом; Идёт направо — песнь заводит, Налево — сказку говорит.»

	Пользователь вводит ключевое слово. Из файла log.txt выберите все строки, содержащие это слово, и сохраните их в filtered_log.txt.
4	Запросите у пользователя 5 строк текста. Сохраните их в файл user_input.txt, записывая каждую строку на новой строке.
5	Предварительно подготовьте файл unsorted.txt с следующим содержимым: «Мороз и солнце; день чудесный! Еще ты дремлешь, друг прелестный — Пора, красавица, проснись: Открой сомкнуты негой взоры Навстречу северной Авроры, Звездой севера явись!» Посчитайте строки из файла unsorted.txt, отсортируйте их по алфавиту и запишите результат в sorted.txt.
6	Пользователь вводит строку с числами через пробел (например, 3 5 8 13). Выведите сумму чисел и среднее арифметическое.
7	Попросите пользователя ввести имя и число покупок. Выведите сообщение вида: “Ирина, спасибо за 5 покупок!”
8	Пользователь вводит температуру в градусах Цельсия. Выведите температуру в градусах Фаренгейта.
9	Попросите пользователя ввести предложение. Разделите его на слова и выведите каждое слово на новой строке.
10	Пользователь вводит имя, возраст и город. Выведите данные в виде таблицы: Имя Возраст Город ----- Анна 22 Москва

2. Написать программу на языке Python в зависимости от варианта. Теме «Операторы в языке Python»

Таблица 2. Варианты заданий.

Вариант	Задание
1	Пользователь вводит два числа. Выведите, какое из них больше, или что они равны.
2	Попросите пользователя ввести свой возраст. Если ему 18 или больше, выведите "Доступ разрешён", иначе — "Доступ запрещён".
3	Пользователь вводит два числа. Найдите остаток от деления первого на второе и выведите результат.
4	Пользователь вводит два числа и оператор (+, -, *, /). Выполните соответствующую операцию с числами. Подсказка: используйте if-elif-else.
5	Проверьте, делится ли введённое число на 3 и на 5 одновременно. Используйте логический оператор and.

6	Пользователь вводит возраст и гражданство. Если возраст больше 17 и гражданство "Россия", выведите "Можно голосовать", иначе — "Нельзя голосовать".
7	С клавиатуры вводится целое число. Вывести на экран цифры из которых оно состоит.
8	Найти сумму первой и последней цифр любого целого положительного числа.
9	Вводится число. Преобразовать его в другое число, цифры которого будут следовать в обратном порядке по сравнению с введенным числом.
10	Вывести на экран кубы чисел от А до В, которые вводит пользователь.

3. Написать программу на языке Python в зависимости от варианта. Теме «Функции в языке Python»

Таблица 3. Варианты заданий.

Вариант	Задание
1	Создайте функцию <code>greet(name)</code> , которая принимает имя и выводит: "Привет, <имя>!" Вызовите её с именем, введенным пользователем.
2	Напишите функцию <code>square(num)</code> , которая возвращает квадрат числа. Выведите результат вызова этой функции.
3	Напишите функцию <code>total(a, b, c)</code> , которая возвращает сумму трёх чисел. Проверьте её работу с пользовательским вводом.
4	Создайте функцию <code>is_even(number)</code> , которая возвращает <code>True</code> , если число чётное, и <code>False</code> — если нет. Выведите результат для введенного числа.
5	Напишите функцию <code>factorial(n)</code> , которая возвращает факториал числа <code>n</code> . Не используйте встроенные модули.
6	Создайте функцию <code>print_info(name, age=18)</code> , которая выводит имя и возраст. Попробуйте вызвать её с одним и двумя аргументами.
7	Напишите функцию <code>sum_to_n(n)</code> , которая возвращает сумму всех чисел от 1 до <code>n</code> . Вызовите её и выведите результат.
8	Создайте функцию <code>find_max(lst)</code> , которая принимает список чисел и возвращает наибольшее значение. Проверьте работу функции с вводом пользователя.
9	Создайте глобальную переменную <code>x</code> . Напишите функцию, которая изменяет <code>x</code> внутри себя. Покажите разницу между использованием и неиспользованием ключевого слова <code>global</code> .
10	Напишите рекурсивную функцию <code>reverse_print(s)</code> , которая выводит символы строки в обратном порядке по одному символу в строке.

Литература

1. Python Lessons. Русскоязычная документация по Python.
<https://pylessons.readthedocs.io/ru/latest/>
2. Функция print() в Python: аргументы и применение
<https://skillbox.ru/media/code/funkciya-print-v-python-argumenty-i-primenenie/>
3. Функции в Python: исчерпывающий гайд <https://skillbox.ru/media/code/funksii-v-python-dlya-chego-oni-nuzhny-i-kak-s-nimi-rabotat/>

Практическое занятие 23 и 24

Тема: Объектно-ориентированное программирование. Объектно-ориентированное программирование на языке Python.

Цель работы

Изучить основные принципы объектно-ориентированного программирования на примере языка Python.

Методические указания

Основные принципы объектно-ориентированного программирования:

- Абстракция
- Инкапсуляция
- Наследование
- Полиморфизм

Абстракция — это использование только значимых характеристик объекта и игнорирование всего остального. Чем меньше характеристик, тем лучше абстракция, но ключевые характеристики убирать нельзя.

Класс в ООП — это абстракция, благодаря которой вы можете объявлять объекты внутри программы, не описывая их подробно.

Инкапсуляция — объединение и скрытие методов и свойств и предоставление доступа к ним через простой внешний интерфейс.

Вся информация, которая нужна для работы конкретного объекта, должна храниться внутри этого объекта. Если нужно вносить изменения, методы для этого тоже должны лежать в самом объекте — посторонние объекты и классы этого делать не могут. Для внешних объектов доступны только публичные атрибуты и методы.

Наследование — возможность описать новый класс на базе существующего. При этом дочерние классы могут заимствовать свойства и методы родительского класса.

В этом принципе — вся суть объектно-ориентированного программирования. Разработчик создаёт:

- Класс с определёнными свойствами;
- Подкласс на его основе, который берёт свойства класса и добавляет свои;
- Объект подкласса, который также копирует его свойства и добавляет свои.

Полиморфизм — возможность взаимодействовать с объектами разных классов через одинаковые интерфейсы; обращаться к свойствам и методам, общим для всех объектов.

Создание классов:

Для создания классов предусмотрена инструкция **class**. Это составная инструкция, которая состоит из строки заголовка и тела. Заголовок состоит из ключевого слова **class**, имени класса и, возможно, названий суперклассов в скобках. Суперклассов может и не быть, в таком случае скобки не требуются. Тело класса состоит из блока различных инструкций. Тело должно иметь отступ (как и любые вложенные конструкции в языке Python).

Схематично класс можно представить следующим образом:

```
class ИМЯКЛАССА:
```

```
    ПЕРЕМЕННАЯ = ЗНАЧЕНИЕ
```

```

....
def ИМЯМЕТОДА(self, ...):
    self.ПЕРЕМЕННАЯ = ЗНАЧЕНИЕ
    ...
....

```

Данная схема не является полной. Например, в заголовке после имени класса могут быть указаны суперклассы (в скобках), а методы могут быть более сложными.

Следует помнить, что **методы** в классах — это те же функции, за одним небольшим исключением. Они принимают один обязательный параметр — `self` (с англ. можно перевести как "собственная личность"). Он нужен для связи с конкретным объектом.

Атрибуты класса — это имена переменных вне функций и имена функций. Эти атрибуты наследуются всеми объектами, созданными на основе данного класса. Атрибуты обеспечивают свойства и поведение объекта. Объекты могут иметь атрибуты, которые создаются в теле метода, если данный метод будет вызван для конкретного объекта.

Создание объектов:

Объекты создаются так:

```
ПЕРЕМЕННАЯ = ИМЯКЛАССА()
```

Здесь скобки обязательны! После такой инструкции в программе появляется объект, доступ к которому можно получить по имени переменной, связанной с ним. При создании объект получает атрибуты его класса, т. е. объекты обладают характеристиками, определенными в их классах.

Количество объектов, которые можно создать на основе того или иного класса, не ограничено.

Объекты одного класса имеют схожий набор атрибутов, а вот значения атрибутов могут быть разными. Другими словами, объекты одного класса похожи, но индивидуально различимы. Чтобы понять это, можно сравнить отношения объектов одного класса в программировании со следующим высказыванием: "Все млекопитающие принадлежат одному классу и обычно имеют по два глаза, однако у каждого животного (объекта) глаза имеют свои особенности".

Self:

Можно сказать, что методы класса — это небольшие программки, предназначенные для работы с объектами. Методы могут создавать новые свойства (данные) объекта, изменять существующие, выполнять другие действия над объектами.

Методу необходимо "знать", данные какого объекта ему предстоит обрабатывать. Для этого ему в качестве первого (а иногда и единственного) аргумента передается имя переменной, связанной с объектом (можно сказать, передается сам объект). Чтобы в описании класса указать передаваемый в дальнейшем объект, используется параметр `self`.

С другой стороны, вызов метода для конкретного объекта в основном блоке программы выглядит следующим образом:

```
ОБЪЕКТ.ИМЯМЕТОДА(...)
```

Здесь под словом **ОБЪЕКТ** имеется в виду переменная, связанная с ним. Это выражение преобразуется в классе, к которому относится объект, в

```
ИМЯМЕТОДА(ОБЪЕКТ, ...)
```

Т. е. конкретный объект подставляется вместо параметра `self`.

Первая ОО-программа:

Попробуем на основе имеющихся уже знаний написать небольшую ОО-программу. Допустим, это будет класс с одним атрибутом вне метода и одним методом, который выводит с небольшим изменением значение этого атрибута на экран:

```
class First:
    color = "red"
    def out(self):
        print (self.color + "!")
```

Теперь создадим пару объектов данного класса:

```
obj1 = First()
obj2 = First()
```

Оба этих объекта (obj1 и obj2) имеют два одинаковых атрибута: color (в виде свойства) и printer (в виде метода). Это легко проверить:

```
print (obj1.color)
print (obj2.color)
obj1.out()
obj2.out()
```

В результате выполнения данного скрипта получается вывод двух надписей red и двух red!. Первые две надписи red – это результат применения встроенной функции print по отношению к свойствам объектов. Вторые две надписи red! - результат применения метода out к объектам.

Усложняем программу:

В предыдущей программе оба созданных объекта абсолютно одинаковы. Класс, на основе которого они созданы слишком прост и не предполагает того, что объекты могут иметь различные значения свойств. Исправим это.

Пусть теперь в классе с помощью атрибутов вне функции устанавливаются два свойства объектов: красный цвет и круглая форма. А методы могут менять эти свойства в зависимости от пожеланий тех, кто создает объекты.

```
class Second:
    color = "red"
    form = "circle"
    def changecolor(self,newcolor):
        self.color = newcolor
    def changeform(self,newform):
        self.form = newform

obj1 = Second()
obj2 = Second()

print (obj1.color, obj1.form) # вывод на экран "red circle"
print (obj2.color, obj2.form) # вывод на экран "red circle"

obj1.changecolor("green") # изменение цвета первого объекта
obj2.changecolor("blue") # изменение цвет второго объекта
obj2.changeform("oval") # изменение формы второго объекта

print (obj1.color, obj1.form) # вывод на экран "green circle"
print (obj2.color, obj2.form) # вывод на экран "blue oval"
```

В данной программе по умолчанию любой созданный объект имеет красный цвет и круглую форму. Однако в дальнейшем с помощью методов данного класса можно поменять и цвет и форму любого объекта. В результате объекты перестают быть одинаковыми (красными и

круглыми), хотя сохраняют тот же набор свойств (цвет и форму).

Как же происходят изменения? Дело в том, что методы помимо параметра `self`, могут иметь и другие параметры, в которых передаются данные для обработки их этим методом. Так, в примере выше, метод `changecolor` имеет дополнительный параметр `newcolor`, с помощью которого, в метод можно передать данные о желаемом цвете фигуры. Далее метод меняет цвет с помощью соответствующих инструкций.

Конструктор класса — метод `__init__`

Большинство классов имеют специальный метод, который автоматически при создании объекта создает ему атрибуты. Т.е. вызывать данный метод не нужно, т.к. он сам запускается при вызове класса. (Вызов класса происходит, когда создается объект.) Такой метод называется **конструктором класса** и в языке программирования Python носит имя `__init__`. (В начале и конце по два знака подчеркивания.)

Первым параметром, как и у любого другого метода, у `__init__` является `self`, на место которого подставляется объект в момент его создания.

Второй и последующие (если есть) параметры заменяются аргументами, переданными в конструктор при вызове класса.

Рассмотрим два класса: в одном будет использоваться конструктор, а в другом нет. Требуется создать два атрибута объекта.

```
class YesInit:
    def __init__(self,one,two):
        self.fname = one
        self.sname = two
```

```
obj1 = YesInit("Peter","Ok")
```

```
print (obj1.fname, obj1.sname)
```

```
class NoInit:
    def names(self,one,two):
        self.fname = one
        self.sname = two
```

```
obj1 = YesInit()
obj1.names("Peter","Ok")
```

```
print (obj1.fname, obj1.sname)
```

Вывод интерпретатора в обоих случаях:

Peter Ok

В обоих программах у объекта появляются два атрибута: `fname` и `sname`. Однако в первом случае они инициализируются при создании объекта и должны передаваться в скобках при вызове класса. Если какие-то атрибуты должны присутствовать у объектов класса обязательно, то использование метода `__init__` - идеальный вариант. Во второй программе (без использования конструктора) атрибуты создаются путем вызова метода `names` после создания объекта. В данном случае вызов метода `names` необязателен, поэтому объекты могут существовать без атрибутов `fname` и `sname`.

Обычно метод `__init__` предполагает передачу аргументов при создании объектов, однако аргумент может не быть передан. Например, если в примере выше создать объект так: `obj1 = YesInit()`, т.е. не передать классу аргументы, то произойдет ошибка. Чтобы избежать подобных ситуаций, можно в методе `__init__` присваивать параметрам значения по умолчанию. Если при вызове класса были заданы аргументы для данных параметров, то хорошо — они и будут использоваться, если нет — еще лучше — в теле метода будут использованы значения по умолчанию. Пример:

```
class YesInit:
    def __init__(self,one="noname",two="nonametoo"):
        self.fname = one
        self.sname = two
```

```
obj1 = YesInit("Sasha","Tu")
obj2 = YesInit()
obj3 = YesInit("Spartak")
obj4 = YesInit(two="Harry")
```

```
print (obj1.fname, obj1.sname)
print (obj2.fname, obj2.sname)
print (obj3.fname, obj3.sname)
print (obj4.fname, obj4.sname)
```

Вывод интерпретатора:

```
Sasha Tu
noname nonametoo
Spartak nonametoo
noname Harry
```

В данном случае, второй объект создается без передачи аргументов, поэтому в методе `__init__` используются значения по умолчанию ("noname" и "nonametoo"). При создании третьего и четвертого объектов передаются по одному аргументу. Если указывается значение не первого аргумента, то следует явно указать имя параметра (четвертый объект). Метод `__init__` может содержать параметры как без значений по умолчанию, так и со значениями по умолчанию. В таком случае, параметры, аргументы которых должны быть обязательно указаны при создании объектов, указываются первыми, а параметры со значениями по умолчанию — после. Например, ниже вторая программа с ошибкой:

```
class fruits:
    def __init__(self,w,n=0):
        self.what = w
        self.numbers = n
```

```
f1 = fruits("apple",150)
f2 = fruits("pineapple")
```

```
print (f1.what,f1.numbers)
print (f2.what,f2.numbers)
```

```
class fruits:
    def __init__(self,n=0,w): #ERROR
        self.what = w
        self.numbers = n
```

```
f1 = fruits(150,"apple")
f2 = fruits("pineapple")

print (f1.what,f1.numbers)
print (f2.what,f2.numbers)
```

Наследование в ООП на Python

Одной из важнейших особенностей ООП является возможность **наследования** объектами атрибутов классов, а также наследование одними классами атрибутов других классов. На самом деле с наследованием мы уже сталкивались, когда создавали любой объект в Python: объекты наследуют атрибуты класса, хотя могут иметь и индивидуальные.

Атрибуты класса наследуются объектами, созданными на его основе; однако атрибуты конкретного объекта не зависят от атрибутов других объектов и представляют собственное пространство имен объекта. Последнее позволяет объектам одного класса иметь различные значения атрибутов, а если потребуется и различный набор атрибутов.

На самом деле, наследование более широкое понятие, чем просто взаимосвязь между классами и объектами. Один класс может быть подклассом другого, дополняя его. Пояснить это можно, проведя аналогию с реальным миром. Например, все столы имеют общие характерные черты («класс»), при этом они имеют разное назначение («подклассы»), хотя продолжают наследовать общие черты. В результате того, что есть такой механизм как наследование можно избежать избыточность кода, просто описав общие свойства и методы в надклассах.

Классы, атрибуты которых наследуются другими классами, могут называть как **надклассами** так и **суперклассами**.

Классы, которые наследуют атрибуты других классов, часто называют **подклассами**.

Класс, являющийся надклассом по отношению к одному классу, сам может быть подклассом по отношению к другому. Другими словами, может существовать целая цепочка наследования.

При обращении к атрибуту объекта (obj.prop) сначала просматривается на наличие этого атрибута сам объект, затем его класс, на основе которого он создан. Если в классе не будет найден атрибут, то его поиск продолжится в суперклассе, к которому относится класс.

Суперклассы класса указываются в скобках в заголовке инструкции class.

Рассмотрим такой пример:

```
class Table:
    def __init__(self,l,w,h):
        self.long = l
        self.width = w
        self.height = h
    def outing(self):
        print (self.long,self.width,self.height)

class Kitchen(Table):
    def howplaces(self,n):
        if n < 2:
```

```

        print ("It is not kitchen table")
    else:
        self.places = n
def outplaces(self):
    print (self.places)

```

```

t_room1 = Kitchen(2,1,0.5)
t_room1.outing()
t_room1.howplaces(5)
t_room1.outplaces()

```

```

t_2 = Table(1,3,0.7)
t_2.outing()
t_2.howplaces(8) # ОШИБКА

```

Здесь создается два класса: Table и Kitchen. Второй является подклассом первого и наследует все его атрибуты (методы `__init__` и `outing`).

Далее создаются два объекта: `t_room1` и `t_2`. Первый объект принадлежит к классу Kitchen и наследует атрибуты этого класса и его суперкласса. Второй объект принадлежит классу Table; к классу Kitchen он никакого отношения не имеет и поэтому не может обращаться к методам `howplaces` и `outplaces`. В данном примере также можно увидеть, что объекты можно создавать как на основе классов, так и суперклассов.

Класс может иметь не один, а несколько суперклассов, которые перечисляются друг за другом в скобках в строке заголовка. Такое **наследование** называется **множественным**. Потребность во множественном наследовании возникает в случае, если объекты класса предполагают использование свойств и методов различных суперклассов.

Полиморфизм и переопределение методов в ООП на Python

Слово «полиморфизм» можно перевести как «много форм». В ОО программировании этим термином обозначают возможность использования одного и того же имени операции или метода к объектам разных классов, при этом действия, совершаемые с объектами, могут существенно различаться. Поэтому можно сказать, что у одного и того же слова много форм. Например, два разных класса могут содержать метод `total`, однако инструкции в методах могут предусматривать совершенно разные операции: так в классе T1 – это прибавление 10 к аргументу, а в T2 – подсчет длины строки символов. В зависимости от того, к объекту какого класса применяется метод `total`, выполняются те или иные инструкции.

```

class T1:
    n=10
    def total(self,N):
        self.total = int(self.n) + int(N)

```

```

class T2:
    def total(self,s):
        self.total = len(str(s))

```

```

t1 = T1()
t2 = T2()

```

```

t1.total(45)
t2.total(45)
print (t1.total) # Вывод: 55
print (t2.total) # Вывод: 2

```

Использование полиморфизма при наследовании классов позволяет переопределять методы суперклассов их подклассами. Например, может возникнуть ситуация, когда все подклассы реализуют определенный метод из суперкласса, и лишь один подкласс должен иметь его другую реализацию. В таком случае метод переопределяется в подклассе.

Пример:

```

class Base:
    def __init__(self,n):
        self.numb = n
    def out(self):
        print (self.numb)

class One(Base):
    def multi(self,m):
        self.numb *= m

class Two(Base):
    def inlist(self):
        self.inlist = list(str(self.numb))
    def out(self):
        i = 0
        while i < len(self.inlist):
            print (self.inlist[i])
            i += 1

obj1 = One(45)
obj2 = Two('abc')

obj1.multi(2)
obj1.out() # Вывод числа 90

obj2.inlist()
obj2.out() # Вывод в столбик букв a, b, c

```

В данном случае объект obj1 использует метод out из суперкласса Base, а obj2 – из своего класса Two. Атрибуты ищутся «снизу вверх»: сначала в классах, затем суперклассах. Поскольку для obj2 атрибут out уже был найден в классе Two, то из класса Base он не используется.

Другими словами, класс Two переопределяет атрибут суперкласса Base.

Расширение методов

При ООП может возникнуть ситуация, когда метод суперкласса в принципе подходит для реализации того или иного действия с объектами класса, однако требует небольших изменений. В таком случае можно использовать так называемое расширение метода, когда из тела метода класса вызывается метод суперкласса и дописываются дополнительные инструкции. В примере ниже в методе класса Subclass вызывается метод другого класса (в данном случае его суперкласса; однако может вызываться метод, не принадлежащий

собственному суперклассу):

```
class Base:
    def __init__(self,N):
        self.numb = N
    def out(self):
        self.numb /= 2
        print (self.numb)
```

```
class Subclass(Base):
    def out(self):
        print ("\n----")
        Base.out(self)
        print ("----\n")
```

```
i = 0
while i < 10:
    if 4 < i < 7:
        obj = Subclass(i)
    else:
        obj = Base(i)
    i += 1
    obj.out()
```

Полиморфизм в объектно-ориентированном программировании дает возможность реализовывать так называемые единые интерфейсы для объектов различных классов. Имеется ввиду, что если есть методы с одинаковыми названиями для всех объектов, то это позволяет писать более очевидный исходный код.

Переопределение методов в подклассах (а также их расширение) позволяет специализировать ранее написанный исходный код, не меняя его в суперклассах, где обычно требуется оставить код в неизменном виде для других подклассов.

Магические методы в Python

Магические методы в Python, также известные как **dunder**-методы (от английского "double underscore"), представляют собой специальные методы, которые начинаются и заканчиваются двумя подчеркиваниями (например, `__init__`, `__str__`). Эти методы позволяют вам определять поведение ваших объектов при использовании встроенных функций и операторов. Магические методы делают ваш код более читаемым и позволяют использовать объекты классов так, как если бы они были встроенными типами данных. Например, вы можете использовать магические методы для создания объектов, которые ведут себя как списки или словари, или для определения поведения объектов при выполнении арифметических операций.

Магические методы играют ключевую роль в Python, так как они позволяют разработчикам создавать более интуитивно понятные и гибкие классы. Они позволяют вам переопределять стандартное поведение объектов, что может быть особенно полезно при создании пользовательских структур данных или при работе с объектно-ориентированным программированием. В этой статье мы рассмотрим основные магические методы, их назначение и примеры использования, чтобы вы могли лучше понять, как и когда их использовать.

Создание и удаление объектов

Любое описание объекта в объектно-ориентированном программировании начинается с создания объекта и его удаления. Давайте подробнее остановимся на каждом из них:

`__new__(cls[, ...])` — метод создания типа класса. Он принимает первым аргументом тип класса, в котором он вызывается, и, как правило, возвращает этот же тип. В основном используется, чтобы настраивать создание экземпляра класса тех объектов, которые наследуются от неизменяемых типов (например, `int`, `str`, или `tuple`).

`__init__(self[, ...])` — конструктор класса. Используется при определении объектов.

`__init_subclass__(cls)` — позволяет переопределить создание подклассов объекта. Например, добавлять дополнительные атрибуты:

```
class Test:
    def __init_subclass__(cls, /, test_param, **kwargs):
        super().__init_subclass__(**kwargs)
        cls.test_param = test_param
class AnotherTest(Test, test_param="Hello World"):
    pass
```

`__del__(self)` — деструктор класса. Вызывается автоматически сборщиком мусора, практически никогда не используется, за исключением, когда пользователя необходимо предупредить о незакрытых дескрипторах.

Общие свойства объектов

Любой объект может содержать дополнительную информацию, полезную при отладке или приведении типов. Например:

`__repr__(self)` — информационная строка об объекте. Выводится при вызове функции `repr(...)` или в момент отладки. Для последнего этот метод и предназначен.

`__str__(self)` — вызывается при вызове функции `str(...)`, возвращает строковый объект.

`__bytes__(self)` — аналогично `__str__(self)`, только возвращается набор байт.

`__format__(self, format_spec)` — вызывается при вызове функции `format(...)` и используется для форматирования строки с использованием строковых литералов.

Методы сравнения объектов между собой

`__lt__(self, other)` — определяет поведение оператора сравнения «меньше», `<`.

`__le__(self, other)` — определяет поведение оператора сравнения «меньше или равно», `<=`.

`__eq__(self, other)` — определяет поведение оператора «равенства», `==`.

`__ne__(self, other)` — определяет поведение оператора «неравенства», `!=`.

`__gt__(self, other)` — определяет поведение оператора сравнения «больше», `>`.

`__ge__(self, other)` — определяет поведение оператора сравнения «больше или равно», `>=`.

`__hash__(self)` — вызывается функцией `hash(...)` и используется для определения контрольной суммы объекта, чтобы доказать его уникальность. Например, чтобы добавить объект в `set`, `frozenset`, или использовать в качестве ключа в словаре `dict`.

`__bool__(self)` — вызывается функцией `bool(...)` и возвращает `True` или `False` в соответствии с реализацией. Если данный метод не реализован в объекте, и объект является какой-либо последовательностью (списком, кортежем и т.д.), вместо него вызывается метод `__len__`.

Доступ к атрибутам объекта

Доступ ко всем свойствам объекта также контролируются отдельными методами:

`__getattr__(self, name)` — вызывается методом `getattr(...)` или при обращении к атрибуту объекта через `x.y`, где `x` — объект, а `y` — атрибут.

`__setattr__(self, name, value)` — вызывается методом `setattr(...)` или при обращении к атрибуту объекта с последующим определением значения переданного атрибута. Например: `x.y = 1`, где `x` — объект, `y` — атрибут, а `1` — значение атрибута.

`__delattr__(self, name)` — вызывается методом `delattr(...)` или при ручном удалении атрибута у объекта с помощью `del x.y`, где `x` — объект, а `y` — атрибут.

`__dir__(self)` — вызывается методом `dir(...)` и выводит список доступных атрибутов объекта.

Создание последовательностей

Любой объект может реализовать методы встроенных последовательностей (словари, кортежи, списки, строки и так далее). Доступ к значениям последовательности переопределяется следующими методами:

`__len__(self)` — вызывается методом `len(...)` и возвращает количество элементов в последовательности.

`__getitem__(self, key)` — вызывается при обращении к элементу в последовательности по его ключу (индексу). Метод должен выбрасывать исключение `TypeError`, если используется некорректный тип ключа, `KeyError`, если данному ключу не соответствует ни один элемент в последовательности.

`__setitem__(self, key, value)` — вызывается при присваивании какого-либо значения элементу в последовательности. Также может выбрасывать исключения `TypeError` и `KeyError`.

`__delitem__(self, key)` — вызывается при удалении значения в последовательности по его индексу (ключу) с помощью синтаксиса ключевого слова `del`.

`__missing__(self, key)` — вызывается в случаях, когда значения в последовательности не существует.

`__iter__(self)` — вызывается методом `iter(...)` и возвращает итератор последовательности, например, для использования объекта в цикле.

`__reversed__(self)` — вызывается методом `reversed(...)` и аналогично методу `__iter__` возвращает тот же итератор, только в обратном порядке.

`__contains__(self, item)` — вызывается при проверке принадлежности элемента к последовательности с помощью `in` или `not in`.

Числовые магические методы

Данные методы делятся на несколько групп: унарные операторы, обычные арифметические, отражённые арифметические операторы, составные присваивания и преобразования типов.

Унарные операторы

`__neg__(self)` — определяет поведение для отрицания (`-a`)

`__pos__(self)` — определяет поведение для унарного плюса (`+a`)

`__abs__(self)` — определяет поведение для встроенной функции `abs(...)`

`__invert__(self)` — определяет поведение для инвертирования оператором `~`

Обычные арифметические операторы

`__add__(self, other)` — сложение, оператор `+`

`__sub__(self, other)` — вычитание, оператор `-`

`__mul__(self, other)` — умножение, оператор `*`

`__matmul__(self, other)` — умножение матриц, оператор `@`

`__truediv__(self, other)` — деление, оператор `/`

`__floordiv__(self, other)` — целочисленное деление, оператор `//`
`__mod__(self, other)` — остаток от деления, оператор `%`
`__divmod__(self, other)` — деление с остатком, определяет поведение для встроенной функции `divmod(...)`
`__pow__(self, other[, modulo])` — возведение в степень, оператор `**`
`__lshift__(self, other)` — двоичный сдвиг влево, оператор `<<`
`__rshift__(self, other)` — двоичный сдвиг вправо, оператор `>>`
`__and__(self, other)` — двоичное И, оператор `&`
`__xor__(self, other)` — исключающее ИЛИ, оператор `^`
`__or__(self, other)` — двоичное ИЛИ, оператор `|`

Отражённые арифметические операторы

Если в обычной арифметике между объектами `a` и `b`, объектом, который мы изменяем, является `a`, и объектом, с которым мы работаем, является `b`, то в отражённой арифметике наоборот — `b` является изменяемым, `a` — объектом, с которым мы работаем, и который передается в качестве аргумента. Например:

```
# Сложение, используется обычная арифметика и метод __add__ в объекте left:
left + right
# Сложение, используется отражённая арифметика и метод __radd__ в объекте left:
right + left
```

Список методов похож на тот, что используется в обычной арифметике, за исключением того, что добавляется префикс «r» ко всем методам:

`__radd__(self, other)` — сложение, оператор `+`
`__rsub__(self, other)` — вычитание, оператор `-`
`__rmul__(self, other)` — умножение, оператор `*`
`__rmatmul__(self, other)` — умножение матриц, оператор `@`
`__rtruediv__(self, other)` — деление, оператор `/`
`__rfloordiv__(self, other)` — целочисленное деление, оператор `//`
`__rmod__(self, other)` — остаток от деления, оператор `%`
`__rdivmod__(self, other)` — деление с остатком
`__rpow__(self, other[, modulo])` — возведение в степень, оператор `**`
`__rlshift__(self, other)` — двоичный сдвиг влево, оператор `<<`
`__rrshift__(self, other)` — двоичный сдвиг вправо, оператор `>>`
`__rand__(self, other)` — двоичное И, оператор `&`
`__rxor__(self, other)` — исключающее ИЛИ, оператор `^`
`__ror__(self, other)` — двоичное ИЛИ, оператор `|`

Составное присваивание

Эти методы — комбинация «обычного» оператора и присваивания. Возвращают тот же тип объекта, который будет присвоен переменной слева. Например:

```
x = 3
x += 2 # другими словами x = x + 2
```

`__iadd__(self, other)` — сложение с присваиванием, оператор `+=`

`__isub__(self, other)` — вычитание с присваиванием, оператор `--`
`__imul__(self, other)` — умножение с присваиванием, оператор `*=`
`__imatmul__(self, other)` — умножение матриц с присваиванием, оператор `@=`
`__itruediv__(self, other)` — деление с присваиванием, оператор `/=`
`__ifloordiv__(self, other)` — целочисленное деление с присваиванием, оператор `//=`
`__imod__(self, other)` — остаток от деления с присваиванием, оператор `%=`
`__ipow__(self, other[, modulo])` — возведение в степень с присваиванием, оператор `**=`
`__lshift__(self, other)` — двоичный сдвиг влево с присваиванием, оператор `<<=`
`__rshift__(self, other)` — двоичный сдвиг вправо с присваиванием, оператор `>>=`
`__iand__(self, other)` — двоичное И с присваиванием, оператор `&=`
`__ixor__(self, other)` — исключающее ИЛИ с присваиванием, оператор `^=`
`__ior__(self, other)` — двоичное ИЛИ с присваиванием, оператор `|=`

Преобразования типов

Помимо всего прочего, в Python множество методов, которые позволяют переопределять поведение встроенных функций преобразования типов, таких как `int(...)`, `float(...)` и т.д. Например:

`__complex__(self)` — преобразование типа в комплексное число
`__int__(self)` — преобразование типа к `int`
`__float__(self)` — преобразование типа к `float`
`__index__(self)` — преобразование типа к `int`, когда объект используется в срезах (выражения вида `[start:stop:step]`)
`__round__(self[, ndigits])` — округление числа с помощью функции `round(...)`
`__trunc__(self)` — вызывается методом `math.trunc(...)`
`__floor__(self)` — вызывается методом `math.floor(...)`
`__ceil__(self)` — вызывается методом `math.ceil(...)`

Вызываемые объекты

`__call__(self[, args...])` — позволяет любому экземпляру класса вести себя как обычная функция. Например:

```
class Test:
    def __call__(self, message):
        print(message)
        return True
```

```
test = Test()
test("Hello World")
... 'Hello World'
... True
```

`__await__(self)` — возвращает итератор, превращая класс в корутину, результат выполнения которой можно получить с помощью `await`. Подробнее об этом можно узнать в PEP 492.

Задание для выполнения

1. Написать программу на языке Python в зависимости от варианта.

Таблица 1. Варианты заданий.

Вариант	Задание
1	Создайте класс «Мебель» с полями «Марка», «Название», «Цена» и методом для вывода подробной информации о предмете. От класса «Мебель» необходимо унаследовать класс «Стол» с унаследованными полями класса «Мебель» и новыми полями «Спинка» (True/False), «Кол-во ножек» и методом для вывода подробной информации.
2	Создайте базовый класс «Транспортное средство» и производные классы «Автомобиль», «Велосипед», «Повозка». Подсчитайте время и стоимость перевозки пассажиров и грузов каждым транспортным средством.
3	Создайте базовый класс «Домашнее животное» и производные классы «Собака», «Кошка», «Попугай» и др. С помощью конструктора установите имя каждого животного и его характеристики.
4	Создайте абстрактный класс Shape для рисования плоских фигур. Необходимо построить производные классы Square (квадрат, который характеризуется координатами левого верхнего угла и длиной стороны), Circle (окружность с заданными координатами центра и радиусом), Ellipse (эллипс с заданными координатами вершин описанного вокруг него прямоугольника), позволяющие рисовать указанные фигуры, а также передвигать их на плоскости.
5	Создайте приложение, в котором необходимо разработать базовый класс Man. Объекты этого класса содержат справочную информацию о конкретном человеке (фамилию, инициалы, телефон, адрес, возраст). Создайте два производных от него класса: Manager и Secretary. Объекты класса Manager дополнительно включают номер отдела и количество подчиненных. Объекты класса Manager дополнительно включают номер отдела и количество подчиненных. Объекты класса Secretary дополнительно включают фамилию начальника. Данные о менеджерах и секретарях введите с клавиатуры и выведите на экран дисплея.
6	Разработайте класс Book: Автор, Название, Издательство, Год, Количество страниц. Создайте массив объектов. Выведите: а) список книг заданного автора; б) список книг, выпущенных заданным издательством; в) список книг, выпущенных после заданного года.
7	Разработайте класс Word: Слово, Номера страниц, на которых слово встречается (от 1 до 10), Число страниц. Создайте массив объектов. Выведите: а) слова, которые встречаются более чем на N страницах; б) слова в алфавитном порядке; в) для заданного слова номера страниц, на которых оно встречается.
8	Разработайте класс Равнобокая трапеция, члены класса – координаты 4-х точек. Предусмотрите в классе конструктор и методы: проверка, является ли фигура равнобокой трапецией; вычисление и вывод сведений о фигуре: длины сторон, периметр, площадь.

2. Написать программу на языке Python в зависимости от варианта.

Таблица 2. Варианты заданий.

Вариант	Задание
1	Создайте класс "Автомобиль" с системой управления состоянием: добавьте атрибут "текущий режим работы" (например, "заглушен", "в движении", "на ремонте"). Реализуйте методы для запуска двигателя, остановки, переключения режимов, а также автоматическую проверку допустимости перехода между режимами. Реализуйте обработку исключений при неправильных операциях.
2	Создайте класс "Студент" с системой оценки и аналитики: реализуйте возможность хранения оценок по разным предметам в виде словаря. Добавьте методы для вычисления среднего балла по каждому предмету и по всему списку оценок, а также метод для определения лучшего предмета.
3	Разработайте класс "Банк" с транзакционной системой: реализуйте возможность ведения нескольких счетов, переводов между ними, а также журнал транзакций с датой, суммой и типом операции. Обеспечьте безопасность операций (например, проверка баланса перед снятием). Реализуйте возможность восстановления состояния банка из файла.
4	Создайте иерархию классов "Животное", "Кошка", "Собака" с добавлением абстрактных методов и свойств: например, метод питание(), которое возвращает тип корма или поведение при голоде. Реализуйте интерфейс для взаимодействия с объектами через базовый класс или протокол.
5	Реализуйте класс "Книга" с системой поиска и фильтрации: создайте коллекцию книг внутри класса или отдельно — реализуйте методы поиска книг по автору, году издания, количеству страниц (например, все книги после определенного года). Добавьте возможность сортировки по различным критериям.
6	Создайте класс "Магазин" с динамическим управлением ассортиментом: реализуйте автоматическую систему обновления цен (например, случайное изменение цен), а также методы для поиска товаров по диапазону цен или популярности (количеству продаж). Введите обработку ошибок при добавлении/удалении товаров.
7	Напишите программу с наследованием и полиморфизмом для фигур: создайте базовый абстрактный класс Фигура с абстрактным методом вычислить_площадь(). Реализуйте подклассы Прямоугольник, Квадрат, Треугольник, Круг с переопределением метода. Добавьте возможность создания коллекции фигур разного типа и выполнения операций над всей коллекцией (например, подсчёт общей площади).
8	Реализуйте класс "Круг" с возможностью масштабирования и сравнения: добавьте методы для увеличения/уменьшения радиуса на заданный коэффициент. Реализуйте магические методы <code>__eq__</code> , <code>__lt__</code> , <code>__gt__</code> для сравнения двух объектов по площади. Создайте список объектов и выполните сортировку по площади.

Литература

1. Объектно-ориентированное программирование на Python
<https://younglinux.info/oopython/course>
2. Введение в объектно-ориентированное программирование (ООП) на Python
https://kabinet-vplaksina.narod.ru/olderfiles/5/Vvedenie_v_OOP_na_Python.pdf
3. Мачуева Д.А., Умарова Л.С. Методические указания к выполнению лабораторных работ по дисциплине «Объектно-ориентированное программирование»
[https://gstou.ru/sveden/files/Metodichka_OOP\(3\).pdf](https://gstou.ru/sveden/files/Metodichka_OOP(3).pdf)
4. Задорожный С.С., Фадеев Е.П. Объектно-ориентированное программирование на языке Python / М. : Физический факультет МГУ им. М. В. Ломоносова, 2022. – 40 с.
5. Знакомство с ООП
<https://javarush.com/quests/lectures/ru.javarush.python.core.lecture.level09.lecture01>
6. Магические методы в Python: что это и как использовать
<https://sky.pro/wiki/python/magicheskie-metody-v-python-cto-eto-i-kak-ispolzovat/>
7. Гайд по магическим методам в Python <https://tproger.ru/articles/gajd-po-magicheskim-metodam-v-python>

Практическое занятие 25 - 27

Тема: Объектно-ориентированное программирование. Объектно-ориентированное программирование на языке Python. Выполнение проекта.

Цель работы

Закрепить на практике основные принципы объектно-ориентированного программирования на примере языка Python.

Методические указания

Выполнение проекта производится в соответствии с техническим заданием. Необходимо разделиться на группы.

Задание для выполнения

Вариант 1.

Проект: Система учета студентов и их оценок

1. Общие сведения

Цель проекта

Разработать модульную, расширяемую систему на Python с использованием объектно-ориентированного подхода, которая позволит администраторам учебных заведений эффективно управлять данными о студентах и их успеваемости. Система должна обеспечивать возможность добавления, редактирования, удаления данных, а также предоставлять отчеты о среднем балле по студентам и курсам.

2. Требования к функциональности

2.1. Основные классы и их функции

Класс: Student

Атрибуты: student_id, name, surname, birth_date, group

Методы:

- Получение полной информации о студенте
- Обновление данных

Класс: Course

Атрибуты: course_id, name, teacher

Методы:

- Получение информации о курсе
- Обновление данных

Класс: Grade

Атрибуты: student (объект Student), course (объект Course), score, date

Методы:

- Получение информации о оценке
- Обновление оценки

Класс: StudentRecord

Хранит список всех студентов, курсов и оценок.

Методы:

- Добавление нового студента/курса/оценки
- Удаление студента/курса/оценки
- Поиск по студенту или курсу
- Расчет среднего балла по студенту или курсу

2.2. Дополнительные функции

- Импорт данных из файла (например, CSV)
- Экспорт данных в файл (CSV или JSON)
- Генерация отчетов:
- Средний балл по студенту
- Средний балл по курсу
- Топ-5 студентов по успеваемости

3. Требования к интерфейсу

- ❖ Реализация через консольное меню или командную строку.
- ❖ Возможность взаимодействия с системой через текстовые команды.

4. Технические требования

- ❖ Использование Python версии не ниже 3.8.
- ❖ Соблюдение принципов ООП.
- ❖ Код должен быть структурированным, с комментариями.
- ❖ Реализовать обработку ошибок (например, при вводе некорректных данных).

5. Дополнительные требования

- ❖ Возможность расширения системы для работы с базой данных (например, SQLite) в будущем.
- ❖ Написание документации по использованию системы.

Вариант 2

Проект: Игра «Крестики-нолики»

1. Общие сведения

Цель проекта: разработать консольную игру "Крестики-нолики" с использованием принципов объектно-ориентированного программирования, обеспечивающую удобство взаимодействия и расширяемость.

2. Требования к функциональности

- ❖ Реализация игрового поля размером 3x3.
- ❖ Возможность играть двум игрокам по очереди.
- ❖ Автоматическая проверка победных условий после каждого хода.
- ❖ Обработка ситуации ничьей.
- ❖ Возможность начать новую игру без перезапуска программы.
- ❖ Вывод текущего состояния игрового поля после каждого хода.
- ❖ Информирование игроков о ходе и результатах игры.

2.1. Основные классы и их функции

Класс: Game

Ответственный за управление процессом игры.

Методы:

- start(): запуск игры, цикл по очереди игроков.
- check_winner(): проверка победных условий.
- is_draw(): проверка ничьей.
- reset(): сброс состояния для новой игры.

Класс: Board

Представляет игровое поле.

Атрибуты:

- `grid`: двумерный список (3x3), хранящий текущие символы ('X', 'O', или пустые).

Методы:

- `display()`: вывод текущего состояния поля на экран.
- `update_cell(row, col, symbol)`: обновление клетки по координатам.
- `is_cell_empty(row, col)`: проверка свободной клетки.

Класс: `Player`

Представляет игрока.

Атрибуты:

- `name`: имя игрока.
- `symbol`: символ ('X' или 'O').

Методы:

- `make_move(board)`: запрос хода у пользователя (ввод координат).

2.2. Взаимодействие компонентов

- Игра запускается вызовом метода `start()` класса `Game`.
- В процессе игры:
- Создаются два объекта класса `Player`.
- Создается объект класса `Board`.
- Игроки по очереди делают ходы через метод `make_move()`.
- После каждого хода вызывается метод проверки победы или ничьей.
- Игра завершается при выявлении победителя или ничьей, выводится соответствующее сообщение.
- Предлагается начать новую игру или завершить программу.

3. Требования к интерфейсу

- ❖ Весь ввод/вывод осуществляется через консоль (терминал).
- ❖ После каждого хода отображается актуальное состояние поля.
- ❖ В случае некорректного ввода (например, неправильные координаты или занятая клетка) запрашивается повторный ввод.

4. Технические требования

- ❖ Использование Python версии не ниже 3.8.
- ❖ Соблюдение принципов ООП.
- ❖ Код должен быть структурированным, с комментариями.
- ❖ Реализовать обработку ошибок (например, при вводе некорректных данных).

5. Дополнительные требования

Обеспечить возможность расширения (например, добавление более сложных правил или графического интерфейса в будущем).

Вариант 3

Проект: Создание модели виртуального питомца (например, `Tamagotchi`)

1. Общие сведения

Цель проекта: разработать объектно-ориентированную модель виртуального питомца, который может выполнять базовые действия, требующие ухода, и взаимодействовать с

пользователем через консольное приложение.

2. Требования к функциональности

- ❖ Создание и инициализация виртуального питомца с базовыми характеристиками.
- ❖ Возможность кормить питомца.
- ❖ Возможность играть с питомцем.
- ❖ Возможность ухаживать за питомцем (например, чистить или укладывать спать).
- ❖ Автоматическое изменение состояния питомца со временем (например, голод, усталость).
- ❖ Отображение текущего состояния питомца.
- ❖ Взаимодействие с пользователем через меню или команды.
- ❖ Возможность сохранять и загружать состояние питомца (опционально).

2.1. Основные классы и их функции

Класс Pet

Основной класс, представляющий виртуального питомца.

Атрибуты:

- name: имя питомца.
- hunger: уровень голода (например, от 0 до 100).
- happiness: уровень счастья (от 0 до 100).
- energy: уровень энергии (от 0 до 100).
- age: возраст питомца.
- alive: статус жизни (True/False).

Методы:

- feed(): покормить питомца, уменьшает голод.
- play(): поиграть с питомцем, повышает счастье, расходует энергию.
- sleep(): отправить питомца спать, восстанавливает энергию.
- pass_time(): моделирует прохождение времени, увеличивая голод и усталость.
- show_status(): выводит текущее состояние питомца.

Класс VirtualPetGame

Управляет процессом взаимодействия пользователя с питомцем.

Методы:

- start(): запуск игрового цикла.
- show_menu(): отображение вариантов действий.
- process_choice(choice): обработка выбранного действия.
- check_alive(): проверка жизненного состояния питомца.

2.2. Взаимодействие компонентов

- При запуске создается объект класса Pet.
- В цикле пользователь выбирает действия из меню: кормление, игра, сон, просмотр статуса
- После каждого действия вызывается метод pass_time(), моделирующий течение времени.
- Если уровень голода или усталости достигает критических значений (например, выше или ниже допустимых границ), питомец умирает (alive =

False), игра завершается или предлагается начать заново.

3. Требования к интерфейсу

- ❖ Весь ввод/вывод осуществляется через консоль.
- ❖ После каждого действия отображается обновленное состояние питомца.
- ❖ Предлагается повторять действия до тех пор, пока питомец жив или пользователь не решит завершить игру.

4. Технические требования

- ❖ Использование Python версии не ниже 3.8.
- ❖ Соблюдение принципов ООП.
- ❖ Код должен быть структурированным, с комментариями.
- ❖ Реализовать обработку ошибок (например, при вводе некорректных данных).

5. Дополнительные требования

- ❖ Код должен быть структурированным и хорошо документированным.
- ❖ Можно реализовать сохранение/загрузку состояния в файл для продолжения игры позже (опционально).

Вариант 4

Проект: Создание системы управления задачами (To-Do List)

1. Общие сведения

Цель проекта: разработать объектно-ориентированную консольную программу для создания, редактирования и управления списком задач (To-Do List), позволяющую пользователю удобно добавлять, просматривать, редактировать и удалять задачи.

2. Требования к функциональности

- ❖ Создание новой задачи с указанием описания, приоритета и срока выполнения.
- ❖ Просмотр списка всех задач с отображением их статуса и основных данных.
- ❖ Возможность редактировать существующие задачи.
- ❖ Возможность удалять выполненные или ненужные задачи.
- ❖ Отметка задачи как выполненной.
- ❖ Фильтрация задач по статусу (все, выполненные, невыполненные).
- ❖ Сохранение списка задач между запусками программы (опционально — через файл).
- ❖ Интерактивное меню для взаимодействия с пользователем.

2.1. Основные классы и их функции

Класс Task

Представляет отдельную задачу.

Атрибуты:

- title: описание задачи.
- priority: уровень приоритета (например, низкий, средний, высокий).
- due_date: дата выполнения (может быть строкой или объектом даты).
- completed: статус выполнения (True/False).
- created_at: дата создания задачи.

Методы:

- mark_completed(): отметить задачу как выполненную.
- edit(): изменить параметры задачи.

- show(): вывести информацию о задаче.

Класс TaskManager

Управляет списком задач.

Атрибуты:

- tasks: список объектов Task.

Методы:

- add_task(): добавить новую задачу.
- remove_task(): удалить задачу по индексу или идентификатору.
- edit_task(): редактировать выбранную задачу.
- list_tasks(): выводить список задач с фильтрацией по статусу.
- save_to_file(): сохранять список задач в файл (например, JSON).
- load_from_file(): загружать список задач из файла.

Класс ToDoApp

Обеспечивает взаимодействие пользователя с системой через меню.

Методы:

- start(): запуск основного цикла программы.
- show_menu(): отображение вариантов действий.
- Обработка пользовательских команд.

2.2. Взаимодействие компонентов

- При запуске создается объект класса TaskManager.
- Пользователь через меню добавляет, редактирует или удаляет задачи.
- После каждого действия список отображается или сохраняется в файл для сохранения состояния между запусками.
- Можно реализовать фильтрацию задач по статусу для удобства просмотра.

3. Требования к интерфейсу

- ❖ Весь ввод/вывод осуществляется через консольное меню.
- ❖ После каждого действия выводится актуальный список задач или сообщение о результате операции.
- ❖ Ввод данных осуществляется по запросу с проверкой корректности.

4. Технические требования

- ❖ Использование Python версии не ниже 3.8.
- ❖ Соблюдение принципов ООП.
- ❖ Код должен быть структурированным, с комментариями.
- ❖ Реализовать обработку ошибок (например, при вводе некорректных данных).

5. Дополнительные требования

- ❖ Реализовать возможность сохранения и загрузки данных из файла (например, JSON) для продолжения работы после закрытия программы.

Вариант 5

Проект: Создание симулятора фермы (животные, растения)

1. Общие сведения

Цель проекта: разработать объектно-ориентированную программу-симулятор фермы, в которой пользователь сможет управлять животными и растениями, следить за их состоянием, ухаживать за ними и развивать ферму.

2. Требования к функциональности

- ❖ Создание и управление различными видами животных (например, коровы, овцы, куры) и растений (например, пшеница, овощи).
- ❖ Возможность добавлять новых животных и растения на ферму.
- ❖ Уход за животными: кормление, поение, уборка.
- ❖ Уход за растениями: полив, удобрение.
- ❖ Прохождение времени (дни), в течение которых состояние животных и растений меняется (например, голод, усталость, рост растений).
- ❖ Возможность получать продукты от животных (молоко, яйца) и урожай с растений.
- ❖ Отображение текущего состояния фермы: список животных и растений с их статусами.
- ❖ Возможность продавать или убирать животных/растения.
- ❖ Сохранение состояния фермы между запусками (опционально — через файл).

2.1. Основные классы и их функции

Класс Organism (базовый класс)

Общие свойства для всех живых существ на ферме.

Атрибуты:

- name: название вида или конкретного экземпляра.
- age: возраст.
- health: состояние здоровья (от 0 до 100).
- hunger: уровень голода (от 0 до 100).
- status: состояние (например, "здоров", "больной", "голодный").

Методы:

- pass_time(): моделирует течение времени — увеличивает голод, возрастает возраст.
- feed(): покормить — уменьшает голод.
- heal(): лечение при необходимости.

Класс Animal (наследует Organism)

Специфические свойства:

- product: продукт животного (молоко, яйца).

Методы:

- produce(): получение продукта при определенных условиях.
- care(): уход за животным.

Класс Plant (наследует Organism)

Специфические свойства:

- growth_stage: стадия роста растения.

Методы:

- grow(): рост растения с течением времени.
- harvest(): сбор урожая.

Класс Farm

- Управляет списками животных и растений.

Атрибуты:

- списки объектов Animal и Plant.

Методы:

- добавление/удаление животных и растений.
- уход за всеми объектами.
- моделирование прохождения времени (`pass_time()`), вызывающее соответствующие методы у всех объектов.
- отображение текущего состояния фермы.

Класс `FarmSimulator`

Обеспечивает взаимодействие пользователя через меню или командную строку.

Методы:

- запуск симуляции (`start()`).
- отображение меню (`show_menu()`).
- обработка команд пользователя.

2.2. Взаимодействие компонентов

- Пользователь создает ферму и добавляет на нее животных и растения.
- В цикле пользователь выбирает действия: ухаживать за существами, собирать продукты, проходить время или сохранять/загружать состояние.
- После каждого действия вызывается метод моделирования времени (`pass_time()`), который обновляет состояние всех объектов.

3. Требования к интерфейсу

- ❖ Весь ввод/вывод осуществляется через консольное меню.
- ❖ После каждого действия выводится актуальное состояние фермы: список животных и растений с их характеристиками.
- ❖ Предлагается возможность сохранять прогресс в файл и загружать его при запуске.

4. Технические требования

- ❖ Использование Python версии не ниже 3.8.
- ❖ Соблюдение принципов ООП.
- ❖ Код должен быть структурированным, с комментариями.
- ❖ Реализовать обработку ошибок (например, при вводе некорректных данных).

5. Дополнительные требования

- ❖ Реализовать возможность расширения системы новыми видами животных или растений без существенных изменений в коде.

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Методические указания
по организации и проведению самостоятельной работы
по дисциплине
«ПАРАДИГМЫ ПРОГРАММИРОВАНИЯ»
для студентов направления подготовки
02.04.01 Математика и компьютерные науки
профиль «Искусственный интеллект в кибербезопасности»

Общие положения

Самостоятельная работа - планируемая учебная, учебно-исследовательская, научно-исследовательская работа студентов, выполняемая во внеаудиторное (аудиторное) время по заданию и при методическом руководстве преподавателя, но без его непосредственного участия (при частичном непосредственном участии преподавателя, оставляющем ведущую роль за работой студентов).

Самостоятельная работа студентов (СРС) в ВУЗе является важным видом учебной и научной деятельности студента. Самостоятельная работа студентов играет значительную роль в рейтинговой технологии обучения.

К основным видам самостоятельной работы студентов относятся:

- **формирование и усвоение содержания конспекта лекций на базе рекомендованной лектором учебной литературы, включая информационные образовательные ресурсы (электронные учебники, электронные библиотеки и др.);**

- **написание докладов;**

- **подготовка к семинарам, практическим и лабораторным работам, их оформление;**

- **составление аннотированного списка статей из соответствующих журналов по отраслям знаний (педагогических, психологических, методических и др.);**

- **выполнение учебно-исследовательских работ, проектная деятельность;**

- **подготовка практических разработок и рекомендаций по решению проблемной ситуации;**

- **выполнение домашних заданий в виде решения отдельных задач, проведения типовых расчетов, расчетно-компьютерных и индивидуальных работ по отдельным разделам содержания дисциплин и т.д.;**

- **компьютерный текущий самоконтроль и контроль успеваемости на базе электронных обучающих и аттестующих тестов;**

- **выполнение курсовых работ (проектов) в рамках дисциплин;**

- **выполнение выпускной квалификационной работы и др.**

Методика организации самостоятельной работы студентов зависит от структуры, характера и особенностей изучаемой дисциплины, объема часов на ее изучение, вида заданий для самостоятельной работы студентов, индивидуальных качеств студентов и условий учебной деятельности.

Процесс организации самостоятельной работы студентов включает в себя следующие этапы:

- **подготовительный (определение целей, составление программы, подготовка методического обеспечения, подготовка оборудования);**

- **основной (реализация программы, использование приемов поиска информации, усвоения, переработки, применения, передачи знаний, фиксирование результатов, самоорганизация процесса работы);**

- заключительный (оценка значимости и анализ результатов, их систематизация, оценка эффективности программы и приемов работы, выводы о направлениях оптимизации труда).

Самостоятельная работа по дисциплине «Парадигмы программирования» направлена на формирование следующих **компетенций**:

ПК-1. Способен демонстрировать базовые знания математических и естественных наук, основ программирования и информационных технологий.

1. Цель и задачи самостоятельной работы

Ведущая цель организации и осуществления СРС совпадает с целью обучения студента – формирование соответствующего набора компетенций будущего бакалавра по направлению подготовки «Математика и компьютерные науки».

При организации СРС важным и необходимым условием становятся формирование умения самостоятельной работы для приобретения знаний, навыков и возможности организации учебной и научной деятельности. Целью самостоятельной работы студентов является овладение фундаментальными знаниями, профессиональными умениями и навыками деятельности по профилю, опытом творческой, исследовательской деятельности. Самостоятельная работа студентов способствует развитию самостоятельности, ответственности и организованности, творческого подхода к решению проблем учебного и профессионального уровня.

Задачами СРС являются:

- систематизация и закрепление полученных теоретических знаний и практических умений студентов;
- углубление и расширение теоретических знаний;
- формирование умений использовать нормативную, правовую, справочную документацию и специальную литературу;
- развитие познавательных способностей и активности студентов: творческой инициативы, самостоятельности, ответственности и организованности;
- формирование самостоятельности мышления, способностей к саморазвитию, самосовершенствованию и самореализации;
- развитие исследовательских умений;
- использование материала, собранного и полученного в ходе самостоятельных занятий на семинарах, на практических и лабораторных занятиях, при написании курсовых и выпускной квалификационной работ, для эффективной подготовки к итоговым зачетам и экзаменам.

2. Технологическая карта самостоятельной работы студента

Коды реализуемых компетенций	Вид деятельности студентов	Средства и технологии оценки	Объем часов, в том числе (акад.)		
			СРС	Контактная работа с преподавателем	Всего
1 семестр					
ПК-1	Самостоятельное изучение литературы	Собеседование	16	2	18

ПК-1	Подготовка к практическим занятиям	Защита практической работы	42	4	46
ПК-1	Подготовка доклада и презентации	Защита доклада и презентации	1	1	2
ПК-1	Подготовка проекта	Защита проекта	4	2	6
Итого за 1 семестр			63	9	72
Итого			63	9	72

3. Порядок выполнения самостоятельной работы студентом

3.1. Методические рекомендации по работе с учебной литературой

При работе с книгой необходимо подобрать литературу, научиться правильно ее читать, вести записи. Для подбора литературы в библиотеке используются алфавитный и систематический каталоги.

Важно помнить, что рациональные навыки работы с книгой – это всегда большая экономия времени и сил.

Правильный подбор учебников рекомендуется преподавателем, читающим лекционный курс. Необходимая литература может быть также указана в методических разработках по данному курсу.

Изучая материал по учебнику, следует переходить к следующему вопросу только после правильного уяснения предыдущего, описывая на бумаге все выкладки и вычисления (в том числе те, которые в учебнике опущены или на лекции даны для самостоятельного вывода).

При изучении любой дисциплины большую и важную роль играет самостоятельная индивидуальная работа.

Особое внимание следует обратить на определение основных понятий курса. Студент должен подробно разбирать примеры, которые поясняют такие определения, и уметь строить аналогичные примеры самостоятельно. Нужно добиваться точного представления о том, что изучаешь. Полезно составлять опорные конспекты. При изучении материала по учебнику полезно в тетради (на специально отведенных полях) дополнять конспект лекций. Там же следует отмечать вопросы, выделенные студентом для консультации с преподавателем.

Выводы, полученные в результате изучения, рекомендуется в конспекте выделять, чтобы они при перечитывании записей лучше запоминались.

Опыт показывает, что многим студентам помогает составление листа опорных сигналов, содержащего важнейшие и наиболее часто употребляемые формулы и понятия. Такой лист помогает запомнить формулы, основные положения лекции, а также может служить постоянным справочником для студента.

Чтение научного текста является частью познавательной деятельности. Ее цель – извлечение из текста необходимой информации. От того на сколько осознанна читающим собственная внутренняя установка при обращении к печатному слову (найти нужные сведения, усвоить информацию полностью или частично, критически проанализировать материал и т.п.) во многом зависит эффективность осуществляемого действия.

Выделяют **четыре основные установки в чтении научного текста:**

информационно-поисковый (задача – найти, выделить искомую информацию)

усваивающая (усилия читателя направлены на то, чтобы как можно полнее осознать и запомнить как сами сведения, излагаемые автором, так и всю логику его рассуждений)

аналитико-критическая (читатель стремится критически осмыслить материал, проанализировав его, определив свое отношение к нему)

творческая (создает у читателя готовность в том или ином виде – как отправной пункт для своих рассуждений, как образ для действия по аналогии и т.п. – использовать суждения автора, ход его мыслей, результат наблюдения, разработанную методику, дополнить их, подвергнуть новой проверке).

Основные виды систематизированной записи прочитанного:

Аннотирование – предельно краткое связное описание просмотренной или прочитанной книги (статьи), ее содержания, источников, характера и назначения;

Планирование – краткая логическая организация текста, раскрывающая содержание и структуру изучаемого материала;

Тезирование – лаконичное воспроизведение основных утверждений автора без привлечения фактического материала;

Цитирование – дословное выписывание из текста выдержек, извлечений, наиболее существенно отражающих ту или иную мысль автора;

Конспектирование – краткое и последовательное изложение содержания прочитанного.

Конспект – сложный способ изложения содержания книги или статьи в логической последовательности. Конспект аккумулирует в себе предыдущие виды записи, позволяет всесторонне охватить содержание книги, статьи. Поэтому умение составлять план, тезисы, делать выписки и другие записи определяет и технологию составления конспекта.

Методические рекомендации по составлению конспекта:

1. Внимательно прочитайте текст. Уточните в справочной литературе непонятные слова. При записи не забудьте вынести справочные данные на поля конспекта;

2. Выделите главное, составьте план;

3. Кратко сформулируйте основные положения текста, отметьте аргументацию автора;

4. Законспектируйте материал, четко следуя пунктам плана. При конспектировании старайтесь выразить мысль своими словами. Записи следует вести четко, ясно.

5. Грамотно записывайте цитаты. Цитируя, учитывайте лаконичность, значимость мысли.

В тексте конспекта желательно приводить не только тезисные положения, но и их доказательства. При оформлении конспекта необходимо стремиться к емкости каждого предложения. Мысли автора книги следует излагать кратко, заботясь о стиле и выразительности написанного. Число дополнительных элементов конспекта должно быть логически обоснованным, записи должны распределяться в определенной последовательности, отвечающей логической структуре произведения. Для уточнения и дополнения необходимо оставлять поля.

Овладение навыками конспектирования требует от студента целеустремленности, повседневной самостоятельной работы.

3.2. Методические рекомендации по подготовке к практическим и лабораторным занятиям

Для того чтобы практические и лабораторные занятия приносили максимальную пользу, необходимо помнить, что упражнение и решение задач проводятся по вычитанному на лекциях материалу и связаны, как правило, с детальным разбором отдельных вопросов лекционного курса. Следует подчеркнуть, что только после усвоения лекционного

материала с определенной точки зрения (а именно с той, с которой он излагается на лекциях) он будет закрепляться на практических занятиях как в результате обсуждения и анализа лекционного материала, так и с помощью решения проблемных ситуаций, задач. При этих условиях студент не только хорошо усвоит материал, но и научится применять его на практике, а также получит дополнительный стимул (и это очень важно) для активной проработки лекции.

При самостоятельном решении задач нужно обосновывать каждый этап решения, исходя из теоретических положений курса. Если студент видит несколько путей решения проблемы (задачи), то нужно сравнить их и выбрать самый рациональный. Полезно до начала вычислений составить краткий план решения проблемы (задачи). Решение проблемных задач или примеров следует излагать подробно, вычисления располагать в строгом порядке, отделяя вспомогательные вычисления от основных. Решения при необходимости нужно сопровождать комментариями, схемами, чертежами и рисунками.

Следует помнить, что решение каждой учебной задачи должно доводиться до окончательного логического ответа, которого требует условие, и по возможности с выводом. Полученный ответ следует проверить способами, вытекающими из существа данной задачи. Полезно также (если возможно) решать несколькими способами и сравнить полученные результаты. Решение задач данного типа нужно продолжать до приобретения твердых навыков в их решении.

3.3. Методические рекомендации по самопроверке знаний

После изучения определенной темы по записям в конспекте и учебнику, а также решения достаточного количества соответствующих задач на практических занятиях и самостоятельно студенту рекомендуется, провести самопроверку усвоенных знаний, ответив на контрольные вопросы по изученной теме.

В случае необходимости нужно еще раз внимательно разобраться в материале.

Иногда недостаточность усвоения того или иного вопроса выясняется только при изучении дальнейшего материала. В этом случае надо вернуться назад и повторить плохо усвоенный материал. Важный критерий усвоения теоретического материала - умение решать задачи или пройти тестирование по пройденному материалу. Однако следует помнить, что правильное решение задачи может получиться в результате применения механически заученных формул без понимания сущности теоретических положений.

3.4. Методические рекомендации по написанию научных текстов (докладов, докладов, эссе, научных статей и т.д.)

Перед тем, как приступить к написанию научного текста, важно разобраться, какова истинная цель вашего научного текста – это поможет вам разумно распределить свои силы и время.

Во-первых, сначала нужно определиться с идеей научного текста, а для этого необходимо научиться либо относиться к разным явлениям и фактам несколько критически (своя идея – как иная точка зрения), либо научиться увлекаться какими-то известными идеями, которые нуждаются в доработке (идея – как оптимистическая позиция и направленность на дальнейшее совершенствование уже известного). Во-вторых, научиться организовывать свое время, ведь, как известно, свободное (от всяких глупостей) время – важнейшее условие настоящего творчества, для него наконец-то появляется время. Иногда именно на организацию такого времени уходит немалая часть сил и талантов.

Писать следует ясно и понятно, стараясь основные положения формулировать четко и недвусмысленно (чтобы и самому понятно было), а также стремясь структурировать свой текст. Каждый раз надо представлять, что ваш текст будет кто-то читать и ему захочется сориентироваться в нем, быстро находить ответы на интересующие вопросы (заодно

представьте себя на месте такого человека). Понятно, что работа, написанная «сплошным текстом» (без заголовков, без выделения крупным шрифтом наиболее важных мест и т. п.), у культурного читателя должна вызывать брезгливость и даже жалость к автору (исключения составляют некоторые древние тексты, когда и жанр был иной и к текстам относились иначе, да и самих текстов было гораздо меньше – не то, что в эпоху «информационного взрыва» и соответствующего «информационного мусора»).

Объем текста и различные оформительские требования во многом зависят от принятых в конкретном учебном заведении порядков.

Доклад – это самостоятельное исследование студентом определенной проблемы, комплекса взаимосвязанных вопросов.

Доклад не должна составляться из фрагментов статей, монографий, пособий. Кроме простого изложения фактов и цитат, в доклад е должно проявляться авторское видение проблемы и ее решения.

Рассмотрим основные этапы подготовки доклада студентом.

Выполнение доклада начинается с выбора темы.

Затем студент приходит на первую консультацию к руководителю, которая предусматривает:

- обсуждение цели и задач работы, основных моментов избранной темы;
- консультирование по вопросам подбора литературы;
- составление предварительного плана.

Следующим этапом является работа с литературой. Необходимая литература подбирается студентом самостоятельно.

После подбора литературы целесообразно сделать рабочий вариант плана работы. В нем нужно выделить основные вопросы темы и параграфы, раскрывающие их содержание.

Составленный список литературы и предварительный вариант плана уточняются, согласуются на очередной консультации с руководителем.

Затем начинается следующий этап работы - изучение литературы. Только внимательно читая и конспектируя литературу, можно разобраться в основных вопросах темы и подготовиться к самостоятельному (авторскому) изложению содержания доклада. Конспектируя первоисточники, необходимо отразить основную идею автора и его позицию по исследуемому вопросу, выявить проблемы и наметить задачи для дальнейшего изучения данных проблем.

Систематизация и анализ изученной литературы по проблеме исследования позволяют студенту написать работу.

Рабочий вариант текста доклада предоставляется руководителю на проверку. На основе рабочего варианта текста руководитель вместе со студентом обсуждает возможности доработки текста, его оформление. После доработки доклад сдается на кафедру для его оценивания руководителем.

Требования к написанию доклада

Написание 1 доклада является обязательным условием выполнения плана СРС по любой дисциплине профессионального цикла.

Тема доклада может быть выбрана студентом из предложенных в рабочей программе или фонде оценочных средств дисциплины, либо определена самостоятельно, исходя из интересов студента (в рамках изучаемой дисциплины). Выбранную тему необходимо согласовать с преподавателем.

Доклад должен быть написан научным языком.

Объем доклада должен составлять 20-25 стр.

Структура доклада:

- Введение (не более 3-4 страниц). Во введении необходимо обосновать выбор темы, ее актуальность, очертить область исследования, объект исследования, основные цели и задачи исследования.

- Основная часть состоит из 2-3 разделов. В них раскрывается суть исследуемой проблемы, проводится обзор мировой литературы и источников Интернет по предмету исследования, в котором дается характеристика степени разработанности проблемы и авторская аналитическая оценка основных теоретических подходов к ее решению. Изложение материала не должно ограничиваться лишь описательным подходом к раскрытию выбранной темы. Оно также должно содержать собственное видение рассматриваемой проблемы и изложение собственной точки зрения на возможные пути ее решения.

- Заключение (1-2 страницы). В заключении кратко излагаются достигнутые при изучении проблемы цели, перспективы развития исследуемого вопроса

- Список использованной литературы (не меньше 10 источников), в алфавитном порядке, оформленный в соответствии с принятыми правилами. В список использованной литературы рекомендуется включать работы отечественных и зарубежных авторов, в том числе статьи, опубликованные в научных журналах в течение последних 3-х лет и ссылки на ресурсы сети Интернет.

- Приложение (при необходимости).

Требования к оформлению:

- текст с одной стороны листа;
- шрифт Times New Roman;
- кегль шрифта 14;
- межстрочное расстояние 1,5;
- поля: сверху 2,5 см, снизу – 2,5 см, слева - 3 см, справа 1,5 см;
- доклад должен быть представлен в сброшюрованном виде.

Порядок защиты доклада:

Защита доклада проводится на практических занятиях, после окончания работы студента над ним и исправления всех недочетов, выявленных преподавателем в ходе консультаций. На защиту доклада отводится 5-7 минут времени, в ходе которого студент должен показать свободное владение материалом по заявленной теме. При защите доклада приветствуется использование мультимедиа-презентации.

Оценка доклада

Доклад оценивается по следующим критериям:

- соблюдение требований к его оформлению;
- необходимость и достаточность для раскрытия темы приведенной в тексте доклада информации;
- умение студента свободно излагать основные идеи, отраженные в докладе;
- способность студента понять суть задаваемых преподавателем и сокурсниками вопросов и сформулировать точные ответы на них.

Критерии оценки:

Оценка «отлично» выставляется студенту, если в докладе студент исчерпывающе, последовательно, четко и логически стройно излагает материал; свободно справляется с задачами, вопросами и другими видами применения знаний; использует для написания доклада современные научные материалы; анализирует полученную информацию; проявляет самостоятельность при написании доклада.

Оценка «хорошо» выставляется студенту, если качество выполнения доклада достаточно высокое. Студент твердо знает материал, грамотно и по существу излагает его, не допуская существенных неточностей в ответе на вопросы по теме доклада.

Оценка «удовлетворительно» выставляется студенту, если материал доклада излагается частично, но пробелы не носят существенного характера, студент допускает неточности и ошибки при защите доклада, дает недостаточно правильные формулировки, наблюдаются нарушения логической последовательности в изложении материала.

Оценка «неудовлетворительно» выставляется студенту, если он не подготовил доклад или допустил существенные ошибки. Студент неуверенно излагает материал доклада, не отвечает на вопросы преподавателя.

Описание шкалы оценивания

Максимально возможный балл за весь текущий контроль устанавливается равным 55. Текущее контрольное мероприятие считается сданным, если студент получил за него не менее 60% от установленного для этого контроля максимального балла. Рейтинговый балл, выставляемый студенту за текущее контрольное мероприятие, сданное студентом в установленные графиком контрольных мероприятий сроки, определяется следующим образом:

Уровень выполнения контрольного задания	Рейтинговый балл (в % от максимального балла за контрольное задание)
Отличный	100
Хороший	80
Удовлетворительный	60
Неудовлетворительный	0

3.5. Методические рекомендации по выполнению исследовательских проектов

Исследовательская проектная работа – это групповая работа, для выполнения которой необходим выбор и приложение научной методики к поставленной задаче, получение собственного теоретического или экспериментального материала, на основании которого необходимо провести анализ и сделать выводы об исследуемом явлении. Выполнение проекта – это всегда коллективная, творческая практическая работа, предназначенная для получения определенного продукта или научно-технического результата. Такая работа подразумевает четкое, однозначное формирование поставленной задачи, определение сроков выполнения намеченного, определение требований к разрабатываемому объекту.

Выполнение 1 группового проекта является обязательным условием выполнения самостоятельной работы по любой дисциплине профессионального цикла. Тема проектного задания может быть выбрана студентом из предложенных в рабочей программе или фонде оценочных средств дисциплины, либо определена самостоятельно, исходя из интересов студента (в рамках изучаемой дисциплины). Выбранную тему необходимо согласовать с преподавателем.

Требования по выполнению и оформлению проекта

При выполнении проекта приветствуется работа в группе (2-3 человека). Проект – это исследовательская работа, в ходе которой студенты должны продемонстрировать владение навыками научного исследования, умения проводить анализ, обобщать информацию, делать выводы, предлагать свои решения проблемы, рассматриваемой в проекте.

При подготовке материалов проекта студенты должны продемонстрировать владение современными методами компьютерной обработки данных.

Критерии оценки работы участника проекта.

Для каждого из участников проекта оцениваются:

- профессиональные теоретические знания в соответствующей области;
- умение работать со справочной и научной литературой, осуществлять поиск необходимой информации в Интернет;
- умение работать с техническими средствами;
- умение пользоваться соответствующими выполняемому проекту информационными технологиями;

- умение готовить материалы проекта для презентации: составлять и редактировать тексты, формировать презентацию проекта;
- умение работать в команде;
- умение публично представлять результаты собственной деятельности;
- коммуникабельность, инициативность, творческие способности.

Критерии выставления оценки участникам проекта

Оценка	Профессиональные компетенции	Компетенции, связанные с использованием соответствующих выполняемому проекту технических средств и информационных технологий	Иные универсальные компетенции (коммуникабельность, инициативность, умение работать в «команде», управленческие навыки и т.д.)	Отчетность
«Отлично»	Работа выполнена на высоком профессиональном уровне. Представленный материал в основном фактически верен, допускаются негрубые фактические неточности. Студент свободно отвечает на вопросы, связанные с проектом.	Технические средства и информационные технологии освоены и использованы для реализации проекта полностью	Студент проявил инициативу, творческий подход, способность к выполнению сложных заданий, навыки работы в коллективе, организационные способности.	Проект представлен полностью и в срок.
«Хорошо»	Работа выполнена на достаточно высоком профессиональном уровне. Допущено до 4–5 фактических ошибок. Студент отвечает на вопросы, связанные с проектом, но недостаточно полно.	Обнаруживаются некоторые ошибки в использовании соответствующих технических средств и информационных технологий	Студент достаточно полно, но без инициативы и творческих находок выполнил возложенные на него задачи.	Проект представлен достаточно полно и в срок, но с некоторыми недоработками.
«Удовлетворительно»	Уровень недостаточно высок. Допущено до 8 фактических ошибок. Студент может ответить лишь на некоторые из заданных вопросов, связанных с проектом.	Обнаруживает недостаточное владение навыками работы с техническими средствами и соответствующим и информационным и технологиями	Студент выполнил большую часть возложенной на него работы.	Проект сдан со значительным опозданием (более недели) и не полностью

«Неудовлетворительно»	Работа не выполнена или выполнена на низком уровне. Допущено более 8 фактических ошибок. Ответы на связанные с проектом вопросы обнаруживают непонимание предмета и отсутствие ориентации в материале проекта.	Навыков работы с техническими средствами нет, информационные технологии не освоены	Студент практически не работал, не выполнил свои задачи или выполнил лишь отдельные не существенные поручения в групповом проекте.	Проект не сдан.
-----------------------	--	--	--	-----------------

Студенты должны: защитить проект в режиме презентации, предъявить файлы выполненного проекта, уметь рассказать о технологиях, использованных ими при выполнении проекта, дать оценку работы каждого члена группы (*если проект групповой*).

Максимально возможный балл за весь текущий контроль устанавливается равным 55. Текущее контрольное мероприятие считается сданным, если студент получил за него не менее 60% от установленного для этого контроля максимального балла. Рейтинговый балл, выставляемый студенту за текущее контрольное мероприятие, сданное студентом в установленные графиком контрольных мероприятий сроки, определяется следующим образом:

Уровень выполнения контрольного задания	Рейтинговый балл (в % от максимального балла за контрольное задание)
Отличный	100
Хороший	80
Удовлетворительный	60
Неудовлетворительный	0

3.6. Методические рекомендации по подготовке к экзаменам и зачетам

Изучение многих общепрофессиональных и специальных дисциплин завершается экзаменом. Подготовка к экзамену способствует закреплению, углублению и обобщению знаний, получаемых, в процессе обучения, а также применению их к решению практических задач. Готовясь к экзамену, студент ликвидирует имеющиеся пробелы в знаниях, углубляет, систематизирует и упорядочивает свои знания. На экзамене студент демонстрирует то, что он приобрел в процессе обучения по конкретной учебной дисциплине.

Экзаменационная сессия – это серия экзаменов, установленных учебным планом. Между экзаменами интервал 3-4 дня. Не следует думать, что 3-4 дня достаточно для успешной подготовки к экзаменам.

В эти 3-4 дня нужно систематизировать уже имеющиеся знания. На консультации перед экзаменом студентов познакомят с основными требованиями, ответят на возникшие у них вопросы. Поэтому посещение консультаций обязательно.

Требования к организации подготовки к экзаменам те же, что и при занятиях в течение семестра, но соблюдаться они должны более строго. Во-первых, очень важно соблюдение режима дня; сон не менее 8 часов в сутки, занятия заканчиваются не позднее, чем за 2-3 часа до сна. Оптимальное время занятий - утренние и дневные часы. В перерывах

между занятиями рекомендуются прогулки на свежем воздухе, неустойчивые занятия спортом. Во-вторых, наличие хороших собственных конспектов лекций. Даже в том случае, если была пропущена какая-либо лекция, необходимо во время ее восстановления (переписать ее на кафедре), обдумать, снять возникшие вопросы для того, чтобы запоминание материала было осознанным. В-третьих, при подготовке к экзаменам у студента должен быть хороший учебник или конспект литературы, прочитанной по указанию преподавателя в течение семестра. Здесь можно эффективно использовать листы опорных сигналов.

Вначале следует просмотреть весь материал по сдаваемой дисциплине, отметить для себя трудные вопросы. Обязательно в них разобраться. В заключение еще раз целесообразно повторить основные положения, используя при этом листы опорных сигналов.

Систематическая подготовка к занятиям в течение семестра позволит использовать время экзаменационной сессии для систематизации знаний.

Контроль самостоятельной работы студентов

Контроль самостоятельной работы проводится преподавателем в аудитории.

Предусмотрены следующие виды контроля: собеседование, оценка выполнения практической работы, оценка доклада, оценка презентации, оценка выполнения проекта.

Подробные критерии оценивания компетенций приведены в Фонде оценочных средств для проведения текущей и промежуточной аттестации.

Список литературы для выполнения СРС

Перечень основной литературы:

1. Городняя, Л. В. Парадигма программирования: курс лекций / Л. В. Городняя ; Новосиб. гос. ун-т. – Новосибирск : РИЦ НГУ, 2015. – 206 с.
2. Пратт Т. Языки программирования. Разработка и реализация. / Т. Пратт, М. Зелковиц; Под общей редакцией А.Матросова. – СПб.: Питер, 2002. 688 с.
3. Кауфман В. Ш. Языки программирования: концепции и принципы. М.: ДМК Пресс, 2010. 464 с.
4. Себеста Р.У. Основные концепции языков программирования: 5-е издание / Себеста Р.У. — М.: Вильямс, 2001. — 672 с.

Перечень дополнительной литературы:

1. Сошников Д. В. Функциональное программирование на языке F#. /Д. В. Сошников. – М.: ДМК Пресс, 2011.
2. Хендерсон П. Функциональное программирование. / П. Хендерсон. – М.: Мир, 1983. 349 с.
3. Бадд Т. Объектно-ориентированное программирование в действии. СПб.: Питер, 1997. 464 с.
4. Роганов Е.А. Основы информатики и программирования: Учебное пособие / Е.А. Роганов. — М.: МГИУ, 2001. — 315 с.
5. Сергиевский Г.М. Функциональное и логическое программирование: учеб. пособие для студ. высш. учеб. заведений / Г. М. Сергиевский, Н.Г. Волчёнков. – М.: Издательский центр «Академия», 2010. – 320 с.
6. Игонина Е.В. Основы логического программирования и реализация программ на языке Пролог: учебное пособие. – Елец: ЕГУ им. И.А. Бунина, 2018. – 92 с.
7. Задорожный С.С., Фадеев Е.П. Объектно-ориентированное программирование на

языке Python / М. : Физический факультет МГУ им. М. В. Ломоносова, 2022. – 40 с.

**Перечень ресурсов информационно-телекоммуникационной сети «Интернет»,
необходимых для освоения дисциплины**

1. Что такое парадигмы программирования и зачем они нужны
<https://practicum.yandex.ru/blog/paradigmy-programmirovaniya/>
2. Что такое парадигмы программирования <https://sky.pro/media/chto-takoe-paradigmy-programmirovaniya/>
3. Введение в язык программирования R <https://textbook.rintro.ru/index.html>
4. Python Lessons. Русскоязычная документация по Python.
<https://pylessons.readthedocs.io/ru/latest/>