Министерство образования и науки РФ
ФГБОУ ВО «Томский государственный университет
систем управления и радиоэлектроники»
Кафедра безопасности информационных систем (БИС)

А.С. Романов

ЯЗЫКИ ПРОГРАММИРОВАНИЯ

Методические указания по лабораторным работам, практическим занятиям, самостоятельной и индивидуальной работе

Томск – 2018

Романов А.С. Языки программирования: Методические указания по лабораторным работам, практическим занятиям, самостоятельной и индивидуальной работе. – Томск: В-Спектр, 2018. – 82 с.

Учебное пособие содержит описания лабораторных работ и задания к практическим занятиям с примерами выполнения, требования по представлению отчётности, вопросы для самоконтроля по дисциплине «Языки программирования».

Предназначено для студентов специальностей: 10.03.01 – «Информационная безопасность», 10.05.02 – «Информационная безопасность телекоммуникационных систем», 10.05.04 – «Информационно-аналитические системы безопасности», 10.05.03 – «Информационная безопасность автоматизированных систем», 38.05.01 – «Экономическая безопасность».

СОДЕРЖАНИЕ

Тема № 1

Автоматизированный анализ текста на естественном языке

Цель работы

Закрепить знания, полученные студентами при изучении учебных курсов по программированию. Познакомиться с алгоритмами обработки текста.

Краткие теоретические сведения

Основными элементами языка программирования являются:

Ввод – считывание значений, поступающих с клавиатуры, портов ввода-вывода, жесткого диска и т.д.

Синтаксис – набор правил, которые определяют, какие символы являются допустимыми.

Семантика – это смысл синтаксических категорий языка программирования.

Данные – сущности, над которыми выполняются вычисления в программах и которые получаются в результате этих вычислений. К данным можно отнести константы, литералы, переменные, массивы и структуры, содержащие числа, текст, адреса переменных и т.д.

Операторы – команды языка программирования.

Вывод – вывод информации на экран, запись на жесткий диск или в порт ввода-вывода.

Условия – выполнение набора операторов только в случае, если выполняется некоторое заданное условие.

Циклы – конструкция языка программирования, реализующая многократное повторение группы операторов.

Подпрограммы – участки исходного кода, которые можно неоднократно выполнять в разных местах программы, вызывая их по имени.

Для того чтобы закрепить знания, полученных ранее при изучении курсов по программированию, рассмотрим простую, на первый взгляд тему – ***автоматизированный анализ текста***.

Специфика алгоритмов разбора текста состоит в том, что в процессе их работы происходит посимвольный или пословный анализ всего текста, в результате которого проверяется, удовлетворяет ли данная последовательность символов или слов определенной группе эвристических правил. Таким образом, для вынесения итогового решения, проверяется несколько промежуточных условий. Текущее состояние решения и очередной символ, поданный на вход алгоритма, определяют следующее состояние. Очевидно, что лучшим техническим решением при реализации данной группы алго-

ритмов является использование ***конечных автоматов***, преимуществом которых является возможность расширения функциональности алгоритма за счет добавления новых состояний.

Процесс обработки текста включает следующие этапы:

1. Лексический анализ.
2. Синтаксический анализ.
3. Семантически анализ.

Собственно эти же этапы проходит ***исходный текст программы*** в процессе ***трансляции*** в ***исполняемый код***: лексический анализатор преобразует последовательности отдельных символов в синтаксические инструкции (лексемы); синтаксический – строит из лексем иерархическую структуру зависимостей и, таким образом, выделяет или проверяет правильность совокупности синтаксических инструкций; семантический – придает смысл этим инструкциям (определяет адреса переменных, подпрограмм, последовательностей параметров в стеке при вызове подпрограммы и т.д.). Поэтому понимание того, как работают подобные алгоритмы для ***естественного языка***, поможет лучше понять, как работают подобные алгоритмы для ***искусственных языков*** программирования.

Однако анализ текста на естественном языке, как показывает практика, намного сложнее, чем анализ исходного текста программ. Рассмотрим один из самых простых алгоритмов автоматизированного анализа текста – ***алгоритм определения границ предложения***.

Обычно, за базовый алгоритм определения конца предложения принимают поиск символов, которыми обозначается его предложения: точка, вопросительный или восклицательный знаки. Началом предложения считается первый символ текста, ASCII код которого больше 32 (код символа «пробел»). Состояния конечного автомата, реализующего этот алгоритм показаны на рис. 1.1.

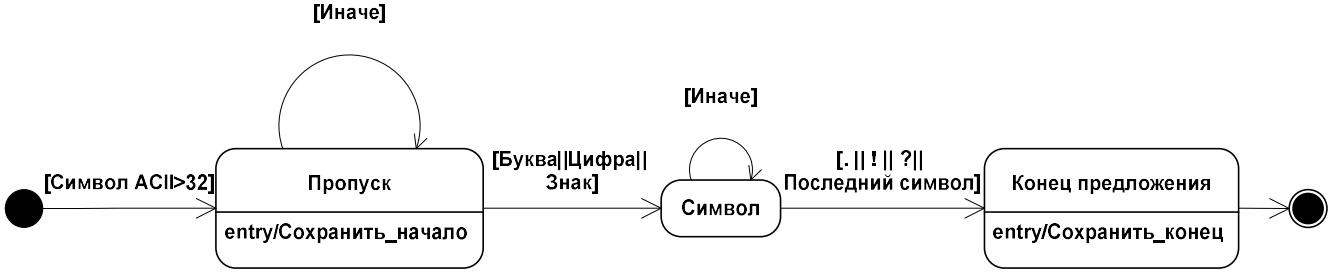

Рис. 1.1. Простой вариант алгоритма определения конца предложения

Но поиск конца предложения не всегда так прост и однозначен. С высокой долей вероятности восклицательный или вопросительный знак означает конец предложения, в отличие от точки, которая может использоваться в качестве разделителя в середине десятичной дроби, после сокращений и т.д. Для точного ответа на вопрос, находится ли данная точка в конце пред-

ложения, потребуется выполнить синтаксический и, возможно, семантический анализ всего фрагмента текста, что на этапе лексического анализа сделать невозможно.

Добавим проверку: точка, вопросительный или восклицательный знак означают конец предложения, если сразу за ним не следует любой из трёх перечисленных символов, а следующая за ними после разделителей буква – прописная (отсекаем ложные срабатывания в «…», «?!» и т.п.). Если стоящий после такого знака символ – закрывающая скобка или кавычка, то он входит в состав предложения (см. рис. 1.2).

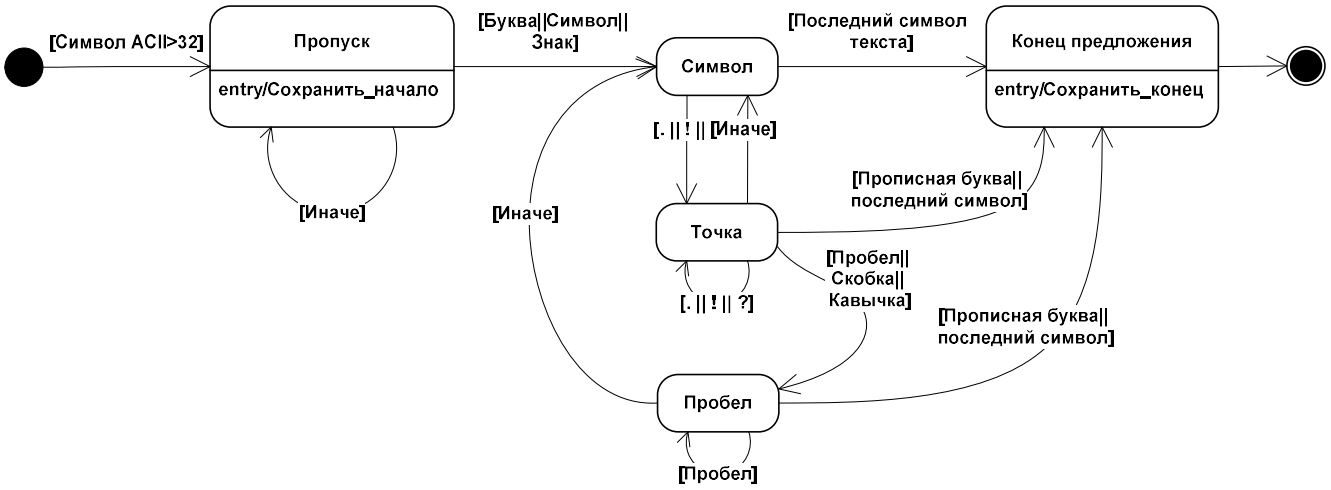

Рис. 1.2. Улучшенный вариант алгоритма определения конца предложения

В алгоритме необходимо учитывать следующие сокращения русского языка:

– употребляющиеся перед именами собственными: г. (город, господин), ул. (улица), тов. (товарищ), и другие;

– которые могут стоять в конце предложения: т.д., т.п., проч., пр., др.;

– которые не могут стоять в конце предложения: т.к., т.о., т.е.;

– инициалы.

Таким образом, схема, согласно которой предложение заканчивается точкой, а начинается обязательно с прописной буквы, должна дополняться проверками различных типов сокращений.

Предложение, следующее за сокращениями второго типа, как правило, начинаются с прописной буквы, слова после сокращений третьего типа начинаются со строчной буквы. Таким образом, в алгоритме необходимо учесть только сокращения первого типа и случай с инициалами.

В первом случае используются словари имен собственных и словарь возможных сокращений, употребляемых с ними. На определенном шаге, когда алгоритм встречает в тексте точку, происходит сопоставление псевдослова слева со словарем сокращений и псевдослова справа со словарем имен собственных. В случае, когда оба псевдослова найдены, точка не считается концом предложения.

При этом под «псевдословом» здесь и далее понимается любая последовательность букв и цифр между двумя символами-разделителями слов.

Чтобы исключить инициалы, вводится следующее правило: одиночная точка, идущая после псевдослова, состоящего из одной прописной буквы, не считается концом предложения. Единственной возможной проблемой при введении такой проверки являются междометия, состоящие из одной гласной буквы. Но, как правило, в этом случае они употребляются с восклицательным или вопросительным знаком или многоточием, т.к. несут эмоциональную нагрузку.

Распространенным является случай написания десятичных дробей с точкой в качестве разделителя, а также цифробуквенных комплексов для обозначения серийного номера и т.п. Для этого в алгоритме вводится дополнительное условие, осуществляющее проверку символов слева и справа от точки. Если оба символа цифры, то точка не считается концом предложения.

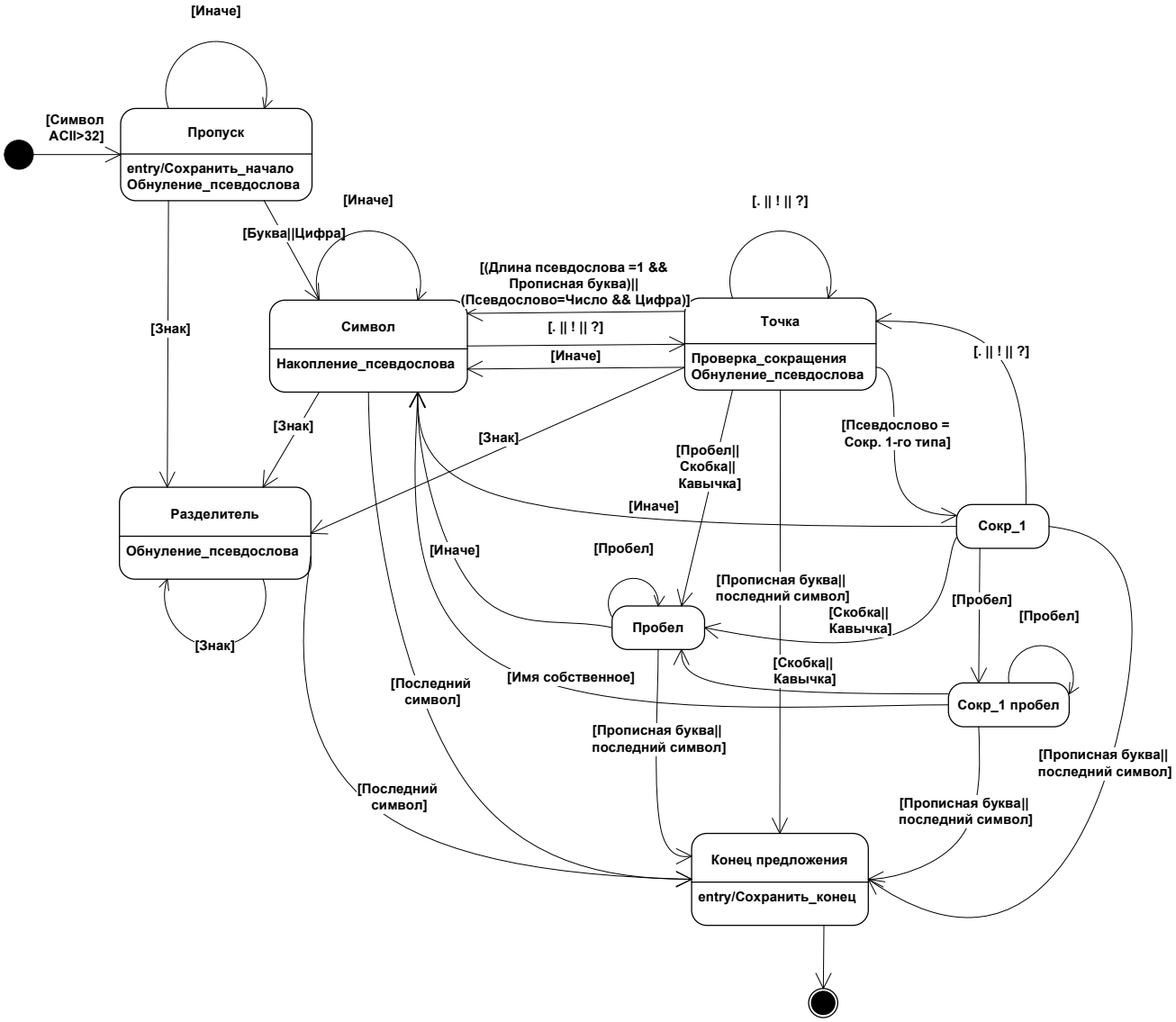

Рис. 1.3. Алгоритм определения границ предложений, учитывающий сокращения

С учетом замечаний был предложен алгоритм, изображенный на рис. 1.3.

Диаграмма состояний графа для определения границ предложений в коротких электронных сообщениях показана на рис. 1.4. Особенностью данного вида текстов является использование авторами эмотиконов.

Началом предложения считается первый печатный символ текста. Концом предложения - последний символ сообщения, точка, вопросительный или восклицательный знак или их группа, а также любой эмотикон. Эмотиконы в большинстве случаев выражают законченность мысли и служат для придания написанным словам дополнительной эмоциональной окраски, тогда как в середине предложения употребляются редко. Также они используются в начале сообщения, чтобы выразить эмоции по отношению, например, к предыдущей фразе собеседника – в этом случае алгоритм не выделяет эмотикон как отдельное предложение, а включает его в состав первого предложения сообщения.

Проверка группы символов на эмотикон происходит в несколько этапов.

1. Длина эмотикона не может быть меньше двух символов. Данной проверкой отсеиваются одиночные знаки пунктуации, которые не являются концом предложения (запятая, тире и т.д.).

2. Последовательность из двух и более знаков, входящих в список допустимых для эмотикона символов, сравнивается со словарем эмотиконов и в случае обнаружения совпадения считается концом предложения. Если последовательность отсутствует в словаре и следующий символ не из списка знаков эмотиконов, она приравнивается к символам в составе предложения. Однако, если в группе встречалась точка, вопросительный или восклицательный знак, то последовательность считается концом предложения.

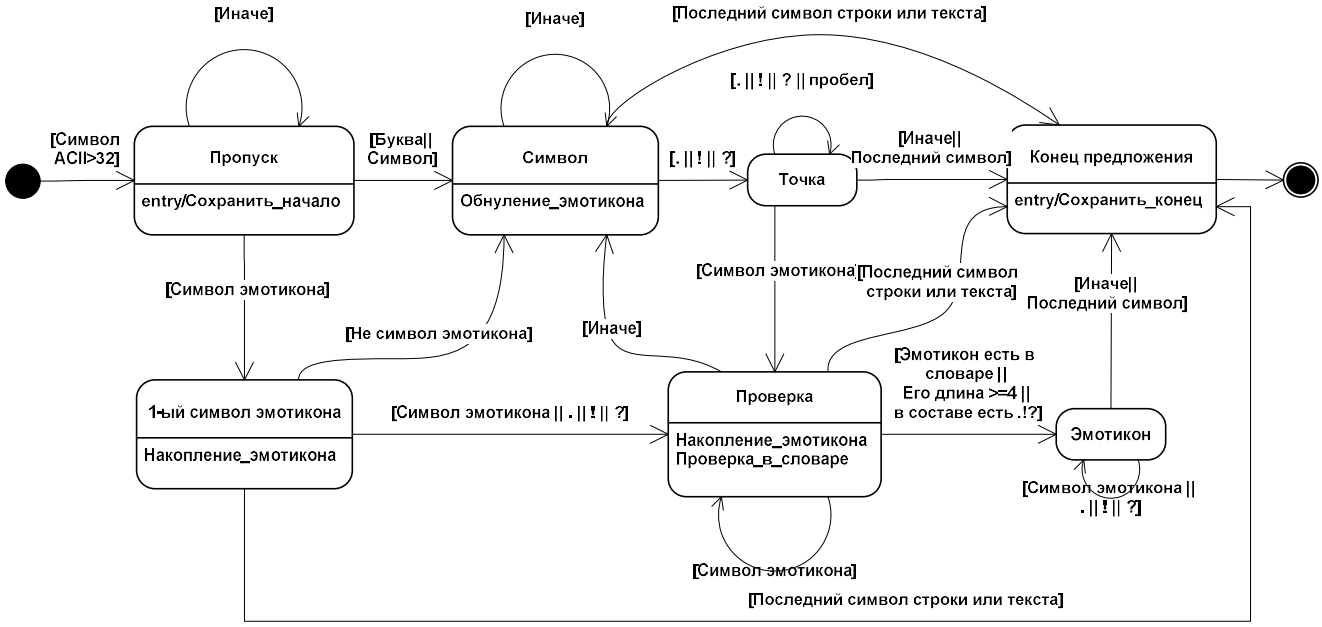

Рис. 1.4 - Алгоритм определения границ предложений в коротких электронных сообщениях

Задание

1. Изучить краткие теоретические сведения.

2. Разработать алгоритм определения конца предложения в текстах, написанных на русском языке. В алгоритме учесть многообразие русского языка. Например, многоточие (а также !!!, ??? и прочее), сокращения, смай-

лики, инициалы, точку как разделитель числа, прямую речь, отсутствие заглавных букв и прочее.

3. Написать программу, реализующую разработанный алгоритм. Язык программирования - Си, компилятор gcc. Сам алгоритм представить в виде функции, возвращающей позицию конца предложения в тексте:

int SentenceEnd(char * text);

или

int SentenceEnd(AnsiString text);

4. Предложить исчерпывающий набор тестов для проверки всех предусмотренных ситуаций, в которых алгоритм должен успешно справляться с выделением конца предложения.

5. Привести примеры ситуаций, которые разработанный алгоритм обрабатывает некорректно.

6. Написать отчет. В отчете представить блок-схему алгоритма, исходный текст программы, тесты и результаты тестирования.

Варианты индивидуальных заданий

Варианты заданий отсутствуют. Учитывается творческий подход.

Приветствуется использование дополнительных библиотек, классов и др. инструментов, реализующих морфологический, синтаксический и семантический анализ текста, а также различных словарей для повышения общего качества работы итогового алгоритма.

Контрольные вопросы

1. Основные элементы языка программирования.
2. Понятие «данные».
3. Понятие «оператор».
4. Понятие «синтаксис».
5. Понятие «семантика».
6. Понятие «условие».
7. Понятие «цикл».
8. Подпрограммы. Вызов подпрограмм.
9. Естественные и искусственные языки.
10. Конечные автоматы.
11. Лексический, синтаксический и семантический анализ при обработке текстов на естественных и искусственных языках.

Тема № 2

Анализ задачи. Абстракция программ и данных. Синтаксис языка программирования

Цель работы

Знакомство с основными элементами языка программирования, расширенной формой записи Бэкуса-Наура для записи синтаксиса языка программирования.

Краткие теоретические сведения

При проектировании сложных программных продуктов очень важно научиться мыслить системно и уметь разбивать программу на отдельные составляющие.

Программа – последовательность символов, определяющих вычисление.

Разработка программного обеспечения начинается с анализа технического задания, определения основных функций программы, входных и выходных данных (как программы в целом, так и для отдельных функций), определения возможных ошибок, методов их устранения и недопущения. Таким образом, производится ***декомпозиция программы*** на отдельные ***модули*** и определение ***интерфейсов*** взаимодействия модулей между собой.

Прежде всего, необходимо определить входные и выходные данные программы, их типы, диапазоны минимальные и максимальные значения.

После чего производится декомпозиция программ на модули. По возможности это должны быть полностью независимые составные части программы, выполняющие строго определенные функции. Каждый модуль имеет входные и выходные данные.

Также необходимо предусмотреть, какого рода ошибки возможны в процессе функционирования каждого из модулей. Как правило, существует три основных ошибки: ошибка при передаче параметров, переполнение (выход за диапазон), неправильное вычисление функции.

Приведем пример. Допустим, необходимо разработать программу, ознакомляющую пользователя с числами Фибоначчи, простыми, треугольными, квадратными, кубическими числами, числами – близнецами, треугольником Паскаля.

На рис. 2.1 приведена декомпозиция программы (условно назовем её «Числа»).

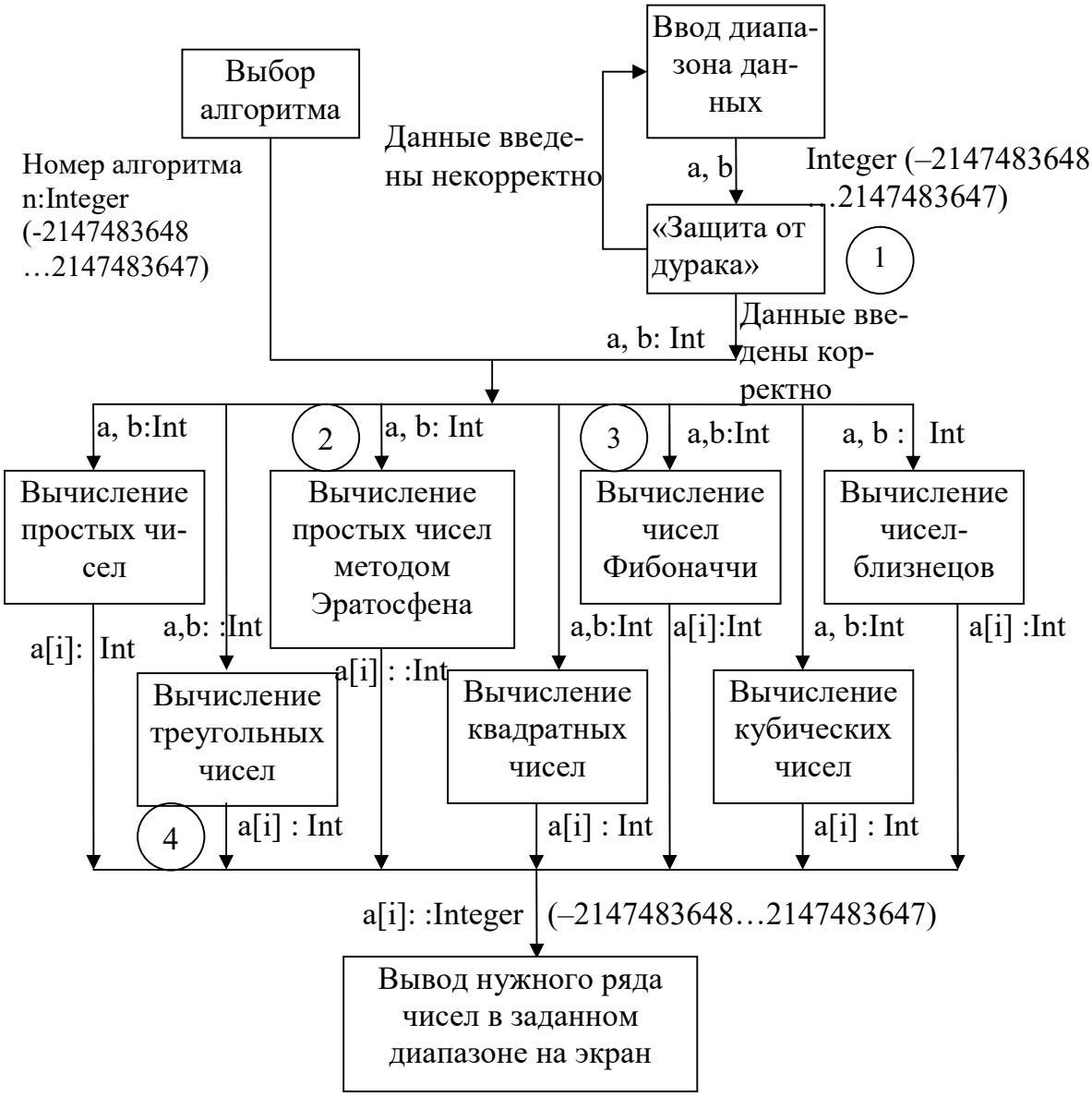

Рис. 2.1. Декомпозиция программы «Числа»

Опишем несколько из модулей программы.

Модуль 1*:*

1) Наименование – «Защита от дурака»;

Назначение – исключение ввода некорректных для задачи входных данных.

2) Входные данные: a – начало диапазона типа Integer (–2147483648 …2147483647), b – начало диапазона типа Integer.

3) Выходные данные: если некорректных данных найдено не было, программа переходит к следующему этапу, если найдены, то в диалоговом окне отображается ошибка и пользователю предлагается снова ввести диапазон.

4) Ошибки – переполнение, выход за диапазон.

Модуль 2:

1) Наименование – вычисление простых чисел методом Эратосфена (решето Эратосфена).

Назначение – создать массив, содержащий простые числа.

2) Входные данные: переменные для циклов *i*, *j* типа Integer, массив на 100000 элементов типа Integer.

3) Выходные данные: массив, содержащий простые числа от 1 до 100000.

4) Ошибки – 0 в качестве одной границы диапазона не допустим (из-за некорректности выражения рода *for i:=1 to 0*), переполнение массива.

Модуль 3:

1) Наименование – вычисление чисел Фибоначчи (*a[i]:=a[i–2]+a[i–1]*;).

Назначение – создать массив, содержащий числа Фибоначчи.

2) Входные данные: переменные для циклов i, j типа Integer, массив на 100000 элементов типа Integer.

3) Выходные данные: массив, содержащий числа Фибоначчи от 1 до 100000.

4) Ошибки – 0 в качестве одной границы диапазона не допустим (из-за некорректности выражения рода *for i:=1 to 0*), переполнение массива.

Модуль 4:

1) Наименование – вычисление треугольных чисел (*a[i]:=(i*(i+1)) div 2;*).

Назначение – создать массив, содержащий треугольные числа.

2) Входные данные: переменные для циклов i, j типа Integer, массив на 100000 элементов типа Integer.

3) Выходные данные: массив, содержащий треугольные числа от 1 до 100000.

4) Ошибки – 0 в качестве одной границы диапазона не допустим (из-за некорректности выражения рода *for i:=1 to 0*), переполнение массива.

После того, как функционал каждого из модулей (и программы в целом) определен, приходит время записать программу в виде исходного кода на одном из языков программирования с учетом синтаксиса этого языка.

Язык программирования – набор правил, описывающих, какие последовательности символов составляют программу, и какое вычисление производит программа.

Язык программирования применяется для составления программ, решающих задачи на определенном уровне абстракции. Появление и применение нового языка программирования может быть оправдано только тем, что он позволяет «легче» составить программу для конкретной области. При этом, чем выше уровень абстракции, тем больше деталей скрыто. Это

позволяет с меньшими затратами создавать программы для тех областей, где уже есть некоторые заготовки в виде написанных модулей, классов, шаблонов и т.д.

Синтаксис языка программирования – это набор правил, которые определяют, какие последовательности символов являются допустимыми.

Синтаксис задается с помощью формальной нотации. Самая распространенная нотация – ***расширенная форма Бекуса-Наура*** (РБНФ). Описание грамматики в РБНФ представляет собой набор правил, определяющих отношения между терминальными символами (терминалами) и нетерминальными символами (нетерминалами).

Терминальные символы – это минимальные элементы грамматики, не имеющие собственной грамматической структуры.

Нетерминальные символы – это элементы грамматики, имеющие собственные имена и структуру. Каждый нетерминальный символ состоит из одного или более терминальных и/или нетерминальных символов, сочетание которых определяется правилами грамматики.

Правила - имеют вид

идентификатор ::= выражение.

Где ***идентификатор*** **-** имя нетерминального символа, а ***выражение*** - комбинация терминальных и нетерминальных символов и специальных знаков. Правило завершается специальным символом - точкой.

В качестве выражения могут выступать:

1. ***Конкатенация***. Правило вида

$$A ::= BC.$$

обозначает, что нетерминал A состоит из двух символов **-** B и C. B и C называют ещё ***синтаксическими факторами***.

2. ***Выбор***. Правило вида

$$A ::= B|C.$$

обозначает, что нетерминал A может состоять либо из B, либо из C. Элементы выбора называют ещё ***синтаксическими термами.***

3. ***Условное вхождение*** - выделяет необязательный элемент выражения, который может присутствовать, а может и отсутствовать. Правило вида

$$A ::= [B].$$

обозначает, что нетерминал A либо является пустым, либо состоит из символа B. Условное вхождение.

4. ***Повторение*** - обозначает конкатенацию любого числа (включая ноль) записанных в ней элементов. Правило вида

$$A ::= \{B\}.$$

обозначает, что A - либо пустой, либо представляет собой конкатенацию любого числа символов B (A это либо пустой элемент, либо B, либо BB, либо BBB и т.д.).

5. ***Группировка*** - группировки элементов при формировании сложных выражений. Правило вида

A ::= (B|C)(D|E).

обозначает, что *A* состоит из двух символов, первый - либо *B*, либо *C*, второй - либо *D*, либо *E*.

6. ***Текстовый элемент***. Правило вида

'...'

обозначает символ или группу символов.

7. ***Комментарий***. Правило вида

(... *)*

8. ***Специальная последовательность*** символов. Правило вида

?...?

Общую форму грамматики РБНФ-описания можно описать в виде РБНФ следующим образом:

Синтаксис ::= { СинтОператор }.
СинтОператор ::= идентификатор "=" СинтВыражение ".".
СинтВыражение ::= СинТерм {"|" СинТерм}.
СинТерм ::= СинтФактор { СинтФактор }.
СинтФактор ::= идентификатор | цепочка
| "(" СинтВыражение ")" | "[" СинтВыражение "]"
| "{" СинтВыражение "}".

Приведем пример. Ниже приведен синтаксис языка программирования, подобного языку Pascal. Определены общий вид программы, абстракции идентификатора, числа, строки и т.д.

```
program ::= 'PROGRAM', white space, identifier, white space,
        'BEGIN', white space,
        { assignment, ";", white space },
        'END.' .
identifier ::= alphabetic character, { alphabetic character | digit } .
number ::= [ "-" ], digit, { digit } .
string ::= "'" , { all characters - "'" }, "'" .
assignment :: = identifier , ":=" , ( number | identifier | string ) .
alphabetic character ::= "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" |
                         "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" |
                         "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" |
                         "Y" | "Z" .
digit ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .
white space ::= ? white space characters ? .
all characters ::= ? all visible characters ? .
```

Синтаксически правильная программа, написанная на разработанном языке, тогда будет иметь вид:

```
PROGRAM EXAMPLE
 BEGIN
   X1:=1;
   X2:=5;
   X1:=X2;
   TEXT:="Hello!";
 END.
```

Задание

1. Изучить краткие теоретические сведения.
2. Получить вариант задания у преподавателя.
3. Для предложенной абстрактной программы описать её подробное назначение, входные и выходные данные программы, определить типы данных, их диапазон и размерность.
4. Декомпозировать программу на модули. Описать основные модули программы, их взаимосвязь.
5. Определить основные функции программы, входные данные, возвращаемые значения, их тип, диапазон и размерность.
6. Определить возможные ошибки, которые могут произойти при выполнении блоков программы.
7. Записать в простейшей форме синтаксис языка, позволяющего реализовать программу. Использовать для этого расширенную форму Бэкуса-Наура.
8. Привести пример программы с использованием синтаксиса разработанного языка программирования, реализующего основную функциональность по заданию.
9. Написать отчет и защитить у преподавателя.

Варианты заданий

1. Архиватор.
2. Простой текстовый редактор.
3. Калькулятор.
4. Проигрыватель.
5. Программа для записи CD, DVD.
6. Просмотрщик картинок.
7. Интернет-мессенджер.
8. Торрент-клиент.
9. Антивирус.
10. Интернет-браузер.
11. Интерпретатор командной строки.
12. SSH-клиент.
13. HTTP-сервер.

14. Почтовый клиент.
15. Пасьянс.
16. Менеджер закачек.
17. Межсетевой экран.
18. Переводчик.
19. Программа для тестирования производительности.
20. Игра «Сапер».
21. Игра «Тетрис».

Контрольные вопросы

1. Понятие «язык программирования».
2. Понятие «программа».
3. Языки программирования низкого и высокого уровней.
4. Ассемблеры и маркоассемблеры.
5. Отличия императивного и декларативного подхода к программированию.
6. Близость языков программирования к естественному языку.
7. Основные элементы языка программирования.
8. Понятие «среда программирования». Основные компоненты среды программирования.
9. Понятие «синтаксиса» языка программирования.
10. Расширенная форма Бекуса-Наура как способ задания синтаксиса языка программирования.
11. Понятие «семантики» языка программирования.
12. Трансляторы. Процесс трансляции. Компиляторы и интерпретаторы.

Тема № 3

Вещественные числа. Ошибки при работе с вещественными числами

Цель работы

Знакомство с основными ошибками, возникающими при обработке вещественных чисел.

Краткие теоретические сведения

Вещественные числа при обработке в вычислительной системе представляются в двоичном виде. Существует два подхода:

1) число представляется в согласно двоичной арифметике в виде числа с плавающей запятой. Такой подход называется двоично-кодированным десятичным числом.

2) Цифры числа кодируются как целое число, дополненное информацией о позиции десятичного разделителя. Этот формат числа называется форматом с фиксированной точкой.

Младший значащий разряд результата каждой операции с плавающей точкой может быть неправильным из-за ошибок округления. Программисты, которые пишут программное обеспечение для численных расчетов, должны хорошо разбираться в методах оценки и контроля этих ошибок.

Существует три основные ошибки вычислений, возникающие при выполнении операции над вещественными числами:

1) ***Исчезновение операнда*** - Операнд может исчезнуть, если он относительно мал по отношению с другим операндом.

Предположим, что мы имеем дело с десятичной арифметикой с пятью цифрами, тогда:

$$0{,}1234\cdot10^{3}+0{,}1234\cdot10^{-4}=0{,}1234\cdot10^{3}.$$

Второй операнд в данном вычислении просто потерялся.

2) ***Умножение ошибки*** – многократное увеличение абсолютной погрешности операнда, которая может появиться при использовании арифметики с плавающей точкой, даже если относительная ошибка мала. Обычно это является результатом умножения деления. Рассмотрим вычисление $x\cdot x$:

$$0{,}1234\cdot10^{3}\cdot0{,}1234\cdot10^{3}=0{,}1522\cdot10^{5}$$

и предположим теперь, что при вычислении x произошла ошибка на единицу младшего разряда, что соответствует абсолютной ошибке 0,1:

$$0{,}1235\cdot10^{3}\cdot0{,}1235\cdot10^{3}=0{,}1525\cdot10^{5}.$$

Абсолютная ошибка теперь равна 30, что в 300 раз превышает ошибку перед умножением.

3) ***Потеря значимости*** **-** полная потеря значимости, вызванная вычитанием почти равных чисел. Также возникает, когда результат вычислений невозможно представить в допустимой форме.

Пусть имеется два числа:

float f1= 0,12342;
float f2 = 0,12346;

В математике *f2 - f1* = 0,00004, что, вполне представимо как четырехразрядное число с плавающей точкой: $0{,}4000 \cdot 10^{-4}$. Однако программа, вычисляющая *f2 - f1* в четырехразрядном представлении с плавающей точкой, даст ответ:

$$0{,}1235 \cdot 10^{0} - 0{,}1234 \cdot 10^{0} = 0{,}1000 \cdot 10^{-3},$$

что даже приблизительно не является приемлемым ответом.

Приведем ещё один пример. Допустим, мы используем трехзначную арифметику и у нас есть число $x = 0{,}123 \cdot 10^{1}$.

Вычтем из него число $a = 0{,}122 \cdot 10^{1}$:

$$x - a = 0{,}1 \cdot 10^{-2}$$

теперь от полученного значения последовательно три раза отнимем $b = 0{,}2 \cdot 10^{-3}$. Получим: $0{,}8 \cdot 10^{-3}$, $0{,}6 \cdot 10^{-3}$, $0{,}4 \cdot 10^{-3}$.

А теперь попробуем вернуться к исходному результату, только сначала прибавим *a*, потом три раза *b*. В результате сначала мы получим число $0{,}1224 \cdot 10^{1}$, которое автоматически будет округлено до $0{,}122 \cdot 10^{1}$, т.к. это определяется форматом представления чисел. Ясно, что сколько бы раз мы теперь не прибавляли *b* к полученном результату, он не изменится. Т.о. от перестановки мест слагаемых сумма изменяется, если речь идет об ограниченной разрядной сетке.

Потеря значимости встречается намного чаще, чем можно было бы предположить, потому что проверка на равенство обычно реализуется вычитанием и последующим сравнением с нулем. Следующий условный оператор, таким образом, совершенно недопустим:

```
f1=...;
f2=...;
if (f1 ==f2)...
```

Простая перестройка выражений для *f1* и *f2*, независимо от того, сделана она программистом или оптимизатором, может вызвать переход в условном операторе по другой ветке. Правильный способ проверки равенства с плавающей точкой состоит в том, чтобы ввести малую величину и затем сравнить ***абсолютное значение разности с малой величиной*** (допустимой абсолютной погрешностью). По той же самой причине нет существенного различия между операторами «< =» и «<» при вычислениях с плавающей точкой.

Ошибки в вычислениях с плавающей точкой часто можно уменьшить изменением порядка действий. Поскольку сложение производится слева направо, четырехразрядное десятичное вычисление

$$1234,0+0,5678+0,5678 = 1234,0$$

лучше делать как:

$$0,5678+0,5678+1234,0 = 1234,0$$

чтобы не было исчезновения слагаемых.

Отметим ещё несколько важных моментов, касающихся операций с вещественными числами.

Для многих широко распространенных математических формул математики разработали ***специальную форму***, которая позволяет значительно уменьшить погрешность при округлении. Например, расчет формулы «x^2-y^2» иногда лучше вычислять используя формулу «(x-y)(x+y)».

Предположим

```
X, Y: Float_4; // тип с точностью представления 4 цифры
Z: Float_7; // тип с точностью представления 7 цифр
```

Вычислим значение Z по обеим формулам:

```
Z = Float_7((X + Y)*(X - Y));
Z = Float_7(X*X - Y*Y);
```

Если мы положим x = 1234,0 и y = 0,6, правильное значение этого выражения будет равно 1522755,64. Результаты, вычисленные с точностью до восьми цифр, таковы:

$$(1234,0 + 0,6) \bullet (1234,0-0,6) =1235,0 \bullet 1233,0=1522755,0$$

и

$$(1234,0 \bullet 1234,0)-(0,6 \bullet 0,6) = 1522756,0-0,36 =1522756,0$$

При вычислении $(x+y)(x-y)$ небольшая ошибка, являющаяся результатом сложения и вычитания, значительно возрастает при умножении. При вычислении по формуле x^2-y^2 уменьшается ошибка от исчезновения слагаемого и результат получается более точным.

Ещё одним подводным камнем может стать то, что округление до ближайшего в некоторых стандартах сделано не так как мы привыкли. Математически показано, что если 0,5 округлять до 1 (в большую сторону), то существует набор операций, при которых ошибка округления будет возрастать до бесконечности. Поэтому, например, в стандарте IEEE754 применяется ***правило округления до четного***. Так, 12,5 будет округлено до 12, а 13,5 – до 14.

Приведем пример нахождения ошибок «вручную».

Положим:

x=12345,678

y=0,000000987

$c=x+y$ =12345,678000987

Считаем, что количество значащих разрядов после запятой не может превышать 4.

Таким образом, получаем, что реальный ответ:

c'=12345,67800

Абсолютная ошибка $A=|c-c'|$=0,000000987

Относительная ошибка $O=A/c$=0,000000000007994700655

Предполагаем, что в младшем разряде числа x произошла ошибка.

$z=x$-0,001=12345,679

производим умножение без ошибки

$m=x\cdot x$=152415765,279684

с ошибкой

$n=x\cdot z$=152415777,625362

Абсолютная ошибка $A=|m-n|$=12,345678

Относительная ошибка $O=A/m$=0,000000081

Округлим x до целого числа

$k=$*округл.до целого(x)*=12346

Абсолютная ошибка $A=|x-k|$=0,322

Относительная ошибка $O=A/x$=0,000026082

Приведем примеры программы на языке программирования C#, демонстрирующие ошибки вещественных чисел.

Исчезновение операнда

```
Console.WriteLine("Исчезновение операнда. Тип float");
float a1 = 0.001f;
float b1 = 1000000f;
float result1 = a1 + b1;
Console.WriteLine("Результат - " + result1.ToString());
Console.WriteLine("Исчезновение операнда. Тип double");
double a2 = 0.00000001; // double точнее
double b2 = 1000000005; // увеличим разницу в операндах
double result2 = a2 + b2;
Console.WriteLine("Результат - " + result2.ToString());
Console.ReadLine();
```

Результат:

```
Ошибка 1 - Исчезновение операнда. Тип float
Результат - 1000000
Ошибка 1 - Исчезновение операнда. Тип double
Результат - 1000000005
```

Как видите, операнды *a1* и *a2* утеряны.

Умножение ошибки

```
Console.WriteLine("Умножение ошибки. Тип float и double");
float a5 = 0.0f;
double b5 = 0.0;
for (int i = 0; i < 10000; i++)
{
  a5 += 0.1f;
  b5 += 0.1;
}
Console.WriteLine("Результат - a = " + a5.ToString() + "; b =
" + b5.ToString());
Console.ReadLine();
```

Результат:

```
Ошибка 2 - Умножение ошибки. Тип float и double
Результат - a = 999,9029; b = 1000,00000000016
```

Переменные должны были содержать 1000, но на практике мы видим небольшие отличия. Происходит накопление ошибки из-за округления.

Потеря значимости

```
Console.WriteLine("Потеря значимости. Тип float");
float a3 = 1111111.1f;
float b3 = 1111111.0f;
float result3 = a3 - b3;
Console.WriteLine("Результат - " + result3.ToString());
Console.WriteLine("Потеря значимости. Тип double");
double a4 = 1111111.1;
double b4 = 1111111.0;
double result4 = a4 - b4;
Console.WriteLine("Результат - " + result4.ToString());
Console.ReadLine();
```

Результат:

```
Ошибка 3 - Потеря значимости. Тип float
Результат - 0,125
Ошибка 3 - Потеря значимости. Тип double
Результат - 0,100000000093132
```

При вычитании близких вещественных чисел мы также получаем неточности.

Задание

7. Изучить теоретические сведения.
8. В качестве исходных значений принять следующие значения:

x=<номер студенческого билета><номер группы>,<дата рождения ддммгггг>;

$y=x*10^{-10}$;

$c=x+y$;

c'= округление с до 9 знака после запятой;

$z=x\pm10^{-8}$;

k=округл.до целого(x).

9. Произвести вычисления, и показать каким образом возникают ошибки при работе с вещественными числами. Все значения вычислять с максимальной точностью, не округляя. Относительную ошибку вычислять с точность до 10 значащих цифр.

10. Объяснить полученные результаты.

11. Написать программы, демонстрирующие ошибки вещественных чисел на языках программирования, соответствующих варианту.

12. Написать отчет и защитить у преподавателя.

Варианты заданий

1. PHP.
2. Perl.
3. Ruby.
4. Java.
5. Pascal/Delphi.
6. Python.
7. C++.
8. Javascript.
9. Objective-C.
10. Scala.
11. C.
12. Visual Basic.
13. C#.
14. Golang.

Контрольные вопросы

1. Представление вещественных чисел.
2. Числа с фиксированной точкой.
3. Числа с плавающей точкой.
4. Смешанная арифметика.
5. Ошибки при работе с вещественными числами.
6. Исчезновение операнда.
7. Потеря значимости.
8. Умножение ошибки.

Тема № 4

Генерирование и обработка исключительных ситуаций

Цель работы

Изучение различных видов исключительных ситуаций и методов их обработки.

Краткие теоретические сведения

Исключение - это необычное, аварийное событие (исключительная ситуация), которое обнаруживается аппаратно или программно и требует специальной обработки (обработки исключения). Обработка производится обработчиком исключения (ловушкой исключения).

Исключительные ситуации могут возникать практически в любом месте программы, и поэтому организация явной проверки всех исключений невозможна или практически нецелесообразна. Примерами подобных ситуаций могут служить попытки деления на ноль или обращения к недоступным областям памяти.

Ошибки - исключительные ситуации, которые время от времени могут возникать в известных местах программы. Поэтому обнаружение ошибок должно предусматриваться логикой работы самой программы, в которую должны быть явно включены участки кода, ответственные за тестирование успешности работы и завершения основных потенциально проблемных операций.

Ниже перечислены характерные признаки участков программного кода, для которых целесообразно предусматривать отдельные обработчики исключений.

1. Возможность возникновения регистрируемых ошибок, включая ошибки системных вызовов, в условиях, когда необходимо организовать устранение последствий ошибки, а не предоставлять программе возможность прекращения выполнения.
2. Интенсивное использование указателей, повышающее вероятность попыток разыменования указателей, инициализация которых не была выполнена должным образом.
3. Интенсивное использование данных в виде массивов, что может сопровождаться выходом значений индексов элементов массива за границы допустимого диапазона.
4. В программе выполняются арифметические операции с участием вещественных чисел (чисел с плавающей точкой), и существует риск того, что могут возникать исключения, связанные с попытками деления на ноль, потерей точности при вычислениях и переполнением.

5. Наличие вызовов функций, которые могут генерировать исключения либо программным путем, либо в силу того, что их работоспособность не была достаточно тщательно проверена.

Таким образом, исключительные ситуации можно классифицировать следующим образом:

1. ***Ошибки ограничений*** (Constrain_Error) – выход за пределы диапазона, возникает при нарушении допустимого диапазона значений или индексов.

2. ***Ошибки памяти*** (Storage_Error) – недостаточность памяти.

3. ***Программные ошибки*** (Program_Error) – нарушение правил языка или некорректное поведении программы (в языке выделяют два случая: Bounded_Error - выход поведения за допустимые границы; Erroneous_Execution - проявление причин, ведущих к неправильному выполнению).

4. ***Ошибки задачи*** (Tasking_Error) – ошибки, возникающие при взаимодействии задач, асинхронных процессов.

Кроме того исключительные ситуации можно разделить на два основных типа: синхронные и асинхронные.

Синхронные исключения могут возникнуть только в определённых, заранее известных точках программы. Примерами таких исключений могут быть: ошибки чтения файла, нехватка памяти, ошибка деления на ноль и т.д.

Асинхронные исключения могут возникать в любой момент времени и не зависят от того, какую конкретно инструкцию программы выполняет система. Примерами таких исключений могут быть: аварийный отказ питания, поступление новых данных и т.д.

Семантика обработки исключений:

1. Если исключительная ситуация не обработана внутри процедуры, попытка ее выполнения оставляется, и исключительная ситуация снова возбуждается в точке вызова.

2. Если исключительная ситуация возбуждается во время выполнения обработчика, а обработчик оставляется, то исключение снова возбуждается в точке вызова.

3. Возможен выбор: возбудить то же самое исключение или другое в точке вызова.

Базовая технология обработки исключений заключается в использовании триады ***try…throw…catch*** (в зависимости от конкретного языка программирования операторы могут отличаться).

Нормальная ситуация обрабатывается кодом, следующим за ключевым словом ***try***, а код, располагающийся за ключевым словом ***catch*** (в некоторых языках ***except, resque***), выполняется только в исключительных случаях. Выбрасывание осуществляется с помощью оператора ***throw*** (в некоторых

языках ***raise***). При выбросе исключения выполнение блока try останавливается и начинается выполнение блока catch (обработчика исключения).

В некоторых языках программирования схема обработки исключительных ситуаций имеет вид ***try…catch…finally***. Оператор ***finally*** (в некоторых языках ***ensure***) – это обработчик завершения, который предоставляет удобные возможности для закрытия дескрипторов, освобождения ресурсов, восстановления масок и выполнения иных действий, направленных на восстановление известного состояния системы после выхода из блока. Блок ***finally*** выполняется всегда в независимости от того имела ли место быть исключительная ситуация или нет.

Ниже приведен пример кода на абстрактном языке программирования, демонстрирующий описанные выше блоки обработки исключительных ситуаций. В блоке *try* происходит деление двух целочисленных переменных. В случае если делитель равен нулю выбрасывается соответствующее этой ситуации исключение с помощью оператора *throw*, которое обрабатывается в первом блоке *catch*. На случай, если приведенный в блоке *try* код способен породить другие исключения, предусмотрен второй блок *catch*. Блок гарантированного завершения *finally* будет выполнен в любом случае.

```
try {
  int a,b;
  double result;
  if (b == 0) {
    throw new DevisionByZeroException("Divider is zero!");
    result = a/b;
  }
  console.printLine("The program ran successfully");
  console.printLine("Result is " + FloatToStr(result));
}
catch (DevisionByZeroException e) {
  console.printLine("Zero handler!");
}
catch (Exception e) {
  console.printLine("Error: " + e.message());
}
finally {
  console.printLine("The program terminates now");
}
```

Описанные схемы обработки имеют один очень важный недостаток – они позволяют уведомить о возникновении исключительной ситуации и спокойно продолжить работу, что в конечном счете может привести к неверным результатам. Поэтому в некоторых случаях целесообразно прервать

выполнение программы с выводом сообщения об ошибке, чем продолжить вычисление с некорректными данными.

Корректная схема обработки исключительных ситуаций была предложена Бертаном Мейером.

В её основе лежит подход к проектированию программной системы на принципах ***проектирования по контракту***. Модули программной системы, вызывающие друг друга, заключают между собой контракты. Вызывающий модуль обязан обеспечить истинность предусловия, необходимого для корректной работы вызванного модуля. Вызванный модуль обязан гарантировать истинность постусловия по завершении своей работы. Возникновение исключительной ситуации в вызванном модуле означает, что он не может выполнить свою часть контракта - корректно выполнить возложенную на него работу.

Обработчик исключительной ситуации, в свою очередь, имеет две возможности:

1. Попытаться внести коррективы и вернуть управление охраняемому модулю ***Retry***, который может предпринять очередную попытку выполнить свой контракт. В случае успеха и если работа модуля соответствует его постусловию, то исключительная ситуация рассматривается как временная успешно преодоленная проблема.

2. Если ситуация возникает вновь и вновь, тогда обработчик события применяет вторую стратегию ***Rescue***, выбрасывая исключение и передавая управление вызывающему модулю, который и должен теперь попытаться исправить ситуацию. Обработчик исключения должен позаботиться о восстановлении состояния, предшествующего вызову модуля, который привел к исключительной ситуации, и это гарантирует нахождение всей системы в корректном состоянии.

В такой схеме точно гарантируется, что не будет продолжена работа, когда контракт не выполняется.

Ниже приведен пример кода на абстрактном языке программирования, реализующего схему Б. Мейера:

```
Random rnd = new Random();
  int level = -10;
  bool Success; //true - нормальное завершение
  int count =1; // число попыток выполнения
  const int maxcount =3;

public void Pattern() {
  do {
    try {
      bool Danger = false;
      Success = true;
```

```
      MakeJob();
      Danger = CheckDanger();
      if (Danger)
        throw (new MyException());
          MakeLastJob();
    }
    catch (MyException me) {
      if(count > maxcount)
        throw(new MyException("Три попытки были
        безуспешны"));
      Success = false;
      count++;
      Console.WriteLine("Попытка исправить ситуацию!");
      level +=1;
  }
}
 while (!Success);
}

// класс исключений
public class MyException :Exception {
  public MyException() {}
  public MyException (string message) : base(message) {}
  public MyException (string message, Exception e) :
  base(message, e) {}
}

// подготовительные работы
void MakeJob() {
  Console.WriteLine("Подготовительные работы завершены");
}

// проверка возможности продолжения работ
bool CheckDanger() {
  int low = rnd.Next(level,10);
  if ( low > 6) return(false);
  return(true);
}

// выполнения оставшейся части работ
void MakeLastJob() {
  Console.WriteLine("Все работы завершены успешно");
}

// Пример вызова метода Pattern
public void TestPattern() {
  Excepts ex1 = new Excepts();
  try {
```

```
    ex1.Pattern();
  }
  catch (Exception e) {
    Console.WriteLine("исключение при вызове Pattern");
    Console.WriteLine(e.ToString());
  }
}
```

Конструкция *try-catch* блоков помещается в цикл *do-while(!Success)*, завершаемый в случае успешной работы охраняемого блока, за чем следит булева переменная *Success*.

Предполагается, что в блоке *try* анализируется возможность возникновения исключительной ситуации и, в случае её обнаружения, выбрасывается собственное исключение, класс которого задан программно. В соответствии с этим тело *try* блока содержит вызов метода *MakeJob,* выполняющего часть работы, после чего вызывается метод *CheckDanger*, определяющий, не возникла ли опасность нарушения спецификации и может ли работа быть продолжена. Если все нормально, то выполняется метод *MakeLastJob*, выполняющий оставшуюся часть работы. Управление вычислением достигает конца блока *try*, он успешно завершается и, т.к. переменная *Success* имеет значение *true*, то цикл while также успешно завершается.

Если в методе *CheckDanger* выясняется, что нормальное продолжение вычислений невозможно, то выбрасывается исключение класса *MyException*. Оно перехватывается соответствующим обработчиком исключения.

Пока не исчерпано число попыток, обработчик исключения присваивает переменной *Success* значение *false*, что гарантирует повтор выполнения блока *try*, увеличивает счетчик числа попыток и пытается исправить ситуацию.

Задание

1. Изучить краткие теоретические сведения.
2. Разработать программу, генерирующую исключительную ситуацию, и обрабатывающую вызванное исключение на языках программирования, соответствующих варианту.
3. Доработать написанную программу, применив схему обработки исключительных ситуаций Б. Мейера.
4. Написать отчет и защитить у преподавателя.

Контрольные вопросы

1. Понятие «исключительная ситуация».
2. Синхронные и асинхронные исключения.
3. Структурная обработка исключений.

4. Неструктурная обработка исключений.
5. Механизмы обработки исключений с возвратом и без возврата.
6. Блоки с гарантированным завершением.
7. Алгоритм генерирования и распознавания исключений.
8. Иерархия исключений.
9. Схема Бертрана Мейера обработки исключительных ситуаций.

Варианты заданий

1. PHP.
2. Perl.
3. Ruby.
4. Java.
5. Pascal/Delphi.
6. Python.
7. C++.
8. Javascript.
9. Objective-C.
10. Scala.
11. C.
12. Visual Basic.
13. C#.
14. Golang.

Тема № 5

Рекурсия. Типы рекурсий

Цель работы

Изучение различных типов рекурсий и способов их применения для решения практических задач.

Краткие теоретические сведения

Функция называется ***рекурсивной***, если во время ее обработки возникает ее повторный вызов, либо непосредственно, либо косвенно, путем цепочки вызовов других функций.

Рассмотрим простой пример рекурсии – ряд Фибоначчи: 1, 1, 2, 3, 5, 8, 13, 21, 34…

Формально функция для этого ряда имеет вид:

$$F(0)=1,$$
$$F(1)=1,$$
$$F(n)=F(n-2)+F(n-1), \text{ для } n>2.$$

Дерево рекурсивных вызовов для вычисления 6-го элемента ряда будет выглядеть следующим образом (см. рис. 5.1).

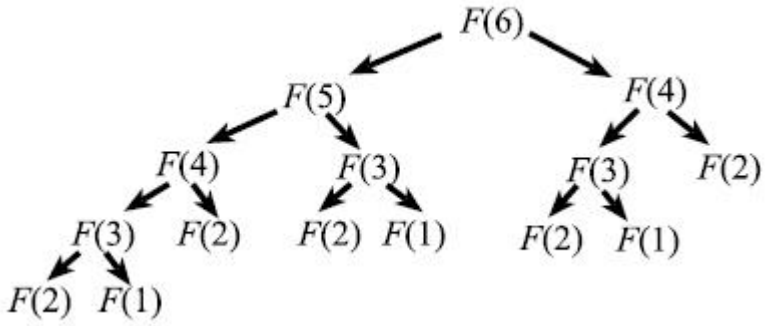

Рис. 5.1. Дерево рекурсивных вызовов

При вычислении значения *F(6)* будут вызваны подпрограммы вычисления *F(5)* и *F(4)*. В свою очередь, для вычисления последних потребуется вычисление двух пар *F(4)*, *F(3)* и *F(3)*, *F(2)*.

Основным недостатком рекурсивных вычислений являются повторные вычисления. В примере с числами Фибоначчи, например, только значение *F(3)* вычисляется три раза. Т.о. вычисление при больших значениях *n* могут потребовать значительных «лишних» расчетов. Для решения этой проблемы необходимо ввести дополнительную структуру данных для запоминания значений, которые уже были вычислены ранее. Такая рекурсия с запоминанием называется ***динамическим программированием сверху***.

Рекурсивное определение функции должно предусматривать ***критерий останова***, иначе дерево рекурсивных вызовов может оказаться бесконечным.

Рассмотрим основные типы рекурсий.

Рекурсия, при которой рекурсивные вызовы на любом рекурсивном срезе, инициируют не более одного последующего рекурсивного вызова, называется ***линейной***. Это наиболее простой и часто встречающийся тип рекурсии.

Частный случай линейной рекурсии с отсутствующими предварительными или отложенными вычислениями называется ***повторительной*** рекурсией.

Рекурсию называют ***каскадной***, если она не является линейной. Рекурсивные вызовы могут возникнуть более одного раза, образуя древовидную схему вызовов.

Рекурсия называется ***удаленной***, если в теле функции при рекурсивных вызовах, в выражениях, являющихся фактическими параметрами, снова встречаются рекурсивные вызовы этой функции.

Циклическая последовательность вызовов нескольких функций F_1, F_2, …, F_k (процедур, алгоритмов) друг друга: F_1 вызывает F_2, F_2 вызывает F_3, …, F_k вызывает F_1 (k>1), называют ***косвенной*** или ***взаимной*** рекурсией.

В качестве примера, приведем исходный код, реализующий пять вышеозначенных видов рекурсии на языке Pascal.

1. Линейная рекурсия:

```
Program Lin_Rec;
Uses crt;
Var a,b:integer;
function lin(a,b:integer):integer;
Begin
 if a<b then lin:=0
 else lin:=lin(a-b,b)+1;
End;
Begin
 ClrScr;
 Writeln('Введите a и b ');
 Readln(a,b);
 Writeln(lin(a,b));
 Readln;
End.
```

2. Повторная рекурсия:

```
Program pov_rec;
Uses Crt;
Var a,b,c: integer;
function
pov(a,b,c:integer):integer;
Begin
 if a<b then pov:=c
 else pov:=pov(a-b,b,c+1);
c:=0;
End;
Begin
 Clrscr;
 Writeln(',Введите a и b');
 Readln(a,b);
 Writeln(pov(a,b,c));
 Readln;
End.
```

3. Взаимная рекурсия:

```
Program vzaim_rec;
Uses crt;
var
 a:integer;
```

```
function vzaim1(a:integer):real;
forward;
function vzaim2(a:integer):real;
begin
 if a>0 then vzaim2:=vzaim1(a-1)
 else vzaim2:=1;
end;
function vzaim1(a:integer):real;
begin
 if a=0 then vzaim1:=3
else if (a mod 2 = 0) then
 vzaim1:=vzaim2(a-1) + 1
else vzaim1:=vzaim1(a-1);
end;
Begin
 Clrscr;
 Writeln('Введите a');
 Readln(a);
 Writeln(vzaim1(a):5:2);
 Readln;
End.
```

4. Каскадная рекурсия:

```
Program Cascad;
Uses Crt;
Var a:integer;
function Cas(a:integer):integer;
Begin
 if (a=0) or (a=1) then
  Cas:=1
 else Cas:=Cas(a-1)+Cas(a-2);
End;
Begin
 Writeln('Введите a');
 Readln(a);
 Writeln(Cas(a));
 Readln;
End.
```

5. Удаленная рекурсия:

```
Program Udal_Rec;
Uses Crt;
Var a,b:integer;
function
udal(a,b:integer):integer;
Begin
 if a=0 then udal:=b+1
 else if b=0 then
  udal:=udal(a-1,1)
 else
 udal:=udal(a-1,udal(a,b-1));
End;
Begin
 ClrScr;
 Writeln('Введите a и b');
 Readln(a,b);
 Writeln(udal(a,b));
 Readln;
End.
```

Задание

1. Изучить краткие теоретические сведения.
2. Разработать программу или комплекс программ, реализующих все типы рекурсий на языке программирования, соответствующем варианту.
3. Написать отчет и защитить у преподавателя.

Контрольные вопросы

1. Понятие «подпрограмма». Основные составляющие подпрограммы.
2. Процедуры и функции.
3. Формальные и фактические параметры подпрограммы. Способы передачи фактического параметра.
4. Понятие «рекурсия»
5. Дерево рекурсивных вызовов.
6. Динамическое программирование сверху.

7. Линейная рекурсия.
8. Повторная рекурсия.
9. Каскадная рекурсия.
10. Взаимная рекурсия.
11. Удаленная рекурсия.
12. Недостатки рекурсивного метода решения задач.

Варианты заданий

1. PHP.
2. Perl.
3. Ruby.
4. Java.
5. Pascal/Delphi.
6. Python.
7. C++.
8. Javascript.
9. Objective-C.
10. Scala.
11. C.
12. Visual Basic.
13. C#.
14. Golang.

Тема № 6

Указатели и ссылки

Цель работы

Изучение механизмов работы указателей и ссылок.

Краткие теоретические сведения

Указатели

Указатели – это переменная, значением которой является адрес другой переменной. Указатель ссылается на блок данных из области памяти, причём на самое его начало.

Значение указательного (ссылочного) типа (pointer type) – это адрес.

Объект, на который указывают, называется **указуемым** или **обозначаемым объектом** (designated object).

Различают типизированные и нетепизированные (void *) указатели, указатели на данные и на функции. При этом множества допустимых операций для указателей разных видов различны.

Размер (формат) адреса зависит только от величины адресного пространства компьютера и специфики его организации.

Указатель позволяет предоставить косвенный доступ к элементам известного типа (**типизированное средство косвенного доступа к объектам данных**).

В общем виде объявление указателя в языке выглядит следующим образом:

/*объявление указателя*/;
/*тип данных*/ * /*имя указателя*/;

Приведем пример.

```
int var = 123;
int *ptrvar;     // объявление указателя
ptrvar = &var;   // присвоили адрес переменной указателю
cout << &var;    // адрес переменной var 0x22ff08
cout << ptrvar;  // адрес переменной var 0x22ff08
cout << var;     // значение в переменной var  123
cout << *ptrvar; // вывод значения содержащегося в
переменной var через указатель, операцией разименования
указателя  123
```

Основными операциями с указателями являются:

1. **Объявление**. *int *pa; float *pb.*
2. **Получение адреса переменной**. *p = & a.*

3. **Разыменование**. *x = *p* - получаем объект, на который указатель ссылается.

4. **Получение адреса указателя**. *&p*.

5. **Увеличение указателя на единицу**. *++p* (перемещение к адресу следующего элемента массива).

6. **Вычитание**. *p2 – p1* (результат отображается в тех же единицах, что и размер данного типа переменной).

Для нетипизированных указателей запрещены операция разадресации, инкремента и т.п.

Для указателя на функции определена операция вызова функции. А само объявление указателя имеет следующий вид

/*тип данных*/ (* /*имя указателя*/)(/*список аргументов функции*/);

Приведем несколько примеров объявлений более сложных указателей.

```
int a;           // Целое число
int *a;          // Указатель на целое
int **a;         // Указатель на указатель на целое
int a[10];       // Массив из десяти целых
int *a[10];      // Массив из десяти указателей на целые
int (*a)[10];    // Указатель на массив из десяти целых
int (*a)(int);   // Указатель на функцию, которая берет
                 // целый аргумент и возвращает целое
int (*a[10])(int);  // Массив из десяти указателей на
                 // функции, которые берут целый
                 // аргумент и возвращают целое
```

При работе с указателями важно четко понимать разницу между самим указателем (как переменной) и указуемым объектом. Приведем несколько примеров, иллюстрирующих эту разницу.

```
int i1 = 10;
int i2 = 20;
int *ptr1 = &i1; // ptr1 указывает на i1// ptr1 = 0x22ff04
int *ptr2 = &i2; // ptr2 указывает на i2// ptr2 = 0x22ff00
if (ptr1 > ptr2) {}; // истина т.к. 0x22ff04>0x22ff00
if (*ptr1 > *ptr2){}; // сравниваем сами переменные, false
*ptr1 = *ptr2;   // косвенно приравниваем i1=i2=20.
if (ptr1 == ptr2)    {}; // false, указатели до сих пор
разные
if (*ptr1 == *ptr2) {}; // true, обознач. объекты равны
ptr1 = ptr2; // указывают на i2, ptr1=ptr2=0x22ff00
```

Указатели могут ссылаться на другие указатели.

При этом в ячейках памяти, на которые будут ссылаться первые указатели, будут содержаться не значения, а адреса вторых указателей.

Число символов * при объявлении указателя показывает **порядок указателя**.

Чтобы получить доступ к значению, на которое ссылается указатель его необходимо разыменовывать соответствующее количество раз.

```
int var = 123; // инициализация переменной var числом 123
int *ptrvar = &var; // указатель на переменную var
int **ptr_ptrvar = &ptrvar; // указатель на ук. на var
int ***ptr_ptr_ptrvar = &ptr_ptrvar; // ук. на ук. на
                                     // указатель
cout << var; //123
cout << *ptrvar; //123
cout << **ptr_ptrvar; // два раза разименовываем указатель
                      // чтобы получить значение 123
cout << ***ptr_ptr_ptrvar;; // три раза разыменовывам
                            // чтобы получить 123
cout << "\n ***ptr_ptr_ptrvar -> **ptr_ptrvar -> *ptrvar -
> var -> "<< var << endl;
cout << "\t  " << &ptr_ptr_ptrvar<< " -> " << "    " <<
&ptr_ptrvar << " ->" << &ptrvar << " -> " << &var << " ->
" << var << endl;
```

Результатом выполнения двух последних строк будет

```
***ptr_ptr_ptrvar -> **ptr_ptrvar -> *ptrvar -> var -> 123
0x22ff00 ->     0x22ff04 ->0x22ff08 -> 0x22ff0c -> 123
```

Важно также понимать различие между **указателем-константой** и **указателем на константный объект**. В первом случае можно менять значение указуемого объекта, но нельзя менять значение самого указателя и переназначать его на другую область памяти. Во втором случае значение указуемого объекта менять нельзя, но сам указатель можно переназначать на другую область памяти.

Приведем примеры, демонстрирующие эти различия.

```
int i1, i2;
/*
указатель-константа. Переназначить на другую область
памяти нельзя, но значение переменной менять можно
*/
int * const p1 = &i1;
/*
```

```
указатель на const. Нельзя модифицировать  значение
переменной по этому адресу.
*/
const int* p2 = &i1;
/*
указатель-константа на константу. Нельзя переназначить
адрес, нельзя изменить значение по этому указателю.
*/
const int* const p3 = &i1; // p1 = &i2; // ошибка,
указатель-константа
*p1 = 5;     // правильно, ук. объект не является const
p2 = &i2;    // правильно, указатель не является const
*p2 = 5;     // ошибка, указуемый объект – константа
p3 = &i2;    // ошибка, указатель-константа
*p3 = 5;     // ошибка, указуемый объект - константа
```

Типизированные указатели неявно могут быть преобразованы в указатели на void. Обратное преобразрвание может привести к ошибке.

```
void *void_ptr;  // нетепизированный указатель
int *int_ptr;    // типизированный ук. на тип int
char *char_ptr;  // типизированный указатель на char
void_ptr = int_ptr;  // правильно
char_ptr = void_ptr; // правильно в C, но ошибка C++
char_ptr = int_ptr;  // предупреждение в C, ошибка в C++
```

Основные преимущества использования указателей:

1. Повышение эффективности. Вместо копирования или пересылки в памяти большой структуры данных можно скопировать или переслать только указатель на эту структуру.

2. Динамические структуры. С помощью записей и указателей можно реализовать структуры данных, которые растут и сжимаются в период выполнения программы. Например, такие как списки и деревья. Помимо элементов данных самой структуры, узел содержит один или несколько указателей со ссылками на другие узлы.

3. Доступ к большим структурам данных. Используют для косвенного обращения к большим структурам данных, повышая скорость и сокращая затраты на организацию доступа.

Например для доступа к элементам массива можно использовать как явное обращение, так и косвенную адресацию, как это показано в следующем примере.

```
int *ptr;
int a[100];
ptr = &a[0];    // явный адрес первого элемента
ptr =a;         // неявный адрес первого элемента
*(ptr+1);       // эквивалент операции  a[i]
```

Ссылки

Ссылка - тип данных, являющийся скрытой формой указателя, который при использовании **автоматически разыменовывается**.

Основное назначение ссылки - организации **прямого** доступа к тому, или иному объекту, в отличии от указателя, который предоставляет **косвенный** доступ. Главное отличие состоит во внутреннем механизме работы: указатели ссылаются на участок в памяти, используя его адрес, а ссылки ссылаются на объект, по его имени (тоже своего рода адрес).

Ссылка может быть объявлена как другим именем, так и псевдонимом переменной, на которую ссылается.

/*объявление ссылки*/;
/*тип*/ &/*имя ссылки*/ = /*имя переменной*/;

Переменная, на которую ссылается ссылка называется **ссылочной переменной**.

Изменение значения содержащегося в ссылке влечёт изменение этого значения в переменной, на которую ссылается ссылка.

Приведем пример, иллюстрирующий работу с указателями.

```
int value = 15;
int &reference = value; // инициализация ссылки значением
                        // переменной value
cout << "value = " << value << endl; // value ==15
cout << "reference = " << reference << endl; // reference=
                                              // =15
reference+=15; // изменяем значение переменной value
               // посредством изменения значения в ссылке
cout << "value = " << value  << endl; // значение
               // поменялось как в ссылке value ==30 ;
cout << "reference = " << reference << endl; //  так и в
               // ссылочной переменной reference==30
```

Задание

1. Изучить краткие теоретические сведения.

2. Подготовить примеры использования указателей и ссылок для языка C++ и языка программирования, определенного выбранным вариантом. Сравнить возможности языков. При этом продемонстрировать:

2.1 разницу между типизированными и нетипизированными указателями;

2.2 разницу между указателями на данные и на функции;

2.3 разницу между указателем и указуемым объектом;

2.4 разницу между указателями-константами и указателями на константу;

2.5 особенности работы с многоуровневыми указателями;

2.6 разницу между указателями и ссылками.

3. Написать отчет и защитить у преподавателя.

Контрольные вопросы

1. Понятие «указатель».
2. Понятие «ссылка».
3. Отличия указателей и ссылок.
4. Ссылки на данные и ссылки на функции.
5. Типизированные и нетипизированные указатели.
6. Указатель константа и указатель на константу.
7. Операции с указателями.
8. Операции со ссылками.
9. Преимущества использования указателей и ссылок.
10. Прямая и косвенная адресация.
11. Динамическая память.
12. Повисшие указатели.
13. Утечки памяти.

Варианты заданий

1. PHP.
2. Perl.
3. Ruby.
4. Java.
5. Object Pascal/Delphi.
6. Python.
7. C/C++.
8. JavaScript.
9. Objective-C.
10. Scala.
11. Visual Basic.
12. C#.
13. Golang.

14. Swift
15. Groovy
16. Go

Тема № 7

Объектно-ориентированное программирование

Цель работы

Знакомство с основными концепциями и приемами объектно-ориентированного анализа и проектирования, выработка практических навыков в построении модели предметной области и элементов модели проектирования.

Краткие теоретические сведения

1. Классы и объекты

Объектно-ориентированное программирование **(ООП) –** это методология программирования, основанная на представлении программы в виде совокупности ***объектов***, каждый из которых является экземпляром определенного ***класса***, а классы образуют ***иерархию наследования***.

Т.о. ***класс*** – это абстрактный тип данных, снабженный некоторой реализацией. Полностью реализованный класс называется ***эффективным***. Класс, который реализован лишь частично или не реализован вовсе называется ***отложенным***.

Различают внутреннее представление класса (реализацию) и внешнее представление класса (интерфейс).

Интерфейс объявляет возможности (услуги) класса, но скрывает его структуру и поведение.

Реализация класса описывает поведение класса. Она включает реализации всех операций, определенных в интерфейсе класса.

Атрибуты класса определяют состав и структуру данных, хранимых в объектах этого класса. Каждый атрибут имеет имя и тип, определяющий, какие данные он представляет.

Класс содержит объявления ***операций***, представляющих собой определения запросов, которые должны выполнять объекты данного класса. Реализация операции в виде процедуры – это ***метод***, принадлежащий классу.

Существует несколько специальных видов классов.

Абстрактный класс – это класс, который не может иметь экземпляров. В них присутствуют функции члены, которые объявлены, но не определены. Эти функции определяются в наследниках, которые уточняют данную абстракцию.

Интерфейс – это абстрактный класс, который содержит только объявление методов и статические константные поля.

Класс-утилита – это класс, в котором присутствуют только статические члены. Используются для группировки наиболее часто используемых алгоритмов.

Объект – это экземпляр некоторого класса.

При реализации объекта в программном коде для атрибутов будет выделена память, необходимая для хранения всех атрибутов, и каждый атрибут будет иметь конкретное значение в любой момент времени работы программы. Объектов одного класса в программе может быть сколь угодно много, все они имеют одинаковый набор атрибутов, описанный в классе, но значения атрибутов у каждого объекта свои и могут изменяться в ходе выполнения программы.

Свойства объектов:

1) ***индивидуальность*** – это характеристика объекта, которая отличает его от всех других объектов.

2) ***состояние*** – характеризуется перечнем всех атрибутов объекта и текущими значениями каждого из этих атрибутов.

3) ***поведение*** – это его деятельность с точки зрения воздействия на другие объекты или подверженности воздействиям со стороны других объектов.

Возможны пять видов операций клиента над объектом:

1) ***модификатор*** - изменяет состояние объекта;

2) ***селектор*** - дает доступ к состоянию, но не изменяет его;

3) ***итератор*** - доступ к содержанию объекта по частям, в строго определенном порядке;

4) ***конструктор*** - операция создания объекта и/или его инициализации;

5) ***деструктор*** - операция, освобождающая состояние объекта и/или разрушающая сам объект.

2. Основные принципы ООП

1. ***Абстрагирование*** – это способ выделить набор существенных характеристик объекта, исключая из рассмотрения незначимые, которые позволяют отличить его от всех других видов объектов (определить его концептуальные границы). Соответственно, абстракция – это набор всех таких характеристик. Иными словами абстрагирование – это способ сконцентрироваться на интерфейсе (внешнее поведение объекта), не обращая внимания на реализацию (механизмы достижения желаемого поведения объекта).

2. ***Инкапсуляция*** – это свойство системы, позволяющее объединить данные и методы, работающие с ними, в классе и скрыть детали реализации от пользователя.

3. ***Полиморфизм*** – это свойство системы использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

4. ***Наследование*** – это свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью. Класс, от которого производится наследование, называется базовым или родительским. Новый класс – потомком, наследником или производным классом.

5. ***Модульность*** – свойство системы, которая была разложена на внутренне связанные, но слабо связанные между собой модули.

6. ***Типизация*** – это способ защититься от использования объектов одного класса вместо другого, или по крайней мере управлять таким использованием.

7. ***Сохраняемость*** – способность объекта существовать во времени, переживая породивший его процесс, и (или) в пространстве, перемещаясь из своего начального адресного пространства.

8. ***Параллелизм*** – свойство, отличающее активные объекты от пассивных.

3. UML. Отношения между классами

Модель предметной области – описание объектов задачи, такое как если бы мы и не собирались писать никакую программу. На модели предметной области приводятся типы объектов, их атрибуты, связи между объектами разных типов.

Модель проектирования – это описание объектов системы, приближенное к реализации в выбранной среде программирования (на конкретном языке программирования) с учетом применяемых технологий (например, базы данных).

Логическое моделирование классов для конкретной предметной области, как правило, осуществляется с помощью языка UML.

UML **(*Unified Modeling Language*,** унифицированный язык моделирования) – это язык графического описания для объектного моделирования в области разработки программного обеспечения. UML является языком широкого профиля, это открытый стандарт, использующий графические обозначения для создания абстрактной модели системы, называемой ***UML моделью***. UML был создан для определения, визуализации, проектирования и документирования в основном программных систем. Однако UML используют также для моделирования бизнес-процессов, системного проектирования и отображения организационных структур.

Всего существует 13 типов диаграмм:

1. диаграмма активности;
2. диаграмма классов;

3. диаграмма связей;
4. диаграмма компонентов;
5. диаграмма составных структур;
6. диаграмма развертывания;
7. диаграмма обзора взаимодействий;
8. диаграмма объектов;
9. диаграмма пакетов;
10. циклограмма;
11. диаграмма машин состояния;
12. диаграмма синхронизации;
13. диаграмма прецедентов.

Диаграмма классов описывает статическую структуру системы, показывая её классы, их атрибуты и методы, и также взаимосвязи этих классов.

Изображение класса в нотации UML представлено на рис. 6.1.

Рис. 6.1. Изображение класса в нотации UML

Класс на диаграмме показывается в виде прямоугольника, разделенного на 3 области. В верхней содержится название класса, в средней – описание атрибутов (свойств), в нижней – названия методов (операций). Имена абстрактных классов записываются курсивом. В зависимости от ситуации секции атрибутов и операций классов могут не отображаться.

Общий синтаксис представления атрибута имеет вид:

Видимость Имя : Тип [Множественность] = НачальноеЗначение {Свойства}

Для каждого атрибута класса можно задать ***видимость*** (visibility). Эта характеристика показывает, доступен ли атрибут для других классов. В UML определены следующие уровни видимости атрибутов:

– ***открытый*** (***public***, «+») – виден для любого другого класса (объекта);

– ***пакетный*** (***package***, «~») - виден для клиента класса, объявленного в том же пакете;

– ***защищенный*** (***protected***, «#») – виден внутри самого класса, для потомков данного класса;

– ***закрытый*** (***private***, «-») – не виден внешними классами (объектами) и может использоваться только объектом, его содержащим.

Другом класса называют класс (функцию), который (которая) имеет доступ ко всем частям этого класса (публичной, защищенной и приватной).

Для операций, как и для атрибутов класса, определено понятие «видимость». Закрытые операции являются внутренними для объектов класса и недоступны из других объектов. Остальные образуют интерфейсную часть класса и являются средством интеграции класса в программную систему.

Общий синтаксис представления операции имеет вид:

Видимость Имя (Список Параметров): ВозвращаемыйТип {Свойства}

В сигнатуре операции можно указать ноль или более параметров, форма представления параметра имеет следующий синтаксис:

Направление Имя : Тип = ЗначениеПоУмолчанию

Элемент *Направление* может принимать одно из следующих значений:

– in - входной параметр, не может модифицироваться;

– out - выходной параметр, может модифицироваться для передачи информации в вызывающий объект;

– inout - входной параметр, может модифицироваться.

Предусмотрено задание области действия атрибута (операции). Если атрибут (операция) подчеркивается – его областью действия является класс, в противном случае – областью действия является экземпляр

Классы, как правило, не являются обособленными и вступают с другими классами в отношения. Диаграмма классов UML позволяет обозначать отношения между классами и их экземплярами. Основные из них рассмотрим на примере.

В качестве предметной области возьмем процесс обучения в университете. UML-диаграмма представлена на рис. 6.2.

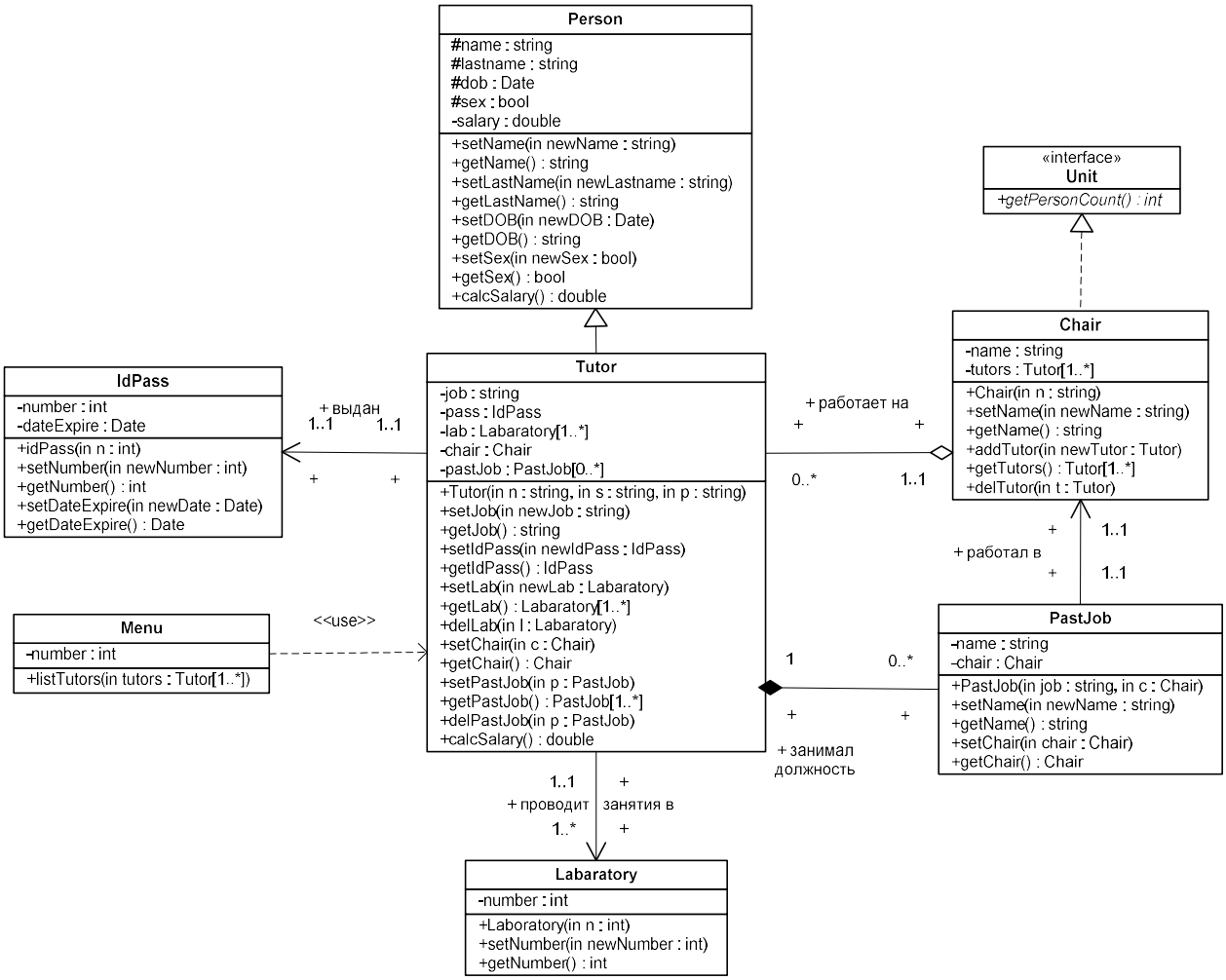

Рис. 6.2. Модель предметной области

1. ***Обобщение***.

Отношение обобщения схоже с наследованием.

Обобщение – это отношение между общей сущностью и специализированной разновидностью этой сущности.

Наследование – это отношение, при котором один класс разделяет структуру и поведение, определенные в другом классе (одиночное наследование) и нескольких других классах (множественное наследование).

Класс `Person` (человек) – более абстрактный, а `Tutor` (преподаватель) более специализированный (см. рисунок 6.3). Здесь стрелка в виде треугольника означает наследование. Звучит это примерно так: «класс `Tutor` является потомком класса `Person`, т.е. включают в себя все определенные в `Person` атрибуты и операции». Повторение имени операции `CalcSalary` у потомка означает, что у него эта операция будет «перекрыта» – написана по-другому при том же названии и параметрах, при этом сам метод `CalcSalary` является ***виртуальным***.

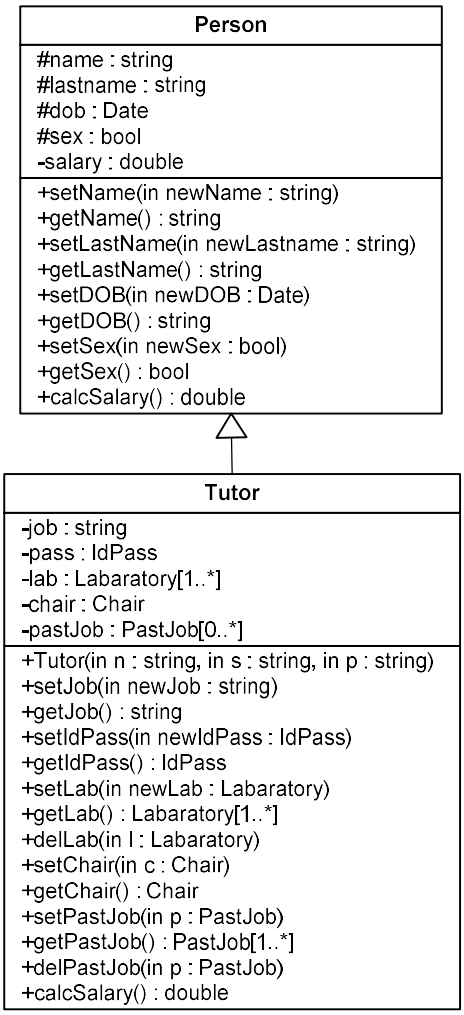

Рис. 6.3. Обобщение

Сделаем заготовки для этих двух классов:

```
public class Person{
    protected String name;
    protected String lastname;
    protected double salary;
    public void setName(String newName){
        name = newName;
    }
    public String getName(){
        return name;
    }
    public void setlastname(String newlastname){
        name = newlastname;
    }
    public String getlastname(){
        return lastname;
    }
```

```
    // виртуальный метод
    abstract double calcSalary();
}

// наследуем класс Person
public class Tutor extends Person{
    private String job;
    // создаем и конструктор
    public Tutor(String n, String s, String p, double sal){
        name = n;
        lastname = s;
        position = p;
      salary = sal;
    }
    public void setJob(String newJob){
        position = newJob;
    }
    public String getJob(){
        return job;
    }
    public double calcSalary(){
        return salary;
    }

}
```

2. ***Ассоциация***

Ассоциация показывает структурные отношения между объектами-экземплярами класса, т.е. соединения между классами.

2.1 ***Бинарная ассоциация***

В модель добавлен класс `IdPass`, представляющий пропуск преподавателя. Каждому преподавателю может соответствовать только один пропуск, мощность связи «один к одному».

Класс `Tutor` имеет поле `Pass`, у которого тип `IdPass`, так же класс имеет методы для присваивания значения (`setIdPass`) этому полю и для получения значения(`getIdPass`). Из экземпляра объекта `Tutor` мы можем узнать о связанном с ним объектом типа `IdPass`, значит навигация (стрелка на линии) направлена от `Tutor` к `IdPass` (см. рисунок 6.4).

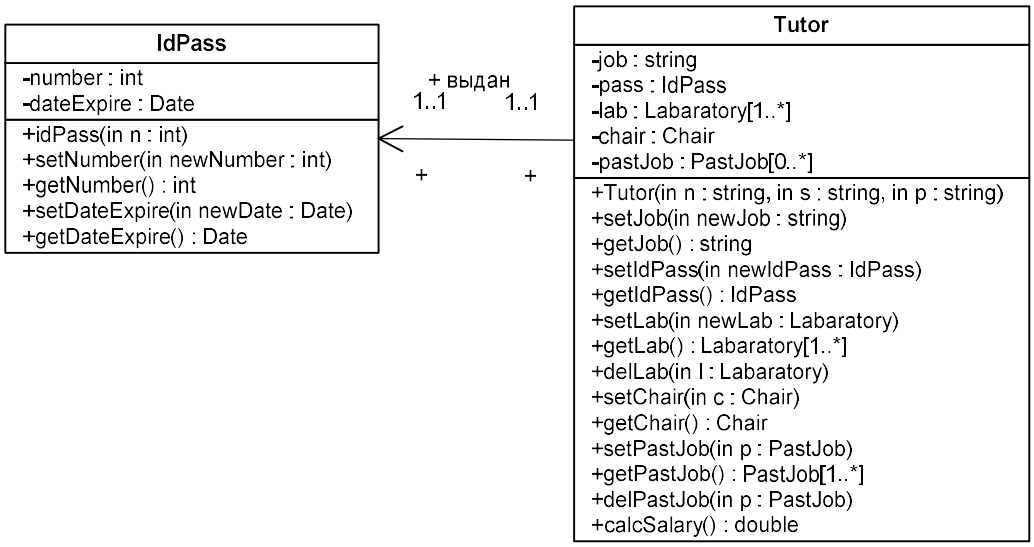

Рис. 6.4. Бинарная ассоциация

Классы примут вид:

```
public class Tutor extends Person{
    private String job;
    private IdPass pass;
    public Tutor(String n, String s, String p, double sal){
        name = n;
        lastname = s;
        position = p;
      salary = sal;
    }
    public void setJob(String newJob){
        position = newJob;
    }
    public String getJob(){
        return job;
    }
    public double calcSalary(){
        return salary;
    }
    public void setIdPass(IdPass c){
        pass = c;
    }
    public IdPass getIdPass(){
        return pass;
    }
}

public class IdPass{
    private Date dateExpire;
```

```
    private int number;
    public IdPass(int n){
        number = n;
    }
    public void setNumber(int newNumber){
        number = newNumber;
    }
    public int getNumber(){
        return number;
    }
    public void setDateExpire(Date newDateExpire){
        dateExpire = newDateExpire;
    }
    public Date getDateExpire(){
        return dateExpire;
    }
}
```

В теле программы создаем объекты и связываем их:

```
IdPass pass = new IdPass(5422);
pass.setDateExpire(new SimpleDateFormat("yyyy-MM-dd").parse("2020-
12-31"));
AssociateProfessor.setIdPass(pass);
System.out.println(AssociateProfessor.getName() +" работает в
должности "+ AssociateProfessor.getPosition());
System.out.println("Пропуск действует до " + new
SimpleDateFormat("yyyy-MM-
dd").format(AssociateProfessor.getIdCard().getDateExpire()) );
```

Здесь `AssociateProfessor` (доцент) – это объект класса Tutor.

2.2 *N-арная ассоциация*

Допустим, что в университете за каждым преподавателем закреплена лаборатория со специальным оборудованием. Добавляем новый класс `Laboratory`. Каждому объекту `Tutor` может соответствовать несколько подобных рабочих помещений, поэтому мощность связи – «один ко многим» (см. рисунок 6.5). Навигация от `Tutor` к `Laboratory`.

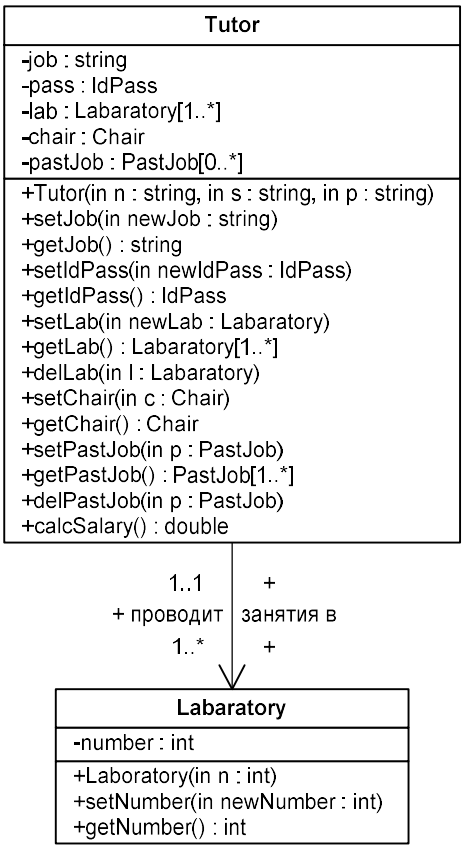

Рис. 6.5. *N*-арная ассоциация

Новый класс Labaratory:

```
public  class Labaratory{
     private int number;
     public Labaratory(int n){
         number = n;
     }
     public void setNumber(int newNumber){
         number = newNumber;
     }
     public int getNumber(){
         return number;
     }
}
```

Добавим в класс `Tutor` атрибут и метод для работы с `Laboratory`:

```
...
private Set lab = new HashSet();
...
public void setLab(Labaratory newLab){
```

```
    lab.add(newLab);
}
public Set getLab(){
    return lab;
}
public void delLab(Labaratory r){
    lab.remove(r);
}
...
```

Пример использования:

```
public static void main(String[] args){
    Tutor AssociateProfessor = new Tutor("Иван", "Иванов",
"Доцент", 25000);
    IdPass pass = new IdPass(5422);
    pass.setDateExpire(new SimpleDateFormat("yyyy-MM-
dd").parse("2020-12-31"));
    AssociateProfessor.setIdPass(pass);
    Labaratiry lab401 = new Labaratory(401);
    Labaratiry lab407 = new Labaratory(407);
    AssociateProfessor.setLab(lab401);
    AssociateProfessor.setLab(lab407);
    System.out.println(AssociateProfessor.getName() +" работает в
должности "+ AssociateProfessor.getPosition());
    System.out.println("Пропуск действует до " + new
SimpleDateFormat("yyyy-MM-
dd").format(AssociateProfessor.getIdCard().getDateExpire()) );
    System.out.println("Имеет право брать ключи от:");
    Iterator iter = AssociateProfessor.getLab().iterator();
    while(iter.hasNext()){
         System.out.println( ((Labaratiry)
iter.next()).getNumber());
    }
}
```

2.3 *Агрегация*

Агрегация обозначает отношение классов в иерархии «целое/часть» и обеспечивает перемещение от целого (агрегата) к его частям (атрибутам).

Добавим в модель класс `Chair` (кафедра) наш университет структурирован по кафедрам. На каждой кафедре может работать один или более человек. Можно сказать, что кафедра включает в себя одного или более сотрудников и таким образом их агрегирует (см. рисунок 6.6).

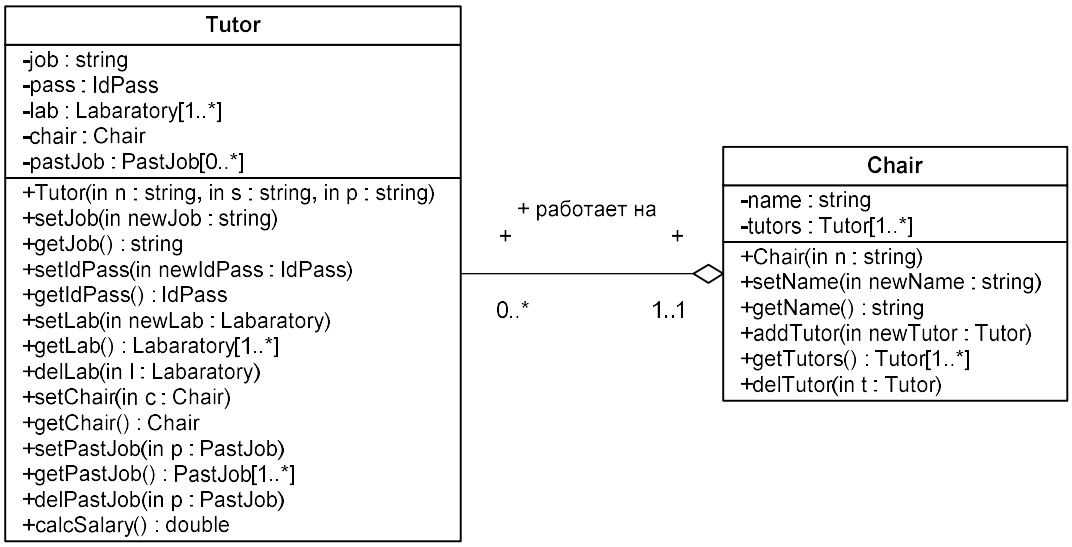

Рис. 6.6. Агрегация

Класс, помимо конструктора и метода изменения имени отдела, имеет методы для занесения на кафедру нового преподавателя, для удаления сотрудника и для получения всех сотрудников, преподающих на данной кафедре. Навигация на диаграмме не показана, значит она является двунаправленной: от объекта типа `Chair` можно узнать о преподавателе и от объекта типа `Tutor` можно узнать к какой кафедре он относится. Для этого в класс `Tutor` добавлены поле и методы для назначения и получения отдела.

Класс `Chair`:

```
public class Chair{
    private String name;
    private Set tutors = new HashSet();
    public Chair(String n){
        name = n;
    }
    public void setName(String newName){
        name = newName;
    }
    public String getName(){
        return name;
    }
    public void addTutor(Tutor newTutor){
        tutors.add(newTutor);
        // связываем преподавателя с этой кафедрой
        newTutor.setChair(this);
    }
```

```
    public Set getTutors(){
        return tutors;
    }
    public void delTutor(Tutor t){
        tutors.remove(t);
    }
}
```

Добавим в класс `Tutor` поле и методы для назначения и получения отдела.

```
...
private Chair chair;
...
public void setChair(Chair c){
    chair = c;
}
public Chair getChair(){
    return chair;
}
```

Использование:

```
Chair KIBEVS = new Chair("КИБЭВС");
KIBEVS.addTutor(AssociateProfessor);
System.out.println("Относится к кафедре
"+AssociateProfessor.getChair().getName());
```

2.4 ***Композиция***

Допустим, что преподаватель в процессе своей трудовой деятельности может переходить с одной кафедры на другую. Поэтому в программной системе нужно хранить данные о прежней занимаемой должности.

Для этого введем новый класс `PastJob` (см. рисунок 6.7). В него, помимо свойства `name` (название должности), добавим и свойство `chair`, которое свяжет его с классом `Chair`.

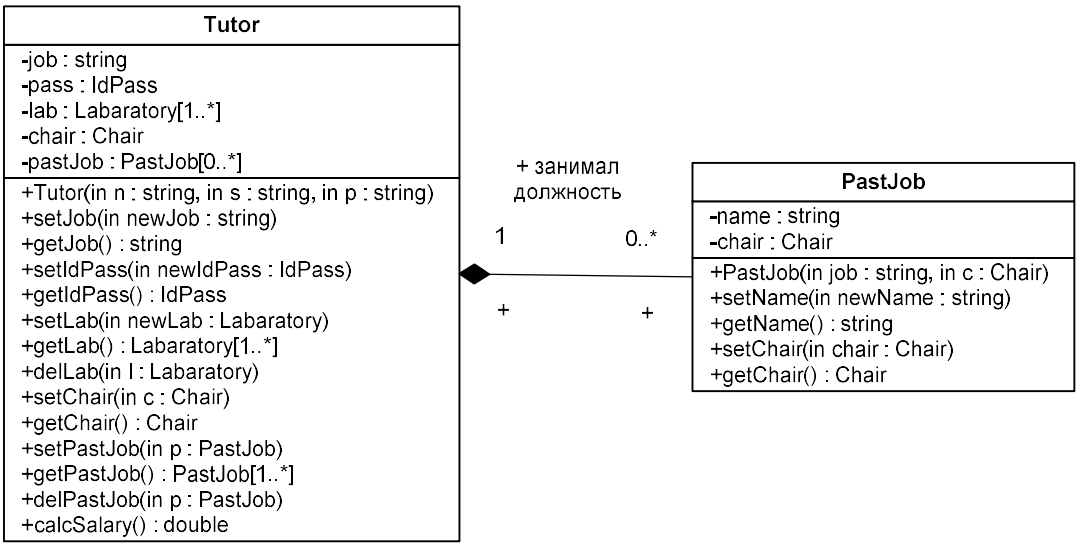

Рис. 6.7. Композиция

Данные о прошлых занимаемых должностях являются частью данных о преподавателе, таким образом, между ними возникает связь «целое-часть» и в то же время, данные о прошлых должностях не могут существовать без объекта типа `Tutor` - именно это условие отличает композицию от агрегации. Уничтожение объекта `Tutor` должно привести к уничтожению объектов `PastJob`.

Класс `PastJob`:

```
private class PastJob{
     private String name;
     private Chair chair;
     public PastJob(String job, Chair c){
         name = job;
         chair = c;
     }
     public void setName(String newName){
         name = newName;
     }
     public String getName(){
         return name;
     }
     public void setChair(Chair c){
         chair = c;
     }
     public Chair getChair(){
         return chair;
     }
}
```

В класс Tutor добавим свойства и методы для работы с данными о прошлой должности:

```
...
private Set pastJob = new HashSet();
...
public void setPastJob(PastJob p){
     pastJob.add(p);
}
public Set getPastJob(){
     return pastJob;
}
public void delPastJob(PastJob p){
     pastJob.remove(p);
}
...
```

Пример использования:

```
// изменяем должность
AssociateProfessor.setPosition("Инженер");
// смотрим ранее занимаемые должности:
System.out.println("В прошлом работал как:");
Iterator iter = AssociateProfessor.getPastJob().iterator();
while(iter.hasNext()){
    System.out.println( ((PastJob) iter.next()).getName());
}
```

3. ***Зависимость***

Зависимость – отношение, которое показывает, что изменение в одном классе (независимом) может влиять на другой класс (зависимый), который использует его.

Для организации диалога с пользователем введем в систему класс Menu (см. рисунок 6.8). Встроим в класс метод showTutors, который показывает список сотрудников и их должности. Параметром для метода является массив объектов Tutors. Таким образом, изменения внесенные в класс Tutors могут потребовать и изменения класса Menu. Заметим, что класс Menu не относится к прикладной области, а представляет собой системный класс воображаемого приложения.

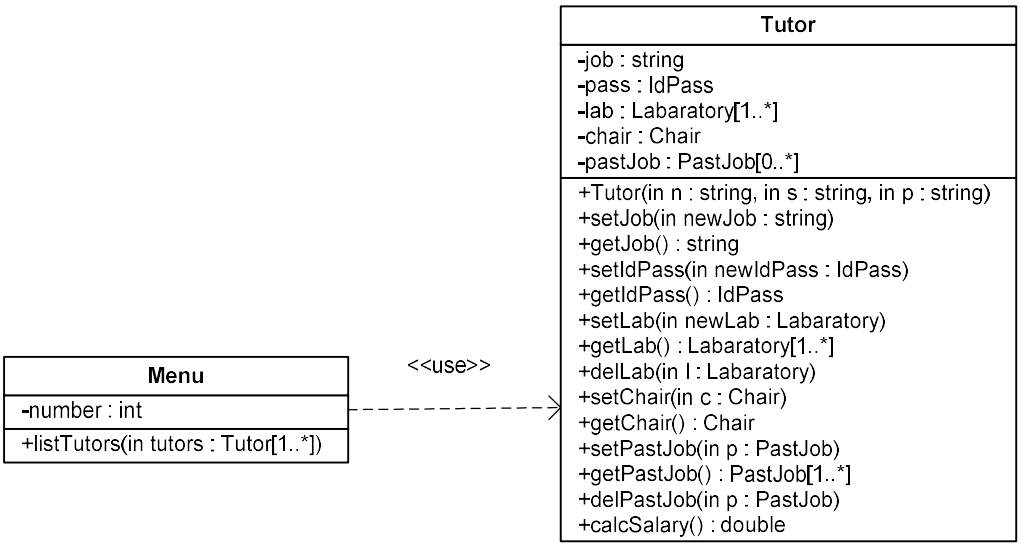

Рис. 6.8. Зависимость

Класс `Menu`:

```
public class Menu{
    private static int i=0;
    public static void listTutors(Tutor[] tutors){
        System.out.println("Список преподавателей:");
        for (i=0; i<tutors.length; i++){
            if(tutors[i] instanceof Tutor){
                System.out.println(tutors[i].getName() +" - " +
tutors[i].getJob());
            }
        }
    }
}
```

Пример использования:

```
// добавление нового преподавателя
Tutor Professor = new Employee("Петр", "Петров", "Профессор");
Menu menu = new Menu();
Tutor tutors[] = new Tutor[10];
employees[0]= AssociateProfessor;
employees[1] = Professor;
Menu.listTutors(tutors);
```

4. ***Реализация***

Реализация – отношение между классами, в котором класс-приемник выполняет реализацию операций интерфейса класса-источника.

Для демонстрации этого отношения создадим интерфейс `Unit` – другие структурные подразделения в университете (факультеты, филиалы университета в других городах и т.д.). Интерфейс `Unit` (см. рисунок 6.9) пред-

ставляет собой самую абстрактную единицу деления. В каждой единице деления работает какое-то количество преподавателей, поэтому метод для получения количества работающих людей будет актуален для каждого класса реализующего интерфейс `Unit`.

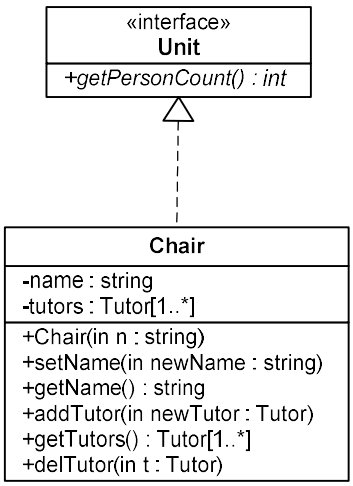

Рис. 6.9. Реализация

Интерфейс `Unit`:

```
public interface Unit{
     int getPersonCount();
}
```

Реализация в классе `Chair`:

```
public class Chair implements Unit{
    ...
    public int getPersonCount(){
        return getTutors().size();
    }
```

Пример использования:

```
System.out.println("На кафедре
"+AssociateProfessor.getChair().getName()+" работает "
+sAssociateProfessor.getChair().getPersonCount()+" человек.");
```

Задание

1. Ознакомиться с основными концепциями объектно-ориентированного анализа и проектирования.

2. Изучить основные приемы объектно-ориентированного программирования, разобраться в технологии их реализации для выбранного языка и среды программирования.

3. Выбрать предметную область в соответствии с вариантом, составить для неё модель проектирования и модель предметной области, используя UML.

4. Реализовать в программе операции бизнес-логики, соответствующие предметной области варианта задания, с применением наследования и полиморфизма:

4.1 Разработать структуру абстрактного класса, который объявляет собой минимально необходимый интерфейс.

4.2 Разработать производный класс, осуществив его наследование от разработанного абстрактного класса, с реализацией всех чисто виртуальных функций.

4.3 В каждом варианте помимо представленных методов, необходимо определить один дополнительный метод для класса самостоятельно.

5. Написать отчет и защитить у преподавателя.

Контрольные вопросы

1. Понятие «объектно-ориентированное программирование».
2. Понятия «объект» и «класс».
3. Отношения между классами: ассоциация, агрегация, композиция, использование, наследование, инстанцирование.
4. Абстракция.
5. Инкапсуляция.
6. Модульность.
7. Иерархия.
8. Типизация.
9. Сохраняемость.
10. Конструктор и деструктор.
11. Методы класса.
12. Понятия «абстрактный класс», «класс-интерфейс», «класс-утилита».
13. Области видимости, статические члены класса.
14. Модель предметной области.
15. Модель проектирования.
16. Диаграммы классов. Основные элементы и обозначения.
17. Инкапсуляция.
18. Наследование. Одиночное и множественное наследование.
19. Проблемы, возникающие при множественном наследовании, и способы их решения.
20. Полиморфизм.
21. Виртуальная функция.
22. Таблица виртуальных функций.
23. Сериализация и десериализация объектов.
24. Методы сериализации объектов.

Варианты заданий

1. Тема выполняемой курсовой работы или проекта ГПО.

2. Комплексные числа. Объект класса хранит действительную и мнимую часть комплексного числа. Предусмотреть методы вычисления модуля и аргумента, сложения, вычитания, деления и умножения комплексных чисел.

3. Полином. Класс хранит вещественные коэффициенты полинома и его степень. Предусмотреть метод вычисления для заданного аргумента, метод сокращения коэффициентов на заданное число.

4. Натуральная дробь. Предусмотреть метод сокращения дроби, умножения и деления на другую дробь (передавать как параметры функции).

5. Десятичная дробь. Методы класса - сложение, вычитание, умножение, деление десятичных дробей.

6. Время. Поля: час (0-23), минута (0-59), секунда (0-59). В классе описать конструктор, а также функции-члены установки времени, получения часа, минуты и секунды, а также две функции вывода на экран: вывода по шаблону: "16 часов 18 минут 3 секунды" и "4 p.m. 18 минут 3 секунды". Функции-члены установки полей класса должны проверять корректность задаваемых параметров.

7. Квадратная матрица. Класс квадратной матрицы целых чисел. Вычисление определителя, сложение, вычитание, умножение, деление матриц, транспонирование.

8. Дата. Поля: день (1-31), месяц (1-12), год (целое число). В классе описать конструктор, а также функции-члены установки дня, месяца и года, получения дня, месяца и года, а также две функции u1074 вывода на экран: вывода по шаблону: "5 января 1997 года" и "05.01.1997". Функции-члены установки полей класса должны проверять корректность задаваемых параметров. Функция-член дает приращение на 1 день.

9. Ориентированный граф. Класс обеспечивает хранение информации о вершинах и дугах ориентированного графа. Реализовать вывод всех возможных ориентированнных путей в графе, самого короткого пути.

10. Неориентированный граф. Класс обеспечивает хранение информации о вершинах и дугах графа. Реализовать метод проверки связности графа.

11. Стек. Обеспечивает хранение структуры стек. Все стандартные методы для стека, задание размера стека.

12. Очередь. Обеспечивает хранение структуры очередь. Все стандартные методы для очереди, задание размера очереди.

13. Дек. Обеспечивает хранение структуры дек. Все стандартные методы для дека, задание размера дека.

14. Файл. Хранение имени файла, размера, даты создания, количества обращений. Создать массив объектов. Вывести: a) список файлов, упорядоченный в алфавитном порядке; b) список файлов, размер которых превос-

ходит заданный; c) список файлов, число обращений к которым превосходит заданное число.

15. Персона. Хранение: Фамилия, Имя, Отчество, Адрес, Пол, Образование, Год рождения. Создать массив объектов. Вывести: a) список граждан, возраст которых превышает заданный; b) список граждан с высшим образованием; c) список граждан мужского пола.

16. Квадратное уравнение. Методы: нахождение корней квадратного уравнения.

17. Кубическое уравнение. Методы: нахождение корней кубического уравнения.

18. Двунаправленный список. Методы: добавление в конец списка, добавление в этот же список в конец списка, удаление указанного элемента из списка, присвоение списков, сравнение списков, получение элемента списка, установка указателя на следующий/предыдущий элемент списка.

19. Отрезок. Поля - координаты концов. Методы ввода координат концов, поиск середины отрезка, поворота отрезка относительно своей середины на заданный угол.

20. Вектор в трехмерном евклидовом пространстве. Задается своими полярными координатами. Обязательны функции-члены класса: ввод вектора, вывод вектора, функция, которая возвращает рабочий вектор в декартовых координатах, вращение, растяжение, сжатие.

21. Прогрессия. Обязательные функции-члены класса: ввода/вывода коэффициентов прогрессии, вычисление суммы, вычисление *N*-того члена, выбор типа – арифметическая или геометрическая.

22. Угол между плоскостями. Обязательные функции-члены класса: ввода/вывода координат плоскости, определение величины угла в градусах, перевода величины угла в радианы, нахождения синуса угла, определение кратчайшего расстояния между точкой и плоскостью.

23. Хэш. Ассоциативный массив, который позволяет хранить пару *<Ключ, Значение>* и выполнять 3 операции: добавление новой пары, операцию поиска, операцию удаления по ключу.

24. Куча. Структура данных типа дерево, которая удовлетворяет свойству кучи: если *B* является узлом-потомком узла *A*, то к*люч(A) ≥ ключ(B)*. Операции: нахождение максимума или минимума (нахождение максимального элемента в max-куче или минимального элемента в min-куче); удаление максимума или удаление минимума (удаление корневого узла в max- или min-куче); увеличение ключа или уменьшения ключа (обновление ключа в max- или min-куче); добавление нового ключа в кучу; слияние (соединение двух куч с целью создания новой кучи, содержащей все элементы обеих исходных).

25. Шар. Обязательные функции-члены класса: ввод/вывод координаты центра на плоскости и радиуса, вычисление площади поверхности, вы-

числение объема, вычисление площади сечения шара плоскостью удаленой от центра шара на некоторое расстояние.

26. Октаэдр. Обязательные функции-члены класса: ввод/вывод координат плоскости, проверка правильности октаэдра, вычисление площади поверхности, вычисление объема.

27. Додекаэдр. Обязательные функции-члены класса: ввод/вывод координат плоскости, проверка правильности додекаэдра, вычисление площади поверхности, вычисление объема.

Тема № 8

Язык функционального программирования Haskell

Цель работы

Знакомство с основами функционального программирования на примере языка Haskell.

Краткие теоретические сведения

Программа на **функциональном языке программирования** состоит из множества определений функций и выражения, чья величина рассматривается как результат программы. Математической моделью функционального программирования является ***λ-исчисление***. Основными языками функционирования в настоящее время являются Lisp и Haskell.

Далее будет рассматриваться интерпретатор языка Haskell и его интерпретатор Hugs. После запуска интерпретатора Hugs на экране появляется диалоговое окно среды разработчика, автоматически загружается специальный файл предопределений типов и определений стандартных функций на языке Haskell (Prelude.hs), и выводится стандартное приглашение к работе. Это приглашение имеет вид

Prelude>

В общем случае перед символом > выводится имя последнего загруженного модуля. После вывода приглашения можно вводить выражения языка Haskell, либо команды интерпретатора. Команды интерпретатора отличаются от выражений языка Haskell тем, что начинаются с символа двоеточия (***:***). Примером команды интерпретатора является ко манда ***:quit***, по которой происходит завершение работы интерпретатора. Команды интерпретатора можно сокращать до одной буквы (***:q*** для команды ***:quit***). Команда ***:set*** используется для того, чтобы установить различные опции интерпретатора. Команда ***:?*** выводит список доступных команд интерпретатора.

Программы на языке Haskell представляют собой выражения, вычисление которых приводит к значениям. Каждое значение имеет тип.

Чтобы узнать тип выражения, можно использовать команду ***:type*** (или сокращенный аналог ***:t***).

Чтобы интерпретатор автоматически печатал тип каждого вычисленного результата можно выполнить команду ***:set +t***.

Язык Haskell является сильно типизированным языком программирования. Объявление типа в явном виде не обязательно (в этом случае за типами используемых переменных следит сам интерпретатор). В совокупности с развитой системой типов эта особенность делает программы на языке Haskell безопасными по типам. Гарантируется, что в правильной программе на языке Haskell все типы используются правильно. С практической точки

зрения это означает, что программа на языке Haskell не может вызвать ошибок доступа к памяти (Access violation). Также гарантируется, что в программе не может произойти использование неинициализированных переменных. Таким образом, многие ошибки в программе отслеживаются на этапе ее компиляции, а не выполнения.

Если программисту всё же потребовалось объявить тип, то следует помнить, что имена типов в языке Haskell всегда начинаются с заглавной буквы и используется конструкция вида

переменная :: Тип

Основными типами языка Haskell являются:

1) **Integer** и **Int –** для представления целых чисел. Целочисленные выражения в языке Haskell вычисляются с неограниченным числом разрядов. Тип **Int** представляет целые числа фиксированной длины.
2) **Float** и **Double –** для представления вещественных чисел.
3) **Bool –** для представления результата логических выражений. Может принимать значения True или False.
4) **Char –** для представления символов.
5) **String** - строковые значения, задаются в двойных кавычках. Являются списками символов.

Интерпретатор Hugs можно использовать для вычисления арифметических выражений. При этом можно использовать операторы **+, –, *, /** с обычными правилами приоритета. Кроме того, можно использовать оператор **^** (возведение в степень). Также можно использовать стандартные математические функции ***sqrt***, ***sin***, ***cos***, ***tg***, ***exp***, ***log*** и т.д. Аргумент в скобки помещать не обязательно, однако стоит помнить, что вызов функции – более приоритетная операция, чем обычная арифметическая. Сравните:

```
Prelude>sqrt 2
1.4142135623731 :: Double
Prelude>1 + sqrt 2
2.4142135623731 :: Double
Prelude>sqrt 2 + 1
2.4142135623731 :: Double
Prelude>sqrt (2 + 1)
1.73205080756888 :: Double
```

Кортеж – структура для хранения фиксированного количества разнородных данных. Представляют собой средство для хранения составных типов данных. Например, пара **((*'v'*, *54*), *3.14*)** принадлежит типу **((*Char*, *Integer*), *Double*)**. Аналогично можно задавать тройки и т.д., записывая их

аналогичным образом. Элементом кортежа может быть значение любого типа, в том числе и другой кортеж.

Для доступа к элементам кортежей, составленных из пар, используются стандартные функции ***fst*** и ***snd***, возвращающие, соответственно, первый и второй элементы пары:

```
Prelude>fst (5, True)
5 :: Integer
Prelude>snd (5, True)
True :: Bool
Prelude> fst (snd (1, ('A', 0.5)))
'A' :: Char
```

Список – вычислительная структура последовательностей, состоящих из элементов одного типа. Однако в качестве типа могут выступать кортежи и другие списки. Список можно задать с помощью ***[ ]***

```
Prelude>[1,2]
[1,2] :: [Integer]
Prelude>['1','2','3']
['1','2','3'] :: [Char]
Prelude>[(1,'a'),(2,'b')]
[(1,'a'),(2,'b')] :: [(Integer,Char)]
Prelude>[[1,2],[3,4,5]]
[[1,2],[3,4,5]] :: [[Integer]]
```

или оператора ***: -*** для добавления элемента в начало списка.

```
Prelude>1:[2,3]
[1,2,3] :: [Integer]
Prelude>'4':['1','2','3']
['4','1','2','3'] :: [Char]
```

Основные функции для работы со списками:

- ***head*** возвращает первый элемент списка.

```
Prelude>head [1,2,3]
1 :: Int

Prelude>head "hello"
'h' :: Char
```

- ***last*** возвращает последний элемент списка.

```
Prelude>last [1,2,3,4,5]
5 :: Int
```

– ***tail*** возвращает список без первого элемента.

```
Prelude>tail [1]
[] :: Integer
```

– ***init*** возвращает список без последнего элемента.

```
Prelude>init [1,2,3,4,5]
[1,2,3,4] :: [Integer]
```

– ***null*** проверяет список на пустоту.

```
Prelude>null []
True :: Bool
```

– ***length*** возвращает длину списка.

```
Prelude>length [1,2,3]
3 :: Int
```

– ***elem*** проверяет наличие элемента в списке.

```
Prelude> elem 2 [1,2,3]
True :: Bool
```

– ***take*** возвращает список, состоящий из n первых элементов исходного списка.

```
Prelude> take 2 [1,2,3]
[1,2] :: [Integer]
```

– ***zip*** возвращает список, состоящий из пар объединенных исходных списков.

```
Prelude> zip ["20","30"] [1,2,3]
[("20",1),("30",2)] :: [([Char],Integer)]
```

– ***!!*** возвращает элемент, номер которого задан (начиная с 0).

```
Prelude> [1,2,3,4,5] !! 3
4 :: Integer
```

В общем виде ***условное выражение*** записывается как:

if *условие* ***then*** *выражение* ***else*** *выражение*

При сравнении можно использовать следующие операторы:

- **<, >, <=,** >= (меньше, больше, меньше или равно, больше или равно).
- == – оператор проверки на равенство.
- /= – оператор проверки на неравенство.
- логические операторы ***&&*** и || (И и ИЛИ)
- функцию отрицания ***not***.

Для преобразования числовых значений в строки и наоборот существуют функции ***read*** и ***show***:

```
Prelude>show 1
"1" :: [Char]

Prelude>1 + read "12"
13 :: Integer
```

Пользовательские ***функции*** должны находиться в файле, который нужно загрузить в Hugs с помощью команды ***:load***. Для редактирования загруженной программы можно использовать команду ***:edit***. Команда ***:reload*** перечитывает последний загруженный файл с пользовательской функцией, если она была изменена из оболочки ОС напрямую.

Функции равноправны с такими значениями, как целые и вещественные числа, символы, строки, списки и т.д. Функции можно передавать в качестве аргументов в другие функции, возвращать их из функций и т.п.

Имена пользовательских функций и переменных должны начинаться с латинской буквы в нижнем регистре. Остальные символы в имени могут быть прописными или строчными латинскими буквами, цифрами или символами подчеркивания и апострофом.

Как и все значения в языке Haskell, функции имеют тип. Тип функции, принимающей значения типа *a* и возвращающей значения типа *b* обозначается как *a->b*.

Создадим файл *C:\\test.hs* и запишем в него следующий текст программы и сохраним:

```
square :: Integer -> Integer
square x = x * x
```

Первая строка объявляет, что мы определяем функцию square, принимающую один параметр типа Integer и возвращающую результат типа Integer (квадрат аргумента).

Загрузим программу

```
Prelude>:load "c:\\test.hs"
```

Если загрузка проведена успешно, приглашение интерпретатора изменится на *Main*>. Выполним команды:

```
Main>:type square
square :: Integer -> Integer
Main>square 5
25 :: Integer
```

Приведем ещё несколько примеров функций:

Функция с двумя переменными на входе для сложения двух целых значений:

```
add :: Integer -  > Integer -  > Integer
add x y = x + y
```

Функция инфиксной конкатенации двух списков:

```
(++) :: [a] -> [a] -> [a]
[] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

Функция с граничными условиями для определения знака числа:

```
sign :: Integer -> Integer
sign x | x > 0 =1
       | x == 0  = 0
       | x < 0 = -  1
```

Данную функцию можно переписать, используя условные выражения**:**

```
sign :: Integer -  > Integer
signum x = if x > 0 then 1
else if x < 0 then -  1
else 0
```

Задание

1. Изучить краткие теоретические сведения.

2. Получить вариант задания у преподавателя.

3. Реализовать программу на языке функционального программирования Haskell в соответствии с заданием.

4. Написать отчет и защитить у преподавателя.

Варианты заданий

1. Вычисление факториала.

2. Калькулятор обратной польской записи.

3. Функции для нахождения наибольшего общего делителя (НОД) и наименьшего общего кратного (НОК).

4. Вычисление списка делителей числа.

5. Проверка числа на простоту.

6. Составить уравнение прямой, проходящей через 2 заданные точки.

7. Быстрая сортировка списка, сортировка списка «пузырьком».

8. Решение квадратного уравнения.

9. Решение кубического уравнения.

10. Составить список произведений заданного диапазона чисел, отсортировав их в порядке убывания с удалением дубликатов.

11. Вычислить высоту заданного двоичного дерева и подсчитать количество листьев в дереве.

12. Высота равнобедренного треугольника H метров, основание L. Найти углы треугольника и длину боковой стороны.

13. Генерация бесконечного списка совершенных чисел. Совершенным называется такое число, которое равно сумме всех своих делителей.

14. Поиск такого пути коня на шахматной доске, что в каждой клетке конь побывает один и только один раз.

15. Генерация списка «счастливых» 2N-значных чисел (т. е. таких, сумма первых N чисел которых равна сумме вторых N чисел).

16. Генерация всех возможных расстановок знаков арифметических операций и скобок в заданном шестизначном числе так, чтобы результатом полученного выражения было число 100.

17. Поиск глобального экстремума полинома заданного порядка.

18. Отсортировать заданное бинарное дерево.

Контрольные вопросы

1. Понятие «функциональный язык программирования».
2. Особенности языков функционального программирования.
3. Лямбда-исчисление.
4. Бета-редукция.
5. Основные типы языка Haskell.
6. Функции для работы с кортежами.
7. Функции для работы со списками.

8. Допустимые имена переменных и функций.
9. Команды интерпретатора для работы с файлами программ.
10. Условные выражения в языке Haskell.
11. Определение функций в языке Haskell.

Тема № 9

Язык логического программирования Prolog

Цель работы

Знакомство с основами логического программирования на примере языка Prolog.

Краткие теоретические сведения

В языке Пролог сочетается использование нескольких важных концепций, к числу которых относятся:

1) применение фраз Хорна для представления знаний;

2) дескриптивный стиль программирования;

3) как декларативная, так и процедурная семантика;

4) возможность чередовать программный текст метауровня с текстом объектного уровня.

Логическая или, точнее, ***хорновская логическая программа*** состоит из набора хорновских фраз, которые в языке Пролог называются фактами и правилами. Структура этих правил и фактов такова, что, с одной стороны, с их помощью довольно естественно описываются многие задачи, а с другой стороны, они допускают простую процедурную интерпретацию. Два этих обстоятельства и позволяют использовать язык фактов и правил в качестве языка программирования.

Фактом называется формула вида

$$P(t1, ..., tn),$$

где P – предикатный символ, а $t1$, ..., tn – термы, построенные из переменных, констант и функциональных символов. С точки зрения математической логики, факты – это атомарные формулы.

Правилом называется формула вида

$$A1 \wedge ... \wedge An \Rightarrow A0 \text{ (равносильная формула } A0 \vee \neg A1 \vee ... \vee \neg An),$$

где все Ai – атомарные формулы (таким образом, факт – это частный случай правила). В обозначениях Пролога эту формулу принято записывать по-иному:

$$A0 :- A1, A2, ..., An.$$

Запросом к логической программе называется формула вида

$$A1 \wedge ... \wedge An,$$

где все Ai – атомарные формулы, вместе с приглашением Пролога на ввод такой запрос записывается в виде

$$?- A1, A2, ..., An.$$

Запрос, не содержащий переменных, читается так: *верно ли*, *что* $A1$ *и ... и* An*?* Если же в атомарных формулах запроса содержатся переменные

*X*1, ..., *Xm*, то его следует читать иначе: *для каких объектов X*1, ..., *Xm верно A*1 *и ... и An?*

Основные синтаксические объекты: атомы, константы и переменные. ***Константы*** состоят из атомов и чисел. ***Числа*** – целые и вещественные. ***Атомы*** – обозначения для других постоянных объектов предметной области. Атомы могут изображаться тремя различными способами:

1) последовательностью латинских букв, цифр, начинающейся со строчной буквы;

2) последовательностью, состоящей из специальных символов <---> | ==>| ... | &;

3) любыми последовательностями символов, заключенными в апострофы (в т.ч. и русскими буквами).

Атомы используются и для обозначения предикатных символов, некоторые предикатные символы являются встроенными. ***Переменные*** обозначаются любыми последовательностями латинских букв или цифр, начинающимися с большой буквы или символа подчеркивания.

Главным компонентом интерпретатора языка Пролог является универсальный механизм решения задач, принцип действия которого основан на правиле резолюции. Для того, чтобы воспользоваться этим механизмом, программист должен четко описать задачу при помощи фраз Хорна, выраженных на языке Пролог. В каждой фразе формулируется некоторое отношение между термами. ***Терм*** – это обозначение, представляющее некоторую сущность из исследуемого мира. Для того, чтобы привести в действие данный механизм решения задач, программист должен написать запрос, согласно которому будет необходимо выяснить, является ли конкретная атомарная формула следствием текущего множества фраз, представленных в программе.

Далее рассмотрим пример решения задачи на языке Prolog.

Пусть имеется генеалогическое древо (рис. 8.1). Овалами показаны женщины, прямоугольниками – мужчины, отношение «муж – жена» – двумя линиями, соединяющими мужчину и женщину, отношение «предок-пото-мок» – одной линией.

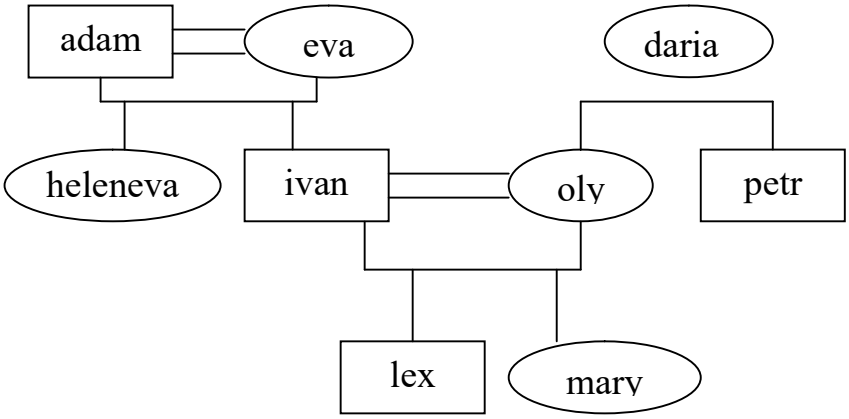

Рис. 8.1. Генеалогическое древо

Данное дерево было реализовано следующими правила(законами) на языке Пролог:

father(*adam*,*helen*).
father(*adam*,*ivan*).
father(*ivan*,*lex*).
father(*ivan*,*mary*).
mother(*eva*,*helen*).
mother(*eva*,*ivan*).
mother(*daria*,*petr*).
mother(*daria*,*oly*).
mother(*oly*,*lex*).
mother(*oly*,*mary*).
married(*adam*,*eva*).
married(*ivan*,*oly*).

Определим предикат «родитель» как отца или мать:

parent(*P*,*I*)*:-father*(*P*,*I*)*;mother*(*P*,*I*).

Определим предикат «брат-сестра» как двух людей, имеющих общего отца или общую мать, и не являющихся сами себе братом или сестрой:

bs(*S*,*B*)*:-*((*father*(*P*,*S*),*father*(*P*,*B*))*;*(*mother*(*P*,*S*),*mother*(*P*,*B*))),\==(*B*,*S*)

Проверим, является ли eva матерью для ivan? Получим от интерпретатора утвердительный ответ:

| *?- mother*(*eva*, *ivan*)

yes

Проверим, кто является родителями helen? Получим от интерпретатора вывод решения:

| *?- parent*(*helen*, *X*)

X=adam
X=eva

Задание

1. Изучить теоретический материал о языках логического программирования.
2. Изучить синтаксис основных команд языка Prolog.
3. Задать генеалогическое дерево вида: я, брат, сестра, отец, мать, бабушка (мать мамы), дедушка (отец мамы), бабушка (мать отца), дедушка (отец отца), жена брата, племянник (сын брата или сестры), племянница (дочь брата или сестры), дядя (со стороны отца), тетя (со стороны отца), дядя (со стороны матери), дядя (со стороны матери), дочь дяди со стороны матери, дочь дяди со стороны отца, прадед (отец деда со стороны матери), прабабушка (мать деда со стороны матери), прадед (отец бабушки со стороны матери), прабабушка (мать бабушки со стороны матери).
4. Нарисовать полученное дерево с отношениями.
5. Данное дерево задать только с использованием предикатов-фактов «мать», «отец».
6. Привести определение предикатов: родитель, потомок, предок, дядя (тетя), племянник, племянница, дед, родной(ая) брат(сестра), двоюродный(ая) брат(сестра), супруг и другие.
7. Написать отчет и защитить у преподавателя.

Контрольные вопросы

1. Понятие «логический язык программирования».
2. Особенности языков логического программирования.
3. Формальная логика.
4. Хорновская логическая программа.
5. Предикаты.
6. Интерпретация и унификация.
7. Принцип резолюции.
8. Представление знаний в виде фактов и правил.
9. Простые и составные логические запросы.
10. Атомы и числа, переменные, структуры в логическом программировании.

Вопросы к контрольным работам, зачету и экзамену

1. Язык программирования. Общие принципы построения и использования языков программирования.
2. Стандарты языков программирования.
3. Лямбда-исчисление. Аппликация, абстракция, редукция, преобразование.
4. Списки и функциональные выражения в функциональных языках программирования.
5. Механизмы и средства взаимодействия программы с операционной системой.
6. Функциональное программирование. Основные положения. Основные отличия от других типов языков программирования.
7. Классификация языков программирования. Близость языков программирования к естественному языку.
8. Унификация и хорновский клоз в логических языках программирования.
9. Модель вычислений функциональных языков программирования.
10. Языки программирования низкого уровня.
11. Средства разработки графического интерфейса пользователя. Эргономические свойства человеко-машинного интерфейса.
12. Процедурные языки программирования. Основные отличия от других типов языков.
13. Обоснование выбора языка программирования.
14. Перегрузка в языках программирования.
15. Логические языки программирования. Основные положения и понятия. Отличия от других типов языков программирования.
16. Объектно-ориентированные языки программирования. Основные отличия от других концепций языков программирования.
17. Полиморфизм в объектно-ориентированном программировании. Виртуальные функции. Таблицы виртуальных функций.
18. Структура языка программирования. Синтаксис и семантика языков программирования. Расширенная форма Бэкуса-Наура.
19. Наследование в объектно-ориентированном программировании. Множественное наследование. Проблемы множественного наследования.
20. Инкапсуляция в объектно-ориентированном программировании. Контроль доступа.
21. Понятие класса и объекта в объектно-ориентированном программировании. Атрибуты, методы, конструктор и деструктор, статические члены класса.
22. Диаграммы классов UML. Основные элементы и обозначения.
23. Данные. Средства описания данных. Типизация языка.

24. Преобразование типов. Контроль соответствия типов данных.

25. Объектно-ориентированное программирование. Основные концепции объектно-ориентированного программирования.

26. Современные интегрированные среды разработки программ. Основные компоненты среды программирования.

27. Декомпозиция программ.

28. Трансляторы. Интерпретация и компиляция.

29. Макропроцессоры и макрогенераторы.

30. Потоки и процессы. Сходства и различия.

31. Мониторы и защищаемые переменные в параллельном программировании.

32. Семафоры в параллельном программировании. Типы семафоров.

33. Отладчики. Генераторы кода и приложений.

34. Параллельная обработка данных и параллелизм. Параллельное и распределенное программирование.

35. Основные проблемы параллельного и распределенного программирования.

36. Оценка максимально возможного параллелизма.

37. Основные модели параллельного программирования.

38. Оптимизатор. Основные функции оптимизатора.

39. Обработка исключительных ситуаций. Иерархия исключительных ситуаций. Виды исключительных ситуаций.

40. Операторы обработки исключительных ситуаций в различных языках программирования.

41. Схема обработки исключительных ситуаций Б. Мейера.

42. Элементарные типы данных.

43. Перегрузка данных, операторов, методов.

44. Составные типы данных.

45. Механизмы логического вывода. Прямая и обратная цепочки рассуждений.

46. Пространство имен, область видимости, время жизни переменных.

47. Ошибки при работе с вещественными числами. Смешанная арифметика.

48. Операторы выбора и условные операторы.

49. Вещественные числа. Способы представления. Операции над вещественными числами.

50. Оператор присваивания. Операторы цикла.

51. Распределение памяти при выполнении программы.

52. Куча. Менеджер кучи. Фрагментация динамической памяти.

53. Концепция виртуальной памяти. Страничная организация памяти.

54. Сегментный принцип организации памяти. Сегментация памяти.

55. Указатели. Операции над указателями. Типизированные и нетипизированные указатели.

56. Динамические структуры данных. Реализация динамических структур данных с помощью указателей.

57. Библиотеки программ и классов. Статические и динамические библиотеки. Критерии проектирования библиотек.

58. Подпрограммы. Формальные и фактические параметры подпрограмм.

59. Передача параметров подпрограмме.

60. Программный стек и его изменение.

61. Рекурсивный и итерационный методы решения задач. Виды рекурсий.

62. Общая характеристика языков ассемблера: назначение, принципы построения и использования; структура языка.

63. Сериализация и десериализация. Методы сериализации объектов в базу данных.

64. Динамическая диспетчеризация.

65. Родовые (настраиваемые) сегменты.

66. Шаблоны.

67. Вариантные записи.

68. Средства описания действий над данными. Операторы, выражения, модули, блоки в языках программирования.

69. Ленивые и жадные вычисления в процедурном и функциональном программировании.

70. Языки высокого уровня.

71. Показатели качества программных средств.

72. Отношения между классами в объектно-ориентированном программировании.

73. Файлы, обработка файлов. Типы доступа к файлам.

74. Абстрактные типы данных: инкапсуляция, спецификация, реализация, параметризация.

75. Основные группы команд, операторы, средства взаимодействия с операционной системой в языках ассемблера.

Темы индивидуальных заданий для самостоятельной работы

Дисциплина «Языки программирования» в качестве самостоятельной и индивидуальной работы предусматривает выполнение реферата.

Реферат выполняется студентами по конкретной теме, которую следует изучить самостоятельно на основе литературных данных и материалов сети Интернет.

Работа над рефератом позволяет приобрести определенные навыки в обобщении и изложении материала по интересующим студента вопросам, а также навыки оформления материала.

Реферат выполняется на листах формата А4 и должен включать титульный лист, оглавление, введение, основную часть, заключение, список использованных источников и приложения. Примерный объем реферата – 10-15 страниц машинописного текста. Реферат должен быть выполнен в соответствии с общими требованиями и правилами оформления, принятыми в университете.

При оценке реферата учитывается полнота раскрытия темы, актуальность представленного материала, соответствие общим требованиям и правилам оформления.

1. Основные концепции, история, сайты, компиляторы и расширения языка «Фортран».
2. Основные концепции, история, сайты, компиляторы и расширения языка «Пролог».
3. Основные концепции, история, сайты, компиляторы и расширения языка «Java».
4. Основные концепции, история, сайты, компиляторы и расширения языка «ASP».
5. Основные концепции, история, сайты, компиляторы и расширения языка «PHP».
6. Основные концепции, история, сайты, компиляторы и расширения языка «Perl».
7. Основные концепции, история, сайты, компиляторы и расширения языка «Ruby».
8. Основные концепции, история, сайты, компиляторы и расширения языка «Python».
9. Построение компиляторов. Основные концепции.
10. Компиляторы для C++. Сравнительный анализ.
11. Языки программирования для сети Интернет. Сравнительный анализ.
12. Объектно-ориентированное программирование. Основные положения, правила создания классов и объектов.
13. Защита программ от несанкционированного копирования.

14. Защита программ от анализа.
15. Взлом программ с защитами. Дизассемблеры.
16. Оценка качества программ. Методы, стандарты и основные положения.
17. Методы оценки стоимости программных продуктов.
18. Оценка сложности вычисления операций.
19. Методы оценки трудоемкости создания программ.
20. Нормативы написания исходного кода.
21. Документирование программ. Основные стандарты.
22. Распространение программ. Shareware, Freeware.
23. Вычисление контрольных сумм файлов программ. Алгоритмы и реализации.
24. Шуточные и эзотерические языки программирования.
25. Методы обфускации исходного кода.
26. Программирование для операционной системы MacOS.
27. Программирование для мобильной платформы Android.
28. Облачные вычисления.
29. Стандарт HTML5.
30. Программирование искусственного интеллекта.
31. Унифицированный язык моделирования UML.
32. Модель компонентных объектов COM.
33. Интеллектуальный анализ данных (Data Mining).
34. Сравнительный анализ языков функционального программирования.
35. Обработка текстов на естественных языках.
36. Основные концепции, история, сайты, компиляторы и расширения языка «Scala»
37. MPI (Message Passing Interface)
38. Автоматическое построение онтологий для извлечения знаний из текста.
39. Библиотека Qt.
40. Параллельное программирование для графических ускорителей. Технология CUDA.
41. Кроссплатформенное программирование. Сравнительный анализ языков программирования.
42. Сравнительный анализ языков логического программирования.
43. Искусственные нейронные сети. Современное состояние вопроса.
44. Динамические библиотеки.
45. Технология ActiveX.
46. Методологии тестирования программного обеспечения.
47. Автоматизированное тестирование программного обеспечения.

48. Основные концепции, история, сайты, компиляторы и расширения языка «R».

49. Основные подходы и принципы разработки дизайна программного обеспечения

50. Особенности, основные подходы и принципы разработка видеоигр.

51. Системы контроля версий программ. Сравнительный анализ.

52. Continuous Integration, непрерывная интеграция и автоматизация сборок проекта.

53. Методы монетизации программного обеспечения.

54. Программирование для мобильной платформы iOS.

55. Программирование для мобильной платформы Windows Phone.

56. Основные концепции, история, сайты, компиляторы и расширения языка «Wolfram language».

57. Аудит информационной безопасности веб-сайта.

58. Паттерны объектно-ориентированного проектирования.

59. Программирование микроконтроллеров.

60. Управление разработкой программного обеспечения. Стандарты, методы, особенности.

Литература

1. Мещеряков Р.В., Давыдова Е.М. Языки программирования: Учебник / Р. В. Мещеряков, Е. М. Давыдова. – 2-е изд., перераб. и доп. – Томск.: В-Спектр. 2007. – 290с.
2. Кауфман В.Ш. Языки программирования. Концепции и принципы. - М.: ДМК Пресс, 2010. – 464 с.
3. Орлов С. А. Теория и практика языков программирования: Учебник для вузов. Стандарт 3-го поколения. – СПб.: Питер, 2013. – 688 с.
4. Автоматическая обработка текстов на естественном языке и компьютерная лингвистика : учеб. пособие / Большакова Е.И., Клышинский Э.С., Ландэ Д.В., Носков А.А., Пескова О.В., Ягунова Е.В. – М.: МИЭМ, 2011. – 272 с.
5. Анашкина Н.В. Технологии и методы программирования: учеб. пособие для студ. учреждений высш. проф. образования / Н.В. Анашкина, Н.Н. Петухова, В.Ю. Смольянинов. – М.: Издательский центр «Академия», 2012. – 384 с.
6. Бен-Ари М. Языки программирования. Практический сравнительный анализ: Пер. с англ. – М. : Мир, 2000. – 366 с.
7. Давыдов В.Г. Программирование и основы алгоритмизации : Учебное пособие для вузов / В. Г. Давыдов. – 2-е изд., стереотип. – М. : Высшая школа, 2005. – 448 с.
8. Гради Буч. Объектно-ориентированный анализ и проектирование с примерами приложений на C++ / перевод с англ. И. Романовского и Ф. Андреева. – М: "Бином", СПб.: "Невский диалект". – 558с.
9. Вольфенгаген В.Э. Конструкции языков программирования: Приемы описания : монография / Вячеслав Эрнстович Вольфенгаген ; Институт актуального образования "ЮрИнфоР-МГУ". Кафедра перспективных компьютерных исследований и информационных технологий. – М. : Центр ЮрИнфоР, 2001. – 278 с.
10. Камаев В.А. Технологии программирования : Учебник для вузов / В.А. Камаев, В.В. Костерин. – М. : Высшая школа, 2005. – 360 с.
11. Кручинин В.В. Технологии программирования : Учебное пособие / В.В. Кручинин. – Томск : ТУСУР, 2006. – 271 с.
12. Ходашинский И.А. Язык ПРОЛОГ в примерах и задачах : учебное пособие для вузов / И.А. Ходашинский. – Томск : ТУСУР, 2006. – 279 с.
13. Зюзьков В.М. Ленивое функциональное программирование : учебное пособие / В.М. Зюзьков. – Томск : Издательство Томского университета, 2007. – 293 с.

Учебное издание

А.С. Романов

Языки программирования

Методические указания по лабораторным работам, практическим занятиям и самостоятельной и индивидуальной работе
для студентов специальностей
10.03.01 – «Информационная безопасность», 10.05.02 – «Информационная безопасность телекоммуникационных систем», 10.05.04 – «Информационно-аналитические системы безопасности», 10.05.03 – «Информационная безопасность автоматизированных систем», 38.05.01 – «Экономическая безопасность»