МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное

образовательное учреждение высшего образования

«Тульский государственный университет»

Институт прикладной математики и компьютерных наук

Кафедра «Вычислительная техника»

|  |
| --- |
| Утверждено на заседании кафедры  «Вычислительная техника»  «29» мая 2020г., протокол № 11 |
| Заведующий кафедрой  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_А.Н. Ивутин |

**МЕТОДИЧЕСКИЕ УКАЗАНИЯ**

**по выполнению лабораторных работ**

**по дисциплине (модулю)**

**«Функциональное и логическое программирование»**

**основной профессиональной образовательной программы**

**высшего** **образования – программы бакалавриата**

по направлению подготовки

09.03.01 «Информатика и вычислительная техника»

с профилем

**«Программное обеспечение средств вычислительной техники и автоматизированных систем»**

Формы обучения: очная

Идентификационный номер образовательной программы: 090301-01-20

Тула 2020 год

**Разработчик(и) методических указаний**

**Разработчик(и):**

\_\_\_\_Волошко А.Г., доцент, к.т.н.\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(ФИО, должность, ученая степень, ученое звание) (подпись)

Содержание

[**Введение** 5](#_Toc18351190)

[**1.** **Знакомство с Visual Prolog 8** 6](#_Toc18351191)

[**1.1** **Цель и задачи работы** 6](#_Toc18351192)

[**1.2** **Описание Visual Prolog** 6](#_Toc18351193)

[**1.3** **Создание проекта в Visual Prolog** 6](#_Toc18351194)

[**1.4** **Компиляция и запуск программы** 8](#_Toc18351195)

[**1.5** **Регистрация программы** 9](#_Toc18351196)

[**1.6** **Объекты исследования, оборудование, материалы и наглядные пособия** 9](#_Toc18351197)

[**1.7** **Задание на работу (рабочее задание)** 10](#_Toc18351198)

[**1.8** **Ход работы (порядок выполнения работы)** 10](#_Toc18351199)

[**1.9** **Содержание отчета** 10](#_Toc18351200)

[**2.** **Создание приложений** 11](#_Toc18351201)

[**2.1** **Цель и задачи работы** 11](#_Toc18351202)

[**2.2** **Теоретические положения** 11](#_Toc18351203)

[**2.3** **Создание графического проекта** 14](#_Toc18351204)

[**2.4** **Создание консольного проекта** 17](#_Toc18351205)

[**2.5** **Функциональный файл проекта** 17](#_Toc18351206)

[**2.6** **Класс Math** 18](#_Toc18351207)

[**2.6.1** **Алгебраические функции** 18](#_Toc18351208)

[**2.6.2** **Тригонометрические функции** 20](#_Toc18351209)

[**2.6.3** **Базовые функции** 22](#_Toc18351210)

[**2.7** **Создание модулей** 23](#_Toc18351211)

[**2.8** **Теоретические исследования, оборудование, материалы и наглядные пособия** 26](#_Toc18351212)

[**2.9** **Задание на работу (рабочее задание)** 26](#_Toc18351213)

[**2.10** **Ход работы (порядок выполнения работы)** 27](#_Toc18351214)

[**2.11** **Содержание отчета** 28](#_Toc18351215)

[**3.** **Рекурсивные программы** 29](#_Toc18351216)

[**3.1** **Цель и задачи работы** 29](#_Toc18351217)

[**3.2** **Теоретические положения** 29](#_Toc18351218)

[**3.3** **Нисходящая рекурсия** 30](#_Toc18351219)

[**3.4** **Хвостовая рекурсия** 31](#_Toc18351220)

[**3.5** **Теоретические исследования, оборудование, материалы и наглядные пособия** 32](#_Toc18351221)

[**3.6** **Задание на работу (рабочее задание)** 33](#_Toc18351222)

[**3.7** **Ход работы (порядок выполнения работы)** 34](#_Toc18351223)

[**3.8** **Содержание отчета** 35](#_Toc18351224)

[**4.** **Списки** 36](#_Toc18351225)

[**4.1** **Цель и задачи работы** 36](#_Toc18351226)

[**4.2** **Объявление списков** 36](#_Toc18351227)

[**4.3** **Головы и хвосты** 37](#_Toc18351228)

[**4.4** **Представления списков** 37](#_Toc18351229)

[**4.5** **Использование списков** 38](#_Toc18351230)

[**4.6** **Вывод списка на печать** 38](#_Toc18351231)

[**4.7** **Подсчет элементов в списке** 39](#_Toc18351232)

[**4.8** **Хвостовая рекурсия** 40](#_Toc18351233)

[**4.9** **Модификация списка** 42](#_Toc18351234)

[**4.10** **Теоретические исследования, оборудование, материалы и наглядные пособия** 43](#_Toc18351235)

[**4.11** **Задание на работу (рабочее задание)** 43](#_Toc18351236)

[**4.12** **Ход работы (порядок выполнения работы)** 46](#_Toc18351237)

[**4.13** **Содержание отчета** 46](#_Toc18351238)

[**5.** **. Множества** 47](#_Toc18351239)

[**5.1** **Цель и задачи работы** 47](#_Toc18351240)

[**5.2** **Теоретические положения** 47](#_Toc18351241)

[**5.3** **Множества чисел** 47](#_Toc18351242)

[**5.4** **Теоретические исследования, оборудование, материалы и наглядные пособия** 49](#_Toc18351243)

[**5.5** **Задание на работу (рабочее задание)** 49](#_Toc18351244)

[**5.6** **Ход работы (порядок выполнения работы)** 53](#_Toc18351245)

[**5.7** **Содержание отчета** 53](#_Toc18351246)

[**6.** **Строки** 54](#_Toc18351247)

[**6.1** **Цель и задачи работы** 54](#_Toc18351248)

[**6.2** **Теоретические положения** 54](#_Toc18351249)

[**6.3** **Теоретические исследования, оборудование, материалы и наглядные пособия** 58](#_Toc18351250)

[**6.4** **Задание на работу (рабочее задание)** 58](#_Toc18351251)

[**6.5** **Ход работы (порядок выполнения работы)** 60](#_Toc18351252)

[**6.6** **Содержание отчета** 60](#_Toc18351253)

[**7.** **Деревья** 61](#_Toc18351254)

[**7.1** **Цель и задачи работы** 61](#_Toc18351255)

[**7.2** **Теоретические положения** 61](#_Toc18351256)

[**7.3** **Дерево поиска** 61](#_Toc18351257)

[**7.4** **Поиск в ширину** 62](#_Toc18351258)

[**7.5** **Поиск в глубину** 66](#_Toc18351259)

[**7.6** **Эвричтический поиск** 66](#_Toc18351260)

[**7.7** **Теоретические исследования, оборудование, материалы и наглядные пособия** 70](#_Toc18351261)

[**7.8** **Задание на работу (рабочее задание)** 70](#_Toc18351262)

[**7.9** **Ход работы (порядок выполнения работы)** 72](#_Toc18351263)

[**7.10** **Содержание отчета** 72](#_Toc18351264)

[**8.** **Файлы** 73](#_Toc18351265)

[**8.1** **Цель и задачи работы** 73](#_Toc18351266)

[**8.2** **Теоретические положения** 73](#_Toc18351267)

[**8.3** **Теоретические исследования, оборудование, материалы и наглядные пособия** 74](#_Toc18351268)

[**8.4** **Задание на работу (рабочее задание)** 74](#_Toc18351269)

[**8.5** **Ход работы (порядок выполнения работы)** 75](#_Toc18351270)

[**8.6** **Содержание отчета** 76](#_Toc18351271)

[**9.** **Haskell: работа со списками** 77](#_Toc18351272)

[**9.1** **Цель и задачи работы** 77](#_Toc18351273)

[**9.2** **Теоретические положения** 77](#_Toc18351274)

[**9.3** **Теоретические исследования, оборудование, материалы и наглядные пособия** 87](#_Toc18351275)

[**9.4** **Задание на работу (рабочее задание)** 88](#_Toc18351276)

[**9.5** **Ход работы (порядок выполнения работы)** 90](#_Toc18351277)

[**9.6** **Содержание отчета** 90](#_Toc18351278)

[**10.** **Haskell** 91](#_Toc18351279)

[**10.1** **Цель и задачи работы** 91](#_Toc18351280)

[**10.2** **Теоретические положения** 91](#_Toc18351281)

[**10.3** **Теоретические исследования, оборудование, материалы и наглядные пособия** 94](#_Toc18351282)

[**10.4** **Задание на работу (рабочее задание)** 94](#_Toc18351283)

[**10.5** **Ход работы (порядок выполнения работы)** 96](#_Toc18351284)

[**10.6** **Содержание отчета** 96](#_Toc18351285)

[**Список используемой литературы** 97](#_Toc18351286)

# **Введение**

Цель работы – изучить язык логического программирования Visual Prolog и язык функционального программирования Haskell.

# **Знакомство с Visual Prolog 8**

## **Цель и задачи работы**

Узнать и усвоить базовые понятия языка и основы написания программ в среде Visual Prolog на языке Пролог

## **Описание Visual Prolog**

**Visual Prolog** – это объектно-ориентированное расширение языка программирования PDC Prolog, развивавшегося из Turbo Prolog (Borland), семейства Prolog, а также система визуального программирования датской фирмы Prolog Development Center.

Visual Prolog автоматизирует построение сложных процедур и освобождает программиста от выполнения тривиальных операций. С помощью Visual Prolog проектирование пользовательского интерфейса и связанных с ним окон, диалогов, меню, строки уведомлений о состояниях и т. д. производится в графической среде. С созданными объектами могут работать различные Кодовые Эксперты (Code Experts), которые используются для генерации базового и расширенного кодов на языке Prolog, необходимых для обеспечения их функционирования. [1]

## **Создание проекта в Visual Prolog**

Среда, которая используется при разработки программ, называется IDE, что является сокращением от Integrated Development Environment.

Система окон и диалогов, которые создаются для общения с потенциальными пользователями программы, называется Graphical User Interface, или сокращенно – GUI.

Для создания нового проекта, нужно выбрать команду «**Project/New»** в меню задач, либо нажать комбинацию клавиш «Ctrl+Shift+N»,рис. 1.1.

В открывшемся диалоговом окне, рис. 1.2, необходимо ввести название проекта в поле «**Project Name**», в раскрывающемся списке «**Project Kind**» нужно выбрать тип проекта: «**Console Application**», «**MDI**» или «**SDI**». Для создания консольного приложения следует выбрать «**Console Application**», а для интерфейсного приложения следует выбрать «**MDI**» В поле «**Base Directory**» необходимо указать путь для сохранения проекта. Остальные поля можно оставить без изменения.

Для создания проекта нужно нажать на кнопку «**Finish**».

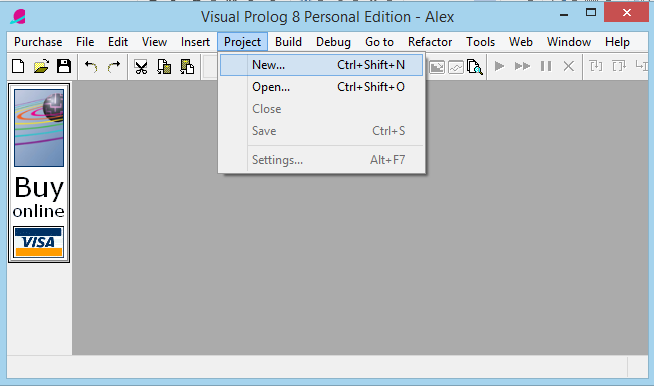


Рисунок 1.1– Создание нового проекта

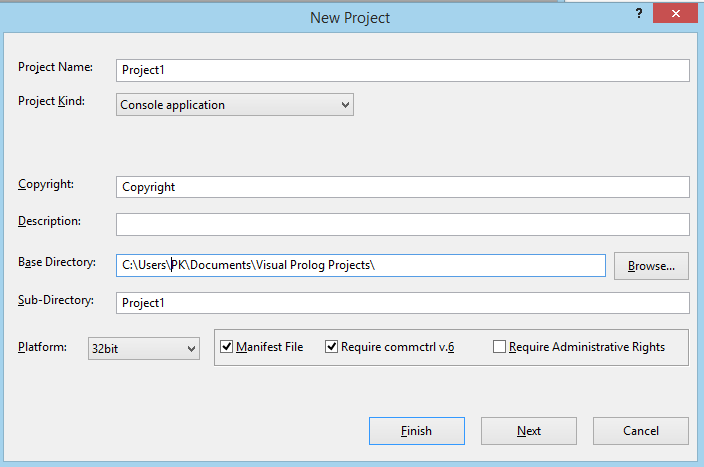


Рисунок 1.2– Настройка проекта

После того, как проект создан, построится дерево проекта, рис. 1.3.

В левой панели окна показано дерево проекта, которое содержит системные шаблоны и встроенные библиотеки, а в правой – информационный браузер.

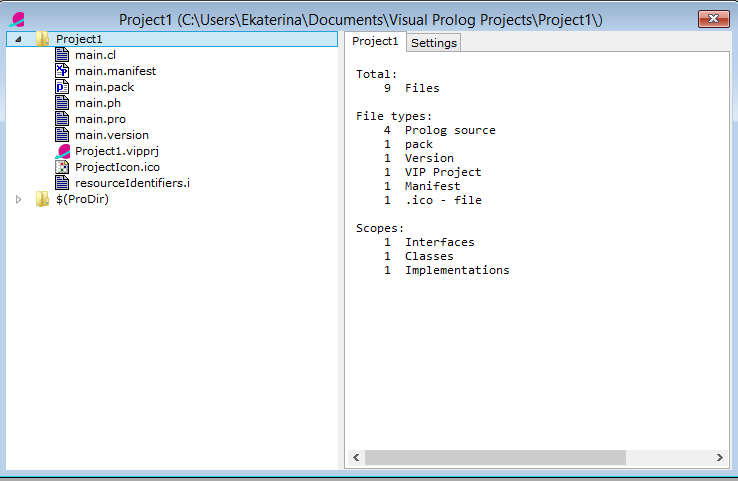


Рисунок 1.3– Дерево проекта

## **Компиляция и запуск программы**

Для того чтобы скомпилировать программу, нужно выбрать команду «**Build/Build**» в меню задач, либо нажать комбинацию клавиш «Ctrl+Shift+B», рис. 1.4. Начиная с 8 версии, Visual Prolog после создания проекта автоматически выполняет компиляцию проекта, поэтому этот шаг можно пропустить.

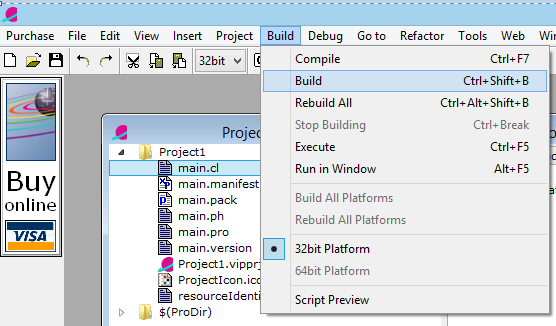


Рисунок 1.4– Компоновка проекта

Запуск программы выполняется командой «**Build/Execute** в меню задач, либо нажатием комбинацией клавиш «Ctrl+F5».

Для того, чтобы запустить программу в окне Windows необходимо нажать выполнить команду «**Build/Run in Window** в меню задач, либо нажатием комбинацией клавиш «Alt+F5».

## **Регистрация программы**

При запуске готовой программы появится окно диалога, в котором будет предложено зарегистрировать персональную версию программы или купить платную версию. Для регистрации Персональной версии продукта необходимо перейти по ссылке «Personal Edition», указав E-mail, на который придет лицензионный ключ. Лицензионный ключ необходимо ввести в поле «**License key**», рис. 1.5.

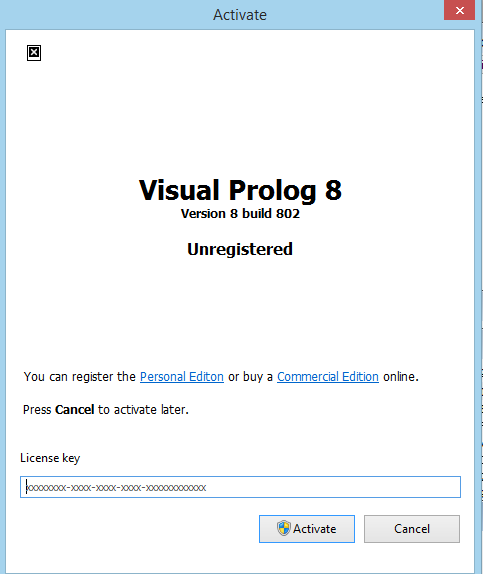


Рисунок 1.5– Регистрация Visual Prolog

## **Объекты исследования, оборудование, материалы и наглядные пособия**

Объектом исследования данной лабораторной работы являются основные компоненты языка Пролог.

Для выполнения работы необходимы ПК и соответствующее программное обеспечение:

* MS WINDOWS;
* VisualProlog;
* MS Office (для оформления отчета).

## **Задание на работу (рабочее задание)**

Создайте предикат, проверяющий, являются ли два человека

* сестрами;
* братьями;
* бабушкой и внуком (внучкой);
* дядей и племянником (племянницей);
* двоюродными братьями;
* родственниками.

## **Ход работы (порядок выполнения работы)**

Для выполнения лабораторной работы необходимо:

1. Ознакомиться с теоретической справкой;
2. Выполнить анализ поставленной задачи, выявить входную и выходную информацию и определить ее формат;
3. Разработать алгоритм решения задачи;
4. Разработать программное обеспечение;
5. Представить ПО преподавателю и получить допуск к защите работы;
6. Оформить отчет по лабораторной работе;
7. Защитить работу преподавателю, ответив на вопросы по ее содержанию и выполнению.

## **Содержание отчета**

В отчете должны присутствовать следующие пункты:

1. задание;
2. описание предметной области в формате IDEF5;
3. текст программы (подпрограммы) расчета;
4. распечатка результатов работы программы.

# **Создание приложений**

## **Цель и задачи работы**

Изучить специфику среды Visual Prolog, отличия языка Пролог от других языков программирования, способы организации управления программой при программировании.

## **Теоретические положения**

Visual Prolog – это компилируемый язык, а не интерпретируемый. В Visual Prolog принята строгая типизация данных (для повышения скорости трансляции и выполнения программ).

В начале программы в Visual Prolog обычно располагаются разделы описаний объектов программы. Здесь отсутствует возможность рассматривать правила как данные, т.е. добавлять и удалять их во время работы. В процессе выполнения программы в нее можно добавлять и из нее можно удалять только факты. В Visual Prolog нельзя определять операции. В целом все это приводит к тому, что Visual Prolog отличается высокой скоростью компиляции и выполнения.

Обобщенная структура программы в Visual Prolog:

* **domains** – объявления доменов;

В данном разделе объявляются любые нестандартные домены, используемые для аргументов предикатов. Домены в Прологе являются аналогами типов в других языках. Основные стандартные домены Visual Prolog:

* **Char** – Символ, реализуемый как беззнаковый byte. Синтаксически это символ, заключенный в одиночные апострофы;
* **Byte** – Все платформы 8 бит (0— 55)
* **Short** – Короткое, знаковое, количественное. Все платформы 16 бит (-32 768-32 767)
* **Ushort** – Короткое, беззнаковое, количественное, Все платформы 16 бит (0—65 535)
* **Word** – Все платформы 16 бит (0-65 535)
* **Integer** – целое число (из промежутка -32768...32767);
* **Long** – Длинное, знаковое, количественное, Все платформы 32 бит (-2 147 483 648-2 147 483 647)
* **Ulong** – Длинное, беззнаковое, количественное, Все платформы 32 бит (0-4 294 967 295)
* **Unsigned** – Без знаковое, количественное имеет платформозависимый размер. Платформы 16 бит (0—65 535) Платформы 32 бит (0-4 294 967 295)
* **Dword** – Все платформы 32 бит (0-4 294 967 295)
* **Real** – действительное число (лежащее между ±1e-307...±1e308);
* **String** – последовательность символов, заключенная в двойные кавычки;
* **Symbol** – символическая константа (начинающаяся со строчной буквы последовательность любых символов, заключенная в кавычки).
* **File** – файл.
* **predicates** – объявления предикатов;

В данном разделе объявляются предикаты и домены (типы) аргументов этих предикатов. Имена предикатов должны начинаться с буквы или цифры, за которой следует последовательность букв, цифр и символов подчеркивания (до 250 знаков). В именах предикатов нельзя использовать символы пробел, минус, звездочка, слэш. Объявление предиката имеет следующую форму:

predicateName(argumentTypel OptionalNamel, argumentType2 OptionalName2, < ... >; argumentTypeN OptionalNameN), где

argument\_type1, ..., argument\_typeN — либо стандартные домены, либо домены, объявленные в разделе domains.

Объявление домена аргумента и описание типа аргумента — суть одна и та же. Имена аргументов OptionalNamel будут игнорироваться компилятором.

* **clauses** – предложения (правила и факты);

Здесь необходимо разместить факты и правила, с которыми будет работать Visual Prolog, пытаясь разрешить цель программы.

Программы на языке Пролог состоят из двух типов фраз: **фактов** и **правил**, называемых предложениями.

**Факты** — это отношения или свойства, о которых известно, что они имеют значение "истина".

Факты имеют общий вид: property(objectl, object2, ..., objectN) ИЛИ relation(objectl, object2, ..., objectN).

**Правила** — это связанные отношения, которые позволяют Прологу логически выводить одну порцию информации из другой. Правило принимает значение "истина", если доказано, что заданный набор условий является истинным.

Правила имеют общую форму заголовок:- тело, которые выглядят так:

relation(object,object,...,object):-relation(object,...,object), relation(obj ect,...,obj ect).

В Прологе все правила имеют 2 части: **заголовок** и **тело**, разделенные специальным знаком :-.

**Заголовок** — это факт, который был бы истинным, если бы были истинными несколько условий. Это называется выводом или зависимым отношением.

**Тело** — это ряд условий, которые должны быть истинными, чтобы Пролог мог доказать, что заголовок правила истинен.

Факты и правила Пролога получают информацию при вызове с аргументами, которые могут быть константами или связанными переменными. Они возвращают информацию в вызывающую процедуру путем связывания аргументов, которые являются несвязанными переменными.

Различия между этими понятиями несущественны, и, поэтому, часто используется обобщенный термин отношение.

* **goal –** подцели.

В данном разделе задается внутренняя цель программы, которая позволяет программе быть скомпилированной, запускаться и выполняться независимо от среды визуальной разработки (VDE).

## **Создание графического проекта**

Для создания графического проекта (GUI) необходимо создать проект с типом «**MDI**».

Графический проект позволяет добавить форму.

Форма — это окно, на котором можно разместить такие компоненты, как кнопки, запускающие действия, поля редактирования, которые можно использовать для ввода текста, и полотна, на которых можно рисовать.

Для добавления формы нужно выполнить команду **File/New in New Package** в меню задач, либо нажатием комбинацией клавиш «Ctrl+N», рис. 2.1

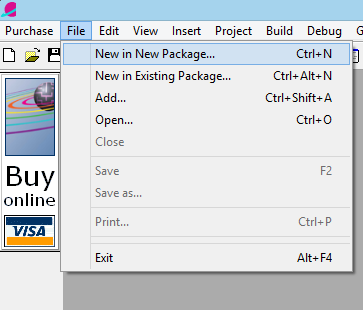


Рисунок 2.1– Добавление новой сущности к дереву проекта

В открывшемся диалоговом окне «Create Project Item» нужно выбрать пункт «Form» и в поле «name» указать название формы, и нажать на кнопку «create» рис. 2.2

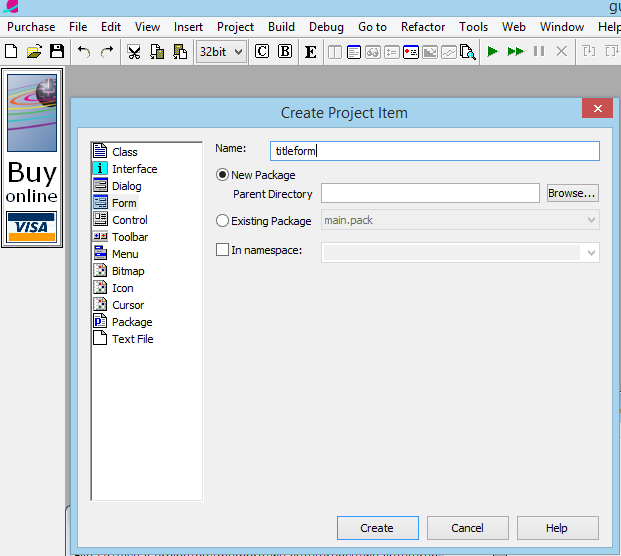


Рисунок 2.2– Создание формы

После создания формы, IDE покажет прототип новой формы, где можно изменить свойства объекта во вкладке «Properties», рис. 2.3

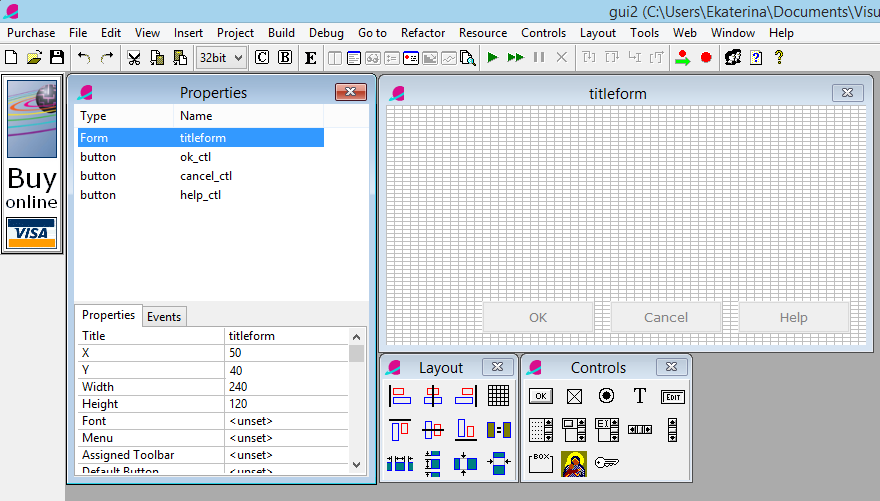


Рисунок 2.3– Свойства формы

На созданную форму можно добавлять элементы управления из панели «Controls», а используя панель «Layout» можно выравнивать добавленные элементы на форму.

Для запуска графического приложения необходимо выполнить команду **Build/Execute** в меню задач, либо воспользоваться комбинацией клавиш «Ctrl+F5».

В том случае, если при запуске приложения пункты меню не активны, то следует активировать их. Это можно сделать следующим образом. Для активации пунктов меню нужно в дереве проектов нажать правой кнопкой мыши по элементу **TaskWindow.win** и в контекстном меню выбрать «Attributes».

В открывшемся диалоговом окне помимо активного меню в приложении, можно настроить его основные параметры, рис. 2.4а.

Для того, чтобы добавить возможность в приложении открывать созданную форму при клике на команду «**File**/**New**» нужно в этом же окне нажать на кнопку «**Code Expert**», после чего откроется диалоговое окно. В открывшемся окне в пункте меню нужно нажать правой кнопкой мыши по элементу «**id\_file\_new** -> **onFileNew**», после чего в контекстном меню выбрать «**Go to Event Code**».

В открывшемся редакторе кода нужно заменить следующий фрагмент кода:

Листинг 2.1

*Clauses*

*onFileNew(\_Source, \_MenuTag).*

На следующий код:

Листинг 2.2

*Clauses*

onFileNew(W, \_MenuTag) :-

\_ = yyy::display(W).

Где «*yyy» –* название формы.

*Перед изменением кода необходимо скомпилировать программу.*

|  |  |
| --- | --- |
| а) | б) |

Рисунок 2.4– Свойства окна (а) и изменение фрагмента кода «onFileNew» (б)

## **Создание консольного проекта**

Для создания консольного проекта необходимо создать проект с типом «Console Application».

Основное отличие консольного приложения от графического состоит в том, что общение пользователя с программой осуществляется через консоль, т.е. при помощи команд.

Для запуска консольного приложения следует выполнить команду **Build/Run in Window** в меню задач, либо воспользоваться комбинацией клавиш «Alt+F5».

## **Функциональный файл проекта**

Для задания функциональности щелчком мыши нужно выбрать файл **main.pro** в дереве проектов, в шаблон которого нужно добавить свои операции, рис. 2.5.

В листинге шаблона программы отмечено место, куда нужно добавить код приложения. Фраза комментария на английском языке:

*% place your own code here*

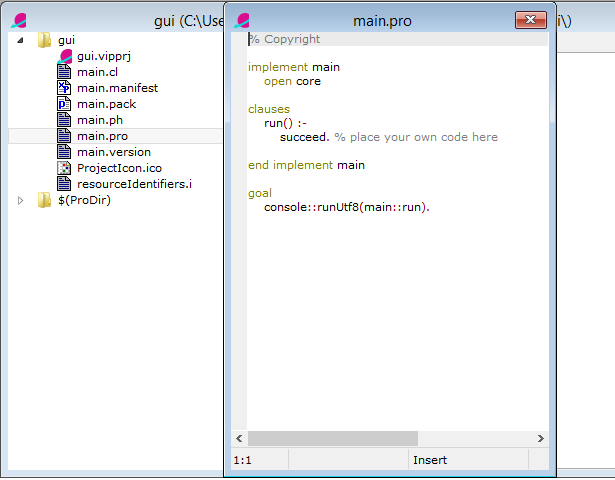


Рисунок 2.5 – Функциональный файл main.pro

## **Класс Math**

В Prolog, как и в других языках программирования, есть класс математических операций Math. В классе определены группы функций:

* алгебраические;
* тригонометрические;
* базовые функции.

*Для того, чтобы работали встроенные функции класса* ***Math*** *необходимо во фрагменте заголовка (строка open core, console) открыть этот класс дописав в строку math.*

### **Алгебраические функции**

В качестве примера будет рассмотрен пример алгебраических функций в консольном проекте:

Листинг 2.3

*implement main*

*open core, console*

*clauses*

*run() :-*

*init(),*

*write("X = "), ;X = read(),*

*write("Y = "), Y = read(), nl,*

*write("Исходные числа ", X, "и ", Y), nl, nl, \_= readLine(),*

*write("Операции с числами: ", X,"и ", Y), nl, \_= readLine(),*

*Z = X + Y,*

*write("Сумма: ", X, " + ", Y, " = ", Z), \_= readLine(),*

*Z1 = X - Y,*

*write("Разность: ", X, " - " Y, " = ", Z1), \_= readLine(),*

*Z2 = X \* Y,*

*write("Произведение: ", X, " \* ", Y, " = ", Z2), \_= readLine(),*

*Z3 = X / Y,*

*write("Частное: ", X, " / ", Y, " = ", Z3), \_= readLine(),*

*Z4 = X ^ Y,*

*write("Степень: ", X, " ^ ", Y, " = ", Z4), nl, \_= readLine(),*

*write("Деления с выделением целого и остатка: ", X," и ",Y), nl,*

*\_= readLine(),*

*Z5 = X div Y,*

*write("Целая часть с округлением к минусу: ", X, " div ", Y, " = ", Z5),*

*\_= readLine(),*

*Z6 = X mod Y,*

*write("Остаток с округлением к минусу: ", X, " mod ", Y, " = ", Z6),*

*\_= readLine(),*

*Z7 = X quot Y,*

*write("Целая часть с округлением к 0: ", X, " quot ", Y, " = ", Z7),*

*\_= readLine(),*

*Z8 = X rem Y,*

*write("Остаток с округлением к 0: ", X, " rem ", Y, " = ", Z8),*

*\_= readLine(), nl,*

*write("Конец, Нажмите: Enter"), \_= readLine(). % Ждем Enter*

*end implement main*

*goal*

*mainExe::run(main::run).*

В данном примере выполняются разные математические операции над двумя операндами X и Y, такие как:

* суммирование (+);
* нахождение разности (-);
* произведение чисел (\*);
* частное деление (/);
* возведение в степень (^);
* нахождение целой части числа с округлением к минусу (div);
* нахождение остатка от деления с округлением к минусу (mod);
* нахождение целой части с округлением к 0 (quot);
* нахождение остатка от деления с округлением к 0 (rem).

После запуска программы появляется окно консоли с именем первого аргумента «X =» в первой строке. После его ввода пользователю необходимо нажать Enter. В следующей строке появляется имя второго аргумента «Y =» и т. д. Далее при каждом нажатии Enter в новой строке отображаются результаты работы программы. Когда все строки показано, отображается строка с сообщением о конце вычислений.

### **Тригонометрические функции**

Перейдём к рассмотрению примера использования группы тригонометрических функций консольного проекта:

Листинг 2.4

*implement main*

*open core, console, math*

*clauses*

*run() :-*

*init(),*

*write("X градусы = "), X0 = read(),*

*X = X0 / 180 \* pi, nl, % Перевод в радианы*

*write("X в градусах и радианах ",X0," и ",X), nl, nl, \_= readLine(),*

*Z=sin(X),*

*write("Синус от ",X0," градусов = ", Z), \_= readLine(),*

*Z10 = arcsin(Z),*

*Z1 = Z10 \* 180 / pi, % Перевод в градусы*

*write("Арксинус от: ", Z, " = ", Z1, " градусов"), \_= readLine(),*

*Z2 = cos(X),*

*write("Косинус от ", X0, " градусов = ", Z2), \_= readLine(),*

*Z30 = arccos(Z2),*

*Z3 = Z30 \* 180 / pi, % Перевод в градусы*

*write("Арккосинус от ", Z2, " = ", Z3, " градусов"), \_= readLine(),*

*Z4 = tan(X),*

*write("Тангенс от ", X0, " градусов = ", Z4), \_= readLine(),*

*Z50 = arctan(Z4),*

*Z5 = Z50 \* 180 / pi, % Перевод в градусы*

*write("Арктангенс от ", Z4, " = ", Z5, " градусов"), nl,nl,*

*write("Конец. Нажмите Enter"), \_= readLine().*

*end implement main*

*goal*

*mainExe::run(main::run).*

В данном примере выполняются прямые и обратные тригонометрические функции с операндом X в градусах, а именно:

* синус (sin) и арксинус (arcsin);
* косинус (cos) и арккосинус (arccos);
* тангенс (tan) и арктангенс (arctan).

После запуска программы появляется окно консоли с именем аргумента «X градусы» в первой строке. Пользователю необходимо ввести значение в градусах и нажать Enter. Далее при каждом нажатии Enter в новой строке отображаются результаты. Когда все показано, отображается строка с сообщением о конце вычислений.

### **Базовые функции**

В следующем примере будет рассмотрена группа базовых функций консольного приложения:

Листинг 2.5

*implement main*

*open core, console, math*

*clauses*

*run():-*

*init(),*

*write("Базовые функции класса math "), nl, nl,*

*write("Отрицательное X = "), X = read(),*

*write("Положительное Y = "), Y = read(), nl,*

*write("Исходные числа ",X," и ",Y), nl, nl, \_= readLine(),*

*Z = abs(X),*

*write("Абсолютное значение abs(",X,") = ", Z), \_= readLine(),*

*Z1 = sqrt(Y),*

*write("Квадратный корень из ", Y, " = ", Z1), \_= readLine(),*

*Z2 = sqr(X),*

*write("Квадрат ", X, " = ", Z2), \_= readLine(),*

*Z3 = max(X,Y),*

*Z4 = min(X,Y),*

*write("Минимальное min(", X, ",", Y, ") = ", Z4), \_= readLine(),*

*Z5 = ln(Y),*

*write("Натуральный логарифм ln(", Y, ") = ", Z5), \_= readLine(),*

*Z6 = log(Y),*

*Z7 = exp(X),*

*write("Экспонента, exp(", X, ") = ", Z7), \_= readLine(), nl,*

*write("Конец. Нажмите Enter "),*

*\_= readLine(),nl.*

*end implement main*

*goal*

*mainExe::run(main::run).*

В данном примере выполняются базовые функции с отрицательным операндом X и положительным операндом Y, такие как:

* нахождение модуля числа (abs);
* квадратный корень числа (sqrt);
* поднесение числа в квадрат (sqr);
* нахождение максимального числа (max);
* минимального числа (min);
* натуральный логарифм числа (ln);
* десятичный логарифм (log);
* нахождение экспоненты (exp).

После запуска проекта появляется окно консоли с заголовком и приглашениями ввода отрицательного операнда X и положительного операнда Y. Далее, как в предыдущих примерах, при каждом нажатии Enter в новой строке отображаются результаты. По окончанию вывода, отображается строка с сообщением о конце вычислений.

## **Создание модулей**

Модульные проекты применяются для упрощения программирования.

В качестве примера, будет рассмотрена программа, которая анализирует список персон. Для персон задаются имя и возраст. Сначала в консоль выводится список имен, затем список возрастов и в конце средний возраст персон.

Для того, чтобы создать модуль, следует выделить корень дерева проекта и выбрать команду (всплывающего) меню «**File/New In New Package»** в меню задач, либо нажать комбинацию клавиш «Ctrl+N». В открывшемся диалоговом окне «Create Project Item» необходимо выбрать в левом поле раздел Class, убрать флажок из поля Create Interface, ввести в поле Name имя модуля, в нашем случае «person», и нажать на кнопку Create, рис. 2.6.

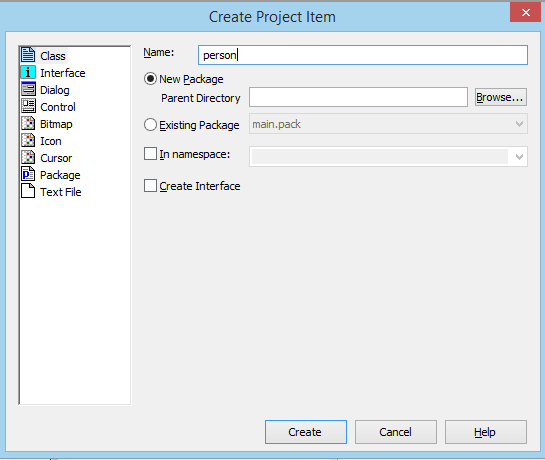


Рисунок 2.6 – Создание модуля

После того, как модуль добавится, в дерево проекта появится новая папка «person», в которой доступны файлы для изменения person.pro и person.cl.

В файле person.cl описан класс, через который будет идти обращение в главный файле проекта main.pro, поэтому необходимо добавить в него объявление предикатов:

Листинг 2.6

*class person*

*open core*

*domains*

*name = string.*

*age = integer.*

*age\_list = integer\_list.*

*predicates*

*person : (name, age) multi (o,o). % Студенты*

*sumlist : (age\_list, age, integer) procedure(i,o,o). % Список их данных*

*end class person*

Теперь нужно перейти к определению этих предикатов в файле person.pro:

*implement person*

*open core*

*clauses*

*% Пустой список, сумма возрастов Sum=0 и число студентов N=0*

*sumlist([],0,0).*

*sumlist([H|T], Sum, N):-*

*sumlist(T, S1,N1), % Вызов для очередного факта*

*Sum=H+S1, % Копим сумму*

*N=1+N1. % увеличиваем N на 1*

*% Список фактов*

*person("Иванов", 42).*

*person("Петров", 36).*

*person("Сидоров", 36).*

*end implement person*

Завершающим этапом является добавление функционала в главный файл проекта main.pro:

Листинг 2.7

*implement main*

*open core, console, person*

*clauses*

*run():-*

*init(),*

*write("Список студентов: "),nl,*

*L1 = [ Name ||person(Name, \_)], % Получить список*

*write(L1),nl,nl,*

*write("Их возраст: "),nl,*

*L2 = [ Age || person(\_, Age)], % Получить список возрастов*

*write(L2),nl,nl,*

*% Получить из списка L2 сумму возрастов Sum и число студентов N*

*sumlist(L2, Sum, N),*

*Ave = Sum/N, % Средний возраст студентов*

*write("Средний возраст = ", Ave),nl,nl,*

*write("Нажмите Enter"),*

*\_= readLine().*

*end implement main*

*goal*

*mainExe::run(main::run).*

Как видно из примера, для обращения к предикатам класса person достаточно добавить его имя в строку open core, console.

## **Теоретические исследования, оборудование, материалы и наглядные пособия**

Объектом исследования данной лабораторной работы являются основные компоненты языка Пролог.

Для выполнения работы необходимы ПК и соответствующее программное обеспечение:

* MS WINDOWS;
* VisualProlog;
* MS Office (для оформления отчета).

## **Задание на работу (рабочее задание)**

1. Создайте программу, решающую квадратное уравнение.

2. Создайте предикат, имеющий пять аргументов, и проверяющий, попадает ли точка, чьи координаты заданы первыми двумя параметрами, в круг, центр которого определяют третий и четвертый параметр, а радиус - пятый.

3. Создайте предикат, находящий абсолютное значение числа.

4. Создайте предикат, вычисляющий длину гипотенузы прямоугольного треугольника по длинам катетов.

5. Создайте предикат, вычисляющий периметр треугольника по двум сторонам и углу между ними.

6. Создайте предикат, вычисляющий площадь треугольника по двум сторонам и углу между ними.

7. Создайте предикат, вычисляющий площадь вписанного правильного треугольника в окружность радиусом R.

8. Создайте предикат, вычисляющий площадь описанного правильного треугольника вокруг окружности радиусом R.

9. Создайте предикат, вычисляющий площадь окружности, описанной вокруг правильного треугольника со стороной а.

10. Создайте предикат, вычисляющий площадь окружности, вписанной в правильный треугольник со стороной а.

11. Создайте предикат, вычисляющий площадь четырехугольника по четырем точкам с проверкой на выпуклость.

12. Создайте предикат, вычисляющий площадь треугольника по трем точкам.

13. Создайте предикат, проверяющий является ли угол между двумя векторами острым, прямым или тупым.

14. Создайте предикат, вычисляющий длину вектора по координатам начала и конца.

15. Создайте предикат, проверяющий направлен ли вектор по оси Х или против.

16. Создайте предикат, проверяющий направлен ли вектор по оси Y или против.

17. Создайте предикат, определяющий проекцию вектора по заданной оси.

18. Создайте предикат, проверяющий направлен ли вектор по оси Z или против.

## **Ход работы (порядок выполнения работы)**

Для выполнения лабораторной работы необходимо:

1. Ознакомится теоретической справкой.
2. Выполнить математическое описание задания с использованием логики предикатов.
3. Разработать программное обеспечение.
4. Оформить отчет.
5. Защитить работу преподавателю.

## **Содержание отчета**

В отчете должны присутствовать следующие пункты:

1. Задание;
2. Математическая постановка задачи;
3. Текст подпрограммы;
4. Результат работы программы.

# **Рекурсивные программы**

## **Цель и задачи работы**

Изучить и освоить на примерах использование механизма рекурсии в Прологе.

## **Теоретические положения**

Очень часто в программах необходимо выполнить одну и ту же задачу несколько раз. В программах на Прологе повторяющиеся операции обычно выполняются при помощи правил, которые используют откат и рекурсию.

Используются следующие методы повторения:

* Метод отката после неудачи;
* Метод отсечения и отката;
* Правило повтора, определяемое пользователем;
* Обобщенное рекурсивное правило.

Правила повтора и рекурсии должны содержать средства управления их выполнением с тем, чтобы их использование было удобными. Встроенные предикаты Пролога fail и cut используются для управления откатами, а условия завершения используются для управления рекурсией.

Цели управляют программой на Прологе, обеспечивая выполнение последовательности определенных задач. Цели могут содержать подцели. Цели и подцели могут содержать правила. Правила часто требуют, чтобы такие задачи, как поиск элементов в базе данных или вывод данных на экран выполнялись несколько раз.

Существуют два способа реализации правил, выполняющих одну и туже задачу многократно:

* повторение;
* рекурсия.

Правила Пролога, выполняющие повторения, используют откат, а правила, выполняющие рекурсию используют самовызов.

Формат правила, выполняющего повторение:

Листинг 3.1

*repetitive\_rule :- /\* правило повторения \*/*

*<предикаты и правила>,*

*fail. /\* неудача \*/*

Конструкция <**предикаты и правила**> в теле правила обозначает предикаты, содержащие несколько утверждений, а так же правила, определенные в программе. Встроенный предикат fail (неудача) вызывает откат, так что предикаты и правила выполняются еще раз.

Формат правила, выполняющего рекурсию:

Листинг 3.2

*recursive\_rule :-/\* правило рекурсии \*/*

*<предикаты и правила>,*

*recursive\_rule.*

Последним правилом в теле данного правила является само правило recursive\_rule. Правила рекурсии содержат в теле правила сами себя.

Правила повтора и рекурсии могут обеспечивать одинаковый результат, хотя алгоритмы их выполнения не одинаковы. Каждый из них в конкретной ситуации имеет свои преимущества.

## **Нисходящая рекурсия**

Листинг 3.3

В качестве примера рекурсии станет нахождение факториала:

*implement main*

*open core, console*

*class predicates*

*fact:(integer, integer) procedure (i, o).*

*clauses*

*fact(0, 1) :- !.*

*fact(N, F) :- fact(N-1, X), F = N \* X.*

*run():-*

*init(),*

*write("Введите число n = "),*

*N=read(),*

*fact(N, F),*

*write(N, "! = ", F),*

*\_=readLine(), nl,*

*write("Конец. Нажмите Enter "),*

*\_=readLine(),*

*nl.*

*end implement main*

*goal*

*mainExe::run(main::run).*

Данная рекурсия является нисходящей. Нисходящая рекурсия последовательно разбивает задачу на все более простые, пока не доходит до граничной ситуации, в которой уже не требуется продолжения рекурсии.

## **Хвостовая рекурсия**

Рекурсия – хвостовая, если вызов предиката самого себя идет последним в правиле, при этом до него нет недетерминированных вызовов. Хвостовая рекурсия соответствует итерации в процедурных языках программирования.

В качестве примера хвостовой рекурсии станет определение факториала. В нем используются счетчик C и накопитель для хранения произведения первых C натуральных чисел:

Листинг 3.4

*implement main*

*open core, console*

*class predicates*

*fact:(integer, integer) procedure (i, o).*

*fact1: (integer, integer, integer, integer) procedure (i, i, i, o).*

*clauses*

*fact(N, F):- fact1(N, 0, 1, F).*

*fact1(N, N, F, F):- !.*

*fact1(N, C, X, F):- fact1(N, C + 1, (C + 1) \* X, F).*

*run():-*

*init(),*

*write("Введите число n = "),*

*N=read(),*

*fact(N, F),*

*write(N, "! = ", F),*

*\_=readLine(), nl,*

*write("Конец. Нажмите Enter "),*

*\_=readLine(),*

*nl.*

*end implement main*

*goal*

*mainExe::run(main::run).*

Данный пример является восходящей рекурсии. В восходящей рекурсии промежуточные результаты вычисляются на каждом шаге рекурсии, так что ответ строится постепенно и передается в виде параметра рабочей памяти до тех пор, пока не будет достигнута конечная ситуация. К этому моменту ответ уже будет вычислен.

## **Теоретические исследования, оборудование, материалы и наглядные пособия**

Объектом исследования данной лабораторной работы является рекурсия среды Visual Prolog.

Для выполнения работы необходимы ПК и соответствующее программное обеспечение:

* MS WINDOWS;
* VisualProlog;
* MS Office (для оформления отчета).

## **Задание на работу (рабочее задание)**

* 1. Создайте предикат, вычисляющий неотрицательную степень целого числа.
  2. Создать предикат, вычисляющий по натуральному числу N сумму чисел от 1 до N.
  3. Создайте предикат, вычисляющий по натуральному числу N сумму нечетных чисел, не превосходящих N.
  4. Создайте предикат, вычисляющий наибольший общий делитель двух натуральных чисел.
  5. Создайте предикат, вычисляющий наименьшее общее кратное двух натуральных чисел.
  6. Создайте предикат, вычисляющий отрицательную целую степень действительного числа.
  7. Создайте предикат, вычисляющий функцию sin(x) с заданной точностью.
  8. Создайте предикат, вычисляющий функцию sin(x), используя разложение в ряд по заданному количеству членов ряда.
  9. Создайте предикат, вычисляющий по натуральному числу N сумму четных чисел, не превосходящих N.
  10. Создайте предикат, вычисляющий функцию exp(x), используя разложение в ряд по заданному количеству членов ряда.
  11. Создайте предикат, вычисляющий функцию exp (x), используя разложение в ряд по заданному количеству членов ряда.
  12. Создайте предикат, вычисляющий функцию cos(x) с заданной точностью.
  13. Создайте предикат, вычисляющий функцию cos(x), используя разложение в ряд по заданному количеству членов ряда.
  14. Создайте предикат, вычисляющий рекурсивно значение, используя функцию квадратного корня.
  15. Создать предикат, вычисляющий по натуральному числу N произведение чисел от (N div 2) до N.
  16. Создайте предикат, вычисляющий по натуральному числу N среднеарифметическое натуральных четных чисел, не превосходящих N.
  17. Создайте предикат, вычисляющий среднегеометрическое натуральных чисел, кратных 3, не превосходящих N.
  18. Создайте предикат, выводящий простые числа на интервале

1. (0, N). Использовать предикат, вычисляющий НОД.
   1. Создайте предикат, выводящий все делители числа N.
   2. Создайте предикат, вычисляющий функцию tg(x) с заданной точностью.
   3. Создайте предикат, вычисляющий функцию tg(x), используя разложение в ряд по заданному количеству членов ряда.
   4. Создайте предикат, вычисляющий по натуральному числу N произведение чисел, кратных 3 не превосходящих N.
   5. Создайте предикат, вычисляющий функцию ln(x), используя разложение в ряд по заданному количеству членов ряда.
   6. Создайте предикат, вычисляющий функцию ln(x), используя разложение в ряд по заданному количеству членов ряда.
   7. Создайте предикат, вычисляющий функцию ctg(x) с заданной точностью.
   8. Создайте предикат, вычисляющий функцию ctg(x), используя разложение в ряд по заданному количеству членов ряда.

## **Ход работы (порядок выполнения работы)**

Для выполнения лабораторной работы необходимо:

1. Ознакомится теоретической справкой.
2. Выполнить математическое описание задания с использованием логики предикатов.
3. Разработать программное обеспечение.
4. Оформить отчет.
5. Защитить работу преподавателю.

## **Содержание отчета**

В отчете должны присутствовать следующие пункты:

1. Задание;
2. Математическая постановка задачи;
3. Текст подпрограммы;
4. Результат работы программы.

# **Списки**

## **Цель и задачи работы**

Изучить структуру создания списков в языке Visual Prolog.

## **Объявление списков**

Список — это упорядоченная последовательность элементов, где слово упорядоченная означает, что порядок перечисления элементов имеет значение. В Прологе список заключается в квадратные скобки, а элементы разделяются запятой. Список обычно имеет голову (head) и хвост (tail).

Для объявления домена - списка целых используется декларация домена, пример:

Листинг 4.1

*domains*

*integer\_list = integer\*.*

Звездочка означает «список этого»; то есть, integer\* означает «список целых».

Элементами в списке может быть, что угодно, включая другие списки. Но все элементы в списке должны принадлежать одному домену, и дополнительно к декларации спискового домена должна быть декларация domains для элементов:

Листинг 4.2

*domains*

*element\_list = elements\*.*

*elements = ....*

Visual Prolog не допускает смешивание стандартных типов в списке. Выходом для объявления списков из integer, real и symbols является объявление домена общего для всех типов, где функтор показывает какому типу принадлежит тот или иной элемент, пример:

Листинг 4.3

*element\_list = elements\*.*

*elements =*

*i(integer);*

*r(real);*

*(symbol).*

*/\* функторами являются i, r и s \*/*

## **Головы и хвосты**

Список на самом деле является рекурсивным составным объетом. Он состоит из двух частей - головы списка, которым является первый элемент, и хвоста - списка, который включает все следующие элементы.

**Хвост списка всегда есть список, а голова списка есть элемент.**

Пустой список не может быть разбит на голову и хвост. Это означает, что, концептуально говоря, списки имеют древовидную структуру подобно другим составным объектам. Древовидная структура списка [a, b, c, d] представлена на рис. 4.1.

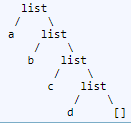


Рисунок 4.1 – Древовидная структура списка

## **Представления списков**

Пролог содержит метод для явного обозначения головы и хвоста списка. Вместо разделения элементов запятыми можно отделять голову от хвоста вертикальной чертой (|), например,

*[a, b, c] эквивалентно [a|[b, c]]*

Примеры списков представлены в табл. 1.

Таблица 1

Пример списков на языке Пролог

|  |  |  |  |
| --- | --- | --- | --- |
| **Список** | **Тип** | **Голова** | **Хвост** |
| [3, 4, 5, 6, 7] | Integer | 3 | [4, 5, 6, 7] |
| ["wo3", "ni3", "ta1"] | String | "wo3" | ["ni3", "ta1"] |
| [4] | Integer | 4 | [] |
| [3.4, 5.6, 2.3] | Real | 3.4 | [5.6, 2.3] |

Один и тот же элемент может быть представлен в списке несколько раз, например: *[1, 2, 1, 3, 1].*

## **Использование списков**

Поскольку списки являются в действительности рекурсивными составными структурами данных, для их обработки необходимы и рекурсивные алгоритмы. Самый естественный способ обработки списков - сквозной просмотр, в ходе которого что-то делается с каждым элементом, до тех пор, пока не достигнут конец.

Как правило, такого рода алгоритмы используют два клауза. Один из них говорит о том, как поступать с обыкновенным списком, который может быть разделен на голову и хвост. Другой говорит о том, что делать с пустым списком.

## **Вывод списка на печать**

Пример кода, выводящий на печать элементы списка:

Листинг 4.4

*class my*

*predicates*

*write\_a\_list : (integer\*).*

*end class*

*implement my*

*clauses*

*write\_a\_list([]). /\* Если список пустой, ничего не делаем. \*/*

*write\_a\_list([H|T]):- /\* Сопоставляем голову с H и хвост с T, и... \*/*

*stdio::write(H),stdio::nl, /\*выводим H и переводим строку\*/*

*write\_a\_list(T).*

*end implement*

*goal*

*console::init(),*

*my::write\_a\_list([1, 2, 3]).*

Здесь мы видим два клауза write\_a\_list, которые можно выразить но обычном языке:

* При выводе на печать пустого списка ничего не произойдет.
* Иначе, при выводе на печать списка, вывести на печать его голову (она есть просто элемент), и потом вывести на печать хвост списка (он, как известно, есть список).

## **Подсчет элементов в списке**

Рассмотрим пример, как подсчитать число элементов в списке или определим его длину.

* Длина пустого списка [] есть 0.
* Длина любого другого списка есть 1 плюс длина его хвоста.

Листинг 4.5

*class my*

*predicates*

*length\_of : (A\*, integer) procedure(i,o).*

*end class*

*implement my*

*clauses*

*length\_of([], 0).*

*length\_of([\_|T], L):-*

*length\_of(T, TailLength),*

*L = TailLength + 1.*

*end implement*

*goal*

*console::init(),*

*my::length\_of([1, 2, 3], L),*

*stdio::*write(L).

Строго говоря, [\_|T] сопоставляется с любым непустым списком, связывая T с хвостом списка. Значение головы неважно, если она есть, она может быть учтена как один элемент.

length\_of вызывает себя рекурсивно, сопоставляя вызов со вторым клаузом.

TailLength в вызове никак не пересекается с TailLength в клаузе, поскольку каждый рекурсивный вызов клауза имеет собственный набор переменных.

## **Хвостовая рекурсия**

Очевидно, что length\_of не является (и не может быть) предикатом с хвостовой рекурсией, поскольку рекурсивный вызов не является последним шагом в его клаузе. Возможно ли создать предикат, определяющий длину, так, чтобы он был предикатом с хвостовой рекурсией? Да, но это потребует некоторых усилий.

Проблема с предикатом length\_of в том, что длину списка нельзя вычислить до тех пор, пока не вычислена длина его хвоста. Но из этой ситуации есть выход. Для этого потребуется предикат, вычисляющий длину списка, с тремя аргументами.

Один из них – это список, от которого компьютер будет забирать по одному элементу на каждом вызове до тех пор, пока этот список, как и прежде, не превратится в пустой список.

Второй – это свободный аргумент, который в конечном итоге вернет результат (длину).

Третий – это счетчик, значение которого начинается с нуля и увеличивается с каждым вызовом.

Когда список в конечном итоге станет пустым, нужно проунифицировать счетчик с несвязанным результатом.

Листинг 4.6

*class my*

*predicates*

*length\_of : (A\*, integer, integer) procedure(i,o,i).*

*end class*

*implement my*

*clauses*

*length\_of([], Result, Result).*

*length\_of([\_|T], Result, Counter):-*

*NewCounter = Counter + 1,*

*length\_of(T, Result, NewCounter).*

*end implement*

*goal*

*console::init(),*

*my::length\_of([1, 2, 3], L, 0), /\* Начинаем со счетчиком Counter = 0 \*/*

*stdio::write(" L = ", L).*

Эта версия предиката length\_of более сложная и во многих смыслах менее логичная, чем предыдущая, но она для того, чтобы показать, что на практике можно часто построить алгоритм с хвостовой рекурсией для задач, которые на первый взгляд требуют рекурсии другого типа.

## **Модификация списка**

Иногда требуется создать другой список из заданного списка. Это делается путем просмотра списка, элемент за элементом, заменяя каждый элемент вычисленным значением, например, как эта программа, которая добавляет 1 к каждому элементу исходного списка:

Листинг 4.7

*class my*

*predicates*

*add1 : (integer\*, integer\*) procedure(i,o).*

*end class*

*implement my*

*clauses*

*add1([], [])./\* граничное условие \*/*

*add1([Head|Tail],[Head1|Tail1]):- /\* отделяем голову от остального списка\*/*

*Head1 = Head+1, /\* добавляем 1 к элементу-голове \*/*

*add1(Tail, Tail1)./\* далаем это с остальной частью списка\*/*

*end implement*

*goal*

*console::init(),*

*my::add1([1,2,3,4], NewList),*

*stdio::write(NewList)).*

На обычном языке это звучит так:

* Добавление 1 ко всем элементам пустого списка порождаем пустой список,
* Для добавления 1 ко всем элемента любого другого списка:
* добавить 1 к голове и сделать эту голову головой результирующего списка, а затем
* добавить 1 к каждому элемента хвоста и этот хвост сделать хвостом результата.

## **Теоретические исследования, оборудование, материалы и наглядные пособия**

Объектом исследования данной лабораторной работы являются списки языка Пролог.

Для выполнения работы необходимы ПК и соответствующее программное обеспечение:

* MS WINDOWS;
* VisualProlog;
* MS Office (для оформления отчета).

## **Задание на работу (рабочее задание)**

* 1. Создайте предикат, заменяющий в исходном списке первое вхождение заданного значения другим.
  2. Создайте предикат, заменяющий в исходном списке все вхождения заданного значения другим.
  3. Создайте предикат, порождающий по заданному натуральному числу N список, состоящий из натуральных чисел от 1 до N (по возрастанию).
  4. Создайте предикат, порождающий по заданному натуральному числу N список, состоящий из натуральных чисел от N до 1 (по убыванию).
  5. Создайте предикат, порождающий по заданному натуральному числу N список, состоящий из N случайных натуральных чисел из промежутка от 1 до 100.
  6. Создайте предикат, порождающий по заданным числам N, M, K список, состоящий из N случайных натуральных чисел из промежутка от M до K.
  7. Создайте предикат, порождающий по заданным числам M, K список, состоящий из случайного количества случайных чисел из промежутка от M до K.
  8. Создайте предикат, который увеличивает элементы исходного списка на величину K.
  9. Создайте предикат, переводящий список цифр от 0 до 9 в список соответствующих им названий (строк).
  10. Создайте предикат, переводящий список цифр от 0 до 9 в список соответствующих им римских чисел.
  11. Создайте предикат, переводящий список римских чисел в список соответствующих им арабских чисел (диапазон от 1 до 20).
  12. Создайте предикат, удваивающий значения элементов списка.
  13. Создайте предикат, преобразующий список, элементами которого являются числа, в список, элементы которого неотрицательны.
  14. Создайте предикат, преобразующий исходный список в список позиций отрицательных элементов.
  15. Создайте предикат, удаляющий из исходного списка элементы с четными номерами.
  16. Создайте предикат, который разделит исходный список из целых чисел на два списка: список положительных чисел и список отрицательных чисел.
  17. Создайте предикат, разделяющий исходный список на два подсписка. В первый из них должны попасть элементы с нечетными номерами, во второй - элементы с четными номерами.
  18. Создайте предикат, вычисляющий по списку и числу, подсписок исходного списка, начинающийся с элемента с указанным номером.
  19. Создайте предикат, осуществляющий удаление указанного количества последних элементов исходного списка.
  20. Создайте предикат, осуществляющий разделение исходного списка на два подсписка. В первый из них должно попасть указанное количество элементов из начала списка, во второй - оставшиеся элементы.
  21. Создайте предикат, осуществляющий разделение исходного списка на два подсписка. В первый из них должно попасть указанное количество элементов с конца списка, во второй - оставшиеся элементы.
  22. Создайте предикат, находящий предпоследний элемент списка.
  23. Создайте предикат, удаляющий предпоследний элемент списка.
  24. Создайте предикат, заменяющий в исходном списке два подряд идущих одинаковых элемента одним.
  25. Создайте предикат, удаляющий в исходном списке все повторные вхождения элементов.
  26. Создайте предикат, осуществляющий перестановку двух элементов списка с заданными номерами.
  27. Создайте предикат, генерирующий все перестановки элементов списка, указанного в качестве первого аргумента предиката.
  28. Создайте предикат, осуществляющий циклический сдвиг элементов списка на один влево (вправо).
  29. Создайте предикат, осуществляющий циклический сдвиг элементов списка на заданное количество шагов влево (вправо).
  30. Создайте предикат, осуществляющий поэлементное перемножение соответствующих элементов двух исходных списков.
  31. Создайте предикат, вычисляющий скалярное произведение векторов, заданных списками целых чисел.
  32. Создайте предикат, осуществляющий подсчет числа вхождений каждого элемента исходного списка. Ответом должен быть список пар, в которых первая компонента - элемент исходного списка, вторая - число его вхождений в первоначальный список.
  33. Создайте предикат, определяющий первую позицию подсписка в списке.
  34. Создайте предикат, добавляющий элементы одного списка во второй список, начиная с заданной позиции.
  35. Создайте предикат, возвращающий по списку и двум числам M и N подсписок исходного списка, состоящий из элементов с номерами от M до N.
  36. Создайте предикат, формирующий список простых чисел, не превосходящих данного числа.
  37. Создайте предикат, транспонирующий матрицу, заданную списком списков.

## **Ход работы (порядок выполнения работы)**

1. Для выполнения лабораторной работы необходимо:
2. Ознакомится теоретической справкой.
3. Выполнить математическое описание задания с ипспользованием логики предикатов.
4. Разработать программное обеспечение.
5. Оформить отчет.
6. Защитить работу преподавателю.

## **Содержание отчета**

В отчете должны присутствовать следующие пункты:

1. Задание;
2. Математическая постановка задачи;
3. Текст подпрограммы;
4. Результат работы программы.

# **. Множества**

## **Цель и задачи работы**

Получить навык создания множества элементов при программировании в среде Пролог и выполнения операции над ними.

## **Теоретические положения**

Множество – одна из наиболее важных структур данных, используемых как в математике, так и в программировании. Множество – это набор элементов, напоминающий список, но отличающийся тем, что вопрос о том, сколько раз и в каком месте что-либо входит в множество в качестве его элемента, не имеет смысла. Так, множество (1, 2, 3) – это то же самое множество, что и (1, 2, 3, 1), поскольку значение имеет только сам факт, принадлежит данный элемент множеству или нет. Элементами множеств могут также быть другие множества. Самой фундаментальной операцией над множествами является определение того, принадлежит некоторый элемент данному множеству или нет.

Поскольку Visual Prolog не имеет специального обозначения для множеств, то их удобно представлять в виде списков. Список может содержать произвольные элементы, включая другие списки, и над списками можно определить предикат принадлежности. Однако, когда множество представляется в виде списка, такой список содержит только по одному элементу на каждый объект, принадлежащий множеству. При работе со списками без повторяющихся элементов упрощаются некоторые операции, такие, как удаление элементов.

## **Множества чисел**

В качестве примера будет рассмотрен следующий пример кода консольного приложения:

Листинг 5.1

*implement main*

*open core*

*clauses*

*run() :-*

*console::init(),*

*Q =*

*[ tuple(X, Y) ||*

*X = std::fromTo(1, 4),*

*Y = std::fromTo(1, 5)*

*],*

*stdio::nl,*

*stdio::write(Q),*

*stdio::nl,*

*stdio::nl,*

*foreach tuple(Num, Den) = list::getMember\_nd(Q) do*

*stdio::write(Num, "/", Den, ", ")*

*end foreach,*

*stdio::nl.*

*end implement main*

*goal*

*console::runUtf8(main::run).*

**Q** — это список пар (X, Y), где, X принадлежит [1, 2, 3, 4], а Y принадлежит [1, 2, 3, 4, 5].

Следующий фрагмент кода выполняет write(Num, "/", Den,", ") для каждой пары (Num, Den), которая является членом списка Q:

Листинг 5.2

*foreach tuple(Num, Den) = list::getMember\_nd(Q) do*

*stdio::write(Num, "/", Den, ", ")*

*end foreach*

Множество рациональных чисел называется так потому, что его элементы могут быть представлены как дроби вида:

, (4.1)

где p ∈ Z и q ∈ N, q ≠ 0

## **Теоретические исследования, оборудование, материалы и наглядные пособия**

Объектом исследования данной лабораторной работы являются множества в среде Visual Prolog.

Для выполнения работы необходимы ПК и соответствующее программное обеспечение:

* MS WINDOWS;
* VisualProlog;
* MS Office (для оформления отчета).

## **Задание на работу (рабочее задание)**

1. Задание на работу (рабочее задание):
2. Создать предикат принадлежности элемента множеству объединения двух множеств
3. Создать предикат принадлежности элемента множеству пересечения двух множеств
4. Создать предикат принадлежности элемента множеству разности двух множеств
5. Создать предикат нахождения множества объединения трех множеств
6. Создать предикат нахождения множества пересечения трех множеств
7. Создать предикат нахождения множества разности трех множеств
8. Создать предикат нахождения множества, которое получается в результате выполнения следующих действий над тремя множествами:   
   
9. Создать предикат нахождения множества, которое получается в результате выполнения следующих действий над тремя множествами:   
   
10. Создать предикат нахождения множества, которое получается в результате выполнения следующих действий над тремя множествами:   
    
11. Создать предикат нахождения множества, которое получается в результате выполнения следующих действий над тремя множествами:   
    
12. Создать предикат нахождения множества, которое получается в результате выполнения следующих действий над тремя множествами:   
    
13. Создать предикат нахождения множества, которое получается в результате выполнения следующих действий над тремя множествами:   
    
14. Создать предикат нахождения множества, которое получается в результате выполнения следующих действий над тремя множествами:   
    
15. Создать предикат нахождения множества, которое получается в результате выполнения следующих действий над тремя множествами:   
    
16. Создать предикат определения множества объединения двух списков
17. Создать предикат определения множества пересечения двух списков
18. Создать предикат определения множества разности двух списков
19. Создать предикат определения множества точек единичного квадрата, попадающих в заштрихованную область, представленную на рисунке:



1. Создать предикат определения множества точек единичного квадрата, попадающих в заштрихованную область, представленную на рисунке:



1. Создать предикат определения множества точек единичного квадрата, попадающих в заштрихованную область, представленную на рисунке:



1. Создать предикат определения множества точек единичного квадрата, попадающих в заштрихованную область, представленную на рисунке:



1. Создать предикат определения множества точек единичного квадрата, попадающих в заштрихованную область, представленную на рисунке:



1. Создать предикат определения множества точек единичного квадрата, попадающих в заштрихованную область, представленную на рисунке:



1. Создать предикат определения множества точек единичного квадрата, попадающих в заштрихованную область, представленную на рисунке:



1. Создать предикат определения множества точек единичного квадрата, попадающих в заштрихованную область, представленную на рисунке:



1. Создать предикат определения множества точек единичного квадрата, попадающих в заштрихованную область, представленную на рисунке:



1. Создать предикат определения множества точек единичного квадрата, попадающих в заштрихованную область, представленную на рисунке:



1. Создать предикат определения множества точек единичного квадрата, попадающих в заштрихованную область, представленную на рисунке:



1. Создать предикат определения множества точек единичного квадрата, попадающих в заштрихованную область, представленную на рисунке:



## **Ход работы (порядок выполнения работы)**

Для выполнения лабораторной работы необходимо:

* 1. Изучить теоретическую справку;
  2. Написать программу на зыке Пролог по индивидуальному заданию;
  3. Оформить отчет;
  4. Защитить работу преподавателю.

## **Содержание отчета**

В отчете должны присутствовать следующие пункты:

1. Задание;
2. Математическая постановка задачи;
3. Текст подпрограммы;
4. Результат работы программы.

# **Строки**

## **Цель и задачи работы**

Получить навыки работы со строками на языке Пролог.

## **Теоретические положения**

Обработка строк в Пролог выполняется с помощью встроенных предикатов.

Строки и их обработка подчиняются определенным правилам. Так, в строках Visual Prolog обратный слэш (\) является управляющим символом, позволяющим вставлять в строки символы, которых нет на клавиатуре.

Ниже приведен список основных предикатов управления строк.

**Предикат frontChar**

Предикат frontChar действует согласно равенству: String1 = объединение Char и String2 и имеет следующий формат:

*frontChar (Stringl, Char, String2).*

Предикат frontChar имеет три аргумента: первый из них – строка, второй – символ (первый символ первой строки), третий – остаток первой строки.

Предикат frontChar можно использовать для расщепления строки в последовательность символов или для создания строки из последовательности символов, а также для проверки символов в строке. Если аргумент String1 связан со строкой нулевой длины, то предикат завершается не успешно.

**Предикат frontToken**

Предикат frontToken выполняет три взаимосвязанные функции, в зависимости от типа потока аргументов, который используется для обращения к нему. Предикат frontToken имеет следующий формат:

*frontToken (Stringl, Token, Rest).*

В случае потока (i, o, o) frontToken находит первую лексему в String1, связывает ее с Token, а остаток String1 связывает с Rest.

Варианты потока (i, i,o), (i, o, i), а также (i, i, i) служат для проверки: если связанные аргументы соответствуют частям String1 (первой лексеме, всему, что находится после первой лексемы, или же тому и другому), то frontToken завершается успешно, в противном случае – не успешно.

В случае если использован поток (o, i, i), предикат создает объединение Token и Rest, связывая String1 с результатом.

Последовательность знаков является лексемой, если это либо имя в соответствии с синтаксисом Visual Prolog, либо число (предшествующий ему знак является отдельной лексемой), либо отличный от пробела знак.

Предикат frontToken отлично приспособлен для разбиения строки на лексические символы.

**Предикат front:**

Предикат front расщепляет String1 на две части и имеет следующий синтаксис:

*front(String1, NumberOfChars, StartStr, EndStr).*

Аргумент StartStr содержит NumberOfChars первых символов из String1, a EndStr содержит остаток.

**Предикат concat**

Предикат concat устанавливает, что строка StringS является результатом сцепления String1 и String2. Он имеет форму:

*StringS = concat(String1, String2).*

Пример:

При вызове Str = concat("croco", "dile"), Str = "crocodile".

**Предикат length**

Предикат length определяет длину строки. Он имеет формат:

*Length = length(StringArg).*

Предикат length связывает переменную Length с длиной строки StringArg. Length имеет тип charCount.

**Предикат create**

Предикат create создает строку указанной длинны. Он имеет формат:

*Str = create(Length) или Str = create(Length, Str1).*

Предикат create связывает переменную Str с длиной строки Length. То есть, если Srt1 не задана, то создается пустая строка длинной Length, в другом случае в Str записывается строка Str1 и дублируется пока не будут заполнены все Length-символов.

**Предикат isName**

Предикат isName проверяет, является ли аргумент допустимым именем согласно синтаксису Visual Prolog, и имеет формат:

*isName(String).*

Имя начинается с буквы алфавита или символа подчеркивания, за которым следует любое число букв, цифр и символов подчеркивания. Предыдущие и последующие пробелы игнорируются.

**Предикат format**

При необходимости отформатировать числа или другие простые термы при преобразовании их в строку, можно использовать функцию форматирования format.

Пример кода:

Листинг 6.1

*implement main*

*open core, console, string*

*clauses*

*run():-*

*init(),*

*Str1= format("%8.3f\n %10.1e\n", 3.45678, 35678.0),*

*Str2= format("%d\n %10d\n", 456, 18),*

*Str3= format("%-10d\n %010d\n", 18, 18),*

*write(Str1, Str2, Str3),*

*\_ = readLine().*

*end implement main*

*goal*

*mainExe::run(main::run).*

В примере формат "%8.3f\n" означает, что нужно отобразить действительное число и выполнить возврат каретки; ширина поля, отведённого для числа, равна 8 знакам, число должно быть отображено с тремя знаками после запятой.

Формат "%010d\n" выражает требование целого числа, выровненного по правому краю поля шириной 10; пустые места поля должны быть заполнены нулями.

Формат "%-10d\n" определяет представление целого числа, выровненного по левому краю поля шириной 10; знак минус указывает на левое выравнивание, правое выравнивание установлено по умолчанию.

Формат "%10.1e\n" определяет научную запись для действительных чисел.

**Предикат subСhar**

Предикат subСhar возвращает символ на данной позиции строки:

*Char = subChar(String, Position).*

Первый символ строки имеет позицию 1. Например: Char = subChar("ABC", 2) свяжет Char с C, так как отсчет начинается с нуля. Если позиция определяет символ за концом строки (несуществующий символ), subChar завершится с ошибкой.

**Предикат subString**

Предикат subString возвращает часть строки. Его формат:

*Str\_out = subString(Str\_in, Pos, Len).*

Переменная Str\_out будет связана с копией части строки Str\_in, начиная с символа на позиции Роs и длиной Len. Например: SubStr = subString("GOLORP", 1, 3) связывает SubStr с "OLO". Если Pos и Len задают строку частично или полностью вне границ Str\_in, то subString завершается с ошибкой. Однако запрос 0 байт на самом конце строки не является ошибкой: SubStr = subString("ABC", 4, 0) свяжет SubStr с пустой строкой, тогда как SubStr = subString("ABC", 4, 1) является ошибкой, так же, как и SubStr = subString("ABC",5,-1).

## **Теоретические исследования, оборудование, материалы и наглядные пособия**

Объектом исследования данной лабораторной работы являются строки в среде Visual Prolog.

Для выполнения работы необходимы ПК и соответствующее программное обеспечение:

* MS WINDOWS;
* VisualProlog;
* MS Office (для оформления отчета).

## **Задание на работу (рабочее задание)**

1. Создайте предикат, который будет находить последнюю позицию вхождения символа в строку.
2. Создайте предикат, который подсчитает общее количество латинских букв в списке символов.
3. Создайте предикат, который будет удалять из данной строки все вхождения заданного символа.
4. Создайте предикат, удаляющий из данной строки все повторные вхождения символов.
5. Создайте предикат, который продублирует вхождение каждого символа в строку.
6. Создайте предикат, "переворачивающий" строку (меняющий в строке порядок символов на обратный).
7. Создайте предикат, проверяющий, является ли данная строка палиндромом.
8. Создайте предикат, составляющий список символов, которые входят одновременно в обе данных строки.
9. Создайте предикат, находящий в исходной строке слово максимальной длины.
10. Создайте предикат, преобразующий строку в список слов, состоящих из четного количества символов.
11. Создайте предикат, преобразующий строку в список слов, которые упорядочены по длине по возрастанию.
12. Создайте предикат, преобразующий исходную строку в строку, состоящую из первых букв слов первоначальной строки.
13. Создайте предикат, преобразующий исходную строку в строку, состоящую из последних букв слов первоначальной строки.
14. Создайте предикат, проверяющий правильность расстановки скобок в исходной строке.
15. Создайте предикат, меняющий местами первую и последнюю буквы в каждом слове исходной строки.
16. Создайте предикат, который будет находить n-ную позицию вхождения символа в строку.
17. Создайте предикат, который подсчитает общее количество цифр в списке символов.
18. Создайте предикат, который будет удалять из данной строки повторные вхождения заданного символа.
19. Создайте предикат, удаляющий из данной строки все первые вхождения символов, если символ входит не один раз.
20. Создайте предикат, который продублирует вхождение заданного символа в строку.
21. Создайте предикат, "переворачивающий" слова в строке (меняющий в словах порядок символов на обратный).
22. Создайте предикат, приписывающий в конец каждого слова его первый символ.
23. Создайте предикат, составляющий список номеров символов, которые входят одновременно в обе данных строки.
24. Создайте предикат, находящий в исходной строке слово минимальной длины.
25. Создайте предикат, преобразующий строку в список слов, состоящих из нечетного количества символов.
26. Создайте предикат, преобразующий строку в список слов, которые упорядочены по длине по убыванию.
27. Создайте предикат, преобразующий исходную строку в строку, состоящую из вторых букв слов первоначальной строки.
28. Создайте предикат, преобразующий исходную строку в строку, состоящую из предпоследних букв слов первоначальной строки.

## **Ход работы (порядок выполнения работы)**

Для выполнения лабораторной работы необходимо:

1. Изучить теоретическую справку;
2. Написать программу на зыке Пролог по индивидуальному заданию;
3. Оформить отчет;
4. Защитить работу преподавателю.

## **Содержание отчета**

В отчете должны присутствовать следующие пункты:

1. Задание;
2. Математическая постановка задачи;
3. Текст подпрограммы;
4. Результат работы программы.

# **Деревья**

## **Цель и задачи работы**

Получить навыки создания базы знаний на языке Пролог с помощью дерева.

## **Теоретические положения**

**Деревом** называется граф, у которого одна вершина корневая, остальные вершины имеют только одного отца и все вершины являются потомками корневой вершины.

**Листом дерева** называется его вершина, не имеющая сыновей. Кроной дерева называется совокупность всех листьев.

**Высотой дерева** называется наибольшая длина пути от корня к листу.

## **Дерево поиска**

Дерево поиска — это распространённая диаграмма, на которой каждое состояние представляется в виде помеченной вершины, а каждый допустимый переход из одного состояния в другое описывается в виде ветви перевернутого дерева, рис. 7.1.

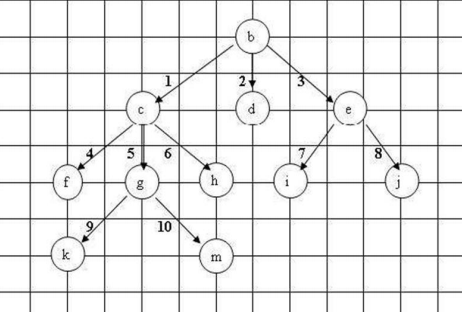


Рисунок 7.1 – Дерево поиска

Вершины b, c, d, t, f, g, h, i, j, k и m являются состояниями, через которые может пройти объект. Ветви, соединяющие две вершины, соответствуют операторам, которые вызывают переход из одного состояния в другое.

Например, ветвь под номером 3 – это оператор, который вызывает переход из состояния b в состояние e. Если вершина X может быть достигнута из другой вершины Y с помощью только одного оператора, говорят, что X — это дочь Y и что Y — это мать X. Таким образом, вершина с является дочерью вершины b, так как c можно достичь из b с помощью оператора 1. Если у вершины нет дочери, она называется листом. Вершина, у которой нет матери, является корнем дерева поиска.

## **Поиск в ширину**

Поиск называется исчерпывающим, если он гарантирует порождение всех достижимых состояний до того, как он завершится по неудаче. Одним из способов осуществить исчерпывающий поиск является генерация всех вершин на определённом уровне, до того как произойдет продвижение к следующему уровню дерева. Этот метод называется поиском в ширину. Он гарантирует, что пространство допустимых операций будет рассматриваться систематически.

Для дерева, приведенного на рисунке 7.1, поиск начинается с рассмотрения уровня (c-d-e), содержащего вершины, расположенные на расстоянии одной ветви от корня. Затем он переходит к уровню (f-g-h-i-j), расположенному на расстоянии двух ветвей от корня. Наконец, механизм поиска посещает уровень (k-m). Программа, выполняющая операцию поиска, обладает параметром, который называется Очередь. Он содержит список путей – кандидатов. Каждый путь представляется в виде списка вершин, при этом вершина описывается в виде *r(integer, string).*

Пример поиска в ширину:

Листинг 7.1

*implement main*

*domains*

*node= r(integer, string).*

*path= node\*.*

*queue= path\*.*

*class facts*

*operator:(integer, string, string).*

*class predicates*

*bSearch:(queue, path) determ (i, o).*

*nextLevel:(path, path) nondeterm (i, o).*

*solved:(path) determ.*

*prtSolution:(path).clauses*

*classInfo("main", "breath").*

*solved(L) :- L= [r(\_, "k")|\_].*

*prtSolution(L) :-*

*foreach P= list::getMember\_nd(list::reverse(L)) do*

*stdio::write(P), stdio::nl*

*end foreach.*

*operator(1, "b", "c").*

*operator(2, "b", "d").*

*operator(3, "b", "e").*

*operator(4, "c", "f").*

*operator(5, "c", "g").*

*operator(6, "c", "h").*

*operator(7, "e", "i").*

*operator(8, "e", "j").*

*operator(9, "g", "k").*

*operator(10, "g", "m").*

*bSearch([T|Queue],Solution) :-*

*if solved(T) then Solution= T*

*else Extentions= [Daughter || nextLevel(T, Daughter)],*

*ExtendedQueue= list::append(Queue, Extentions),*

*bSearch(ExtendedQueue, Solution) end if.*

*nextLevel([r(Branch, N)|Path], [r(Op, Daughter),r(Branch, N)|Path]):-*

*operator(Op, N, Daughter),*

*not(list::isMember(r(Op, Daughter),Path)).*

*run():- console::init(),*

*if bSearch([[r(0, "b")]], L) then prtSolution(L)*

*else stdio::write("No solution!"), stdio::nl end if.*

*end implement main /\* breath\*/*

*goal mainExe::run(main::run).*

Изначально **Queue** содержит единственный путь, который начинается и заканчивается в корне:

Листинг 7.2

*bSearch([[r(0, "b")]], L)*

Так как этот путь не ведёт к цели, он продолжается до каждой дочери вершины «b», при этом продолжения дописываются в конец очереди:

Листинг 7.3

*bSearch([ [r(1, "c"),r(0, "b")],*

*[r(2, "d"),r(0, "b")],*

*[r(3, "e"),r(0, "b")]], L)*

Пока программа посетила только одну вершину, а именно вершину «b». Затем в конец очереди дописываются продолжения из вершины «c»:

Листинг 7.4

*bSearch([ [r(2, "d"),r(0, "b")],*

*[r(3, "e"),r(0, "b")],*

*[r(4, "f"),r(3, "c"),r(0, "b")],*

*[r(5, "g"),r(3, "c"),r(0, "b")],*

*[r(6, "h"),r(3, "c"),r(0, "b")]], L)*

Вершина «d» достигнута, но, так как дочерей у нее нет, этот путь просто удаляется из очереди:

Листинг 7.5

*bSearch([ [r(3, "e"),r(0, "b")],*

*[r(4, "f"),r(3, "c"),r(0, "b")],*

*[r(5, "g"),r(3, "c"),r(0, "b")],*

*[r(6, "h"),r(3, "c"),r(0, "b")]], L)*

Теперь дочери вершины «e» дописываются в конец очереди:

Листинг 7.6

*bSearch([ [r(4, "f"),r(3, "c"),r(0, "b")],*

*[r(5, "g"),r(3, "c"),r(0, "b")],*

*[r(6, "h"),r(3, "c"),r(0, "b")],*

*[r(7, "i"), r(3, "e"),r(0, "b")],*

*[r(8, "j"), r(3, "e"),r(0, "b")]], L)*

Вершина «f», не имеющая потомков, удаляется из очереди, и в конец очереди добавляются дочери вершины «g»:

Листинг 7.7

*bSearch([ [r(6, "h"),r(3, "c"),r(0, "b")],*

*[r(7, "i"), r(3, "e"),r(0, "b")],*

*[r(8, "j"), r(3, "e"),r(0, "b")],*

*[r(9, "k"), r(5, "g"),r(3, "c"),r(0, "b")],*

*[r(10, "m"), r(5, "g"),r(3, "c"),r(0, "b")]], L)*

Вершины «h», «i» и «j» посещаются поочередно и удаляются из очереди. Поиск заканчивается, когда алгоритм находит путь:

[r(9, "k"), r(5, "g"),r(3, "c"),r(0, "b")]

Процедура **prtSolution** разворачивает этот путь и распечатывает каждый его элемент:

Листинг 7.8

*prtSolution(L) :-*

*foreach P= list::getMember\_nd(list::reverse(L)) do*

*stdio::write(P), stdio::nl*

*end foreach.*

Конечный результат, который выводится на экран:

*r(0,"b")*

*r(1,"c")*

*r(5,"g")*

*r(9,"k")*

## **Поиск в глубину**

Поиск в глубину после посещения вершины «b» перемещается к ее самой левой дочери — дочери «c». После посещения «c» он идёт к вершине «f». Только после посещения всех потомков “c” поиск в глубину посещает вершину «g». Для того чтобы реализовать эту стратегию, всё, что вам нужно сделать – это дописывать потомков текущей вершины в начало очереди **Queue**, а не в ее конец.

Единственная необходимая модификация, сделанная для предыдущей программы, с изменением названия предиката на **dSearch**:

Листинг 7.9

*dSearch([T|Queue],Solution) :-*

*if solved(T) then Solution= T*

*else Extentions= [Daughter || nextLevel(T, Daughter)],*

*ExtendedQueue= list::append(Extentions, Queue),*

*dSearch(ExtendedQueue, Solution) end if.*

## **Эвричтический поиск**

Стратегия данного поиска оценивает стоимость перемещения от корня к цели через вершину N до перехода в N. Функция стоимости определяется по следующей формуле:

f(N) = g(N) + h(N) (7.1)

В этой формуле используются следующие обозначения:

g(N) – это известная стоимость перемещения из корня до вершины N;

функция h(N) возвращает оценку стоимости перемещения из N до целевой вершины. Эвристическая стратегия сортирует очередь таким образом, чтобы пути с наименьшими значениями для f(N) посещались первыми.

В качестве примера будет осуществлена эвристическая стратегия, для того, чтобы найти путь между двумя городами на карте, рис. 7.2.

Города будут представляться с помощью своих координат:

*xy("a", 2, 4). xy("b", 5, 6).*

Двухместные факты описывают односторонние дороги, которые соединяют города:

*op("a", "b" ). op("b", "m"). op("m", "f").*

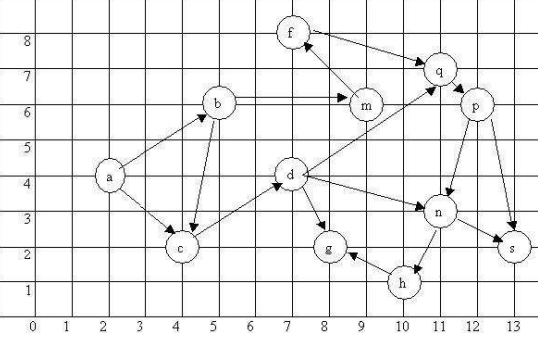


Рисунок 7.2 – Эвристическая стратегия

При эвристическом поиске очередь должна быть отсортирована так, чтобы наиболее многообещающие пути были поставлены вперёд. Данный алгоритм сортировки очень интересен. Он был изобретен Тони Хоаре.

Каждый путь представляется с помощью следующей конструкции:

*t(real, real, it),*

в которой первый аргумент хранит значение эвристической функции f для вершины, а второй аргумент хранит значение g(N). Алгоритм сортировки имеет предикат сравнения в качестве первого аргумента. Он передвигает самые многообещающие ветви в начало очереди.

Пример эвристического поиска:

Листинг 7.10

*implement main /\*heureka\*/*

*domains*

*item=r(string, string).*

*it= item\*.*

*node=t(real, real, it).*

*tree=node\*.*

*class facts*

*xy: (string, integer, integer).*

*op: (string, string).*

*class predicates*

*getXY:(string, integer, integer) determ (i, o, o).*

*cost: (string, string) -> real.*

*hn: (string) -> real.*

*not\_in\_circle: (string, it) determ (i,i).*

*theGoal: (string) procedure (o).*

*toDaughter: (node, node) nondeterm (i,o).*

*init:(string) procedure (o).*

*goalReached:(node) determ.*

*search: (tree, node) determ (i,o).*

*prtSolution: (node) procedure (i).*

*solve: () procedure.*

*cmp:(node, node) -> core::compareResult.*

*clauses*

*classInfo("main", "heureka-1.0").*

*cmp(t(A, \_, \_), t(B, \_, \_)) = core::greater() :- A > B, !.*

*cmp(t(A, \_, \_), t(B, \_, \_)) = core::equal() :- A=B, !.*

*cmp(\_, \_) = core::less().*

*op("a", "b" ). op("b", "m"). op("m", "f").*

*op("f", "q"). op("q", "p"). op("p", "n").*

*op("p", "s"). op("b", "c"). op("c", "d").*

*op("d", "q"). op("d", "n"). op("d", "g").*

*op("n", "h"). op("n", "s"). op("h", "g").*

*init("a").*

*goalReached(t(\_,\_,[r(\_,M)|\_])):- theGoal(R), R=M.*

*theGoal("s").*

*not\_in\_circle(Stt, Path):-not(list::isMember(r("", Stt), Path)).*

*xy("a", 2, 4). xy("b", 5, 6).*

*xy("c", 4, 2). xy("d", 7, 4).*

*xy("f", 7, 8). xy("g", 8, 2).*

*xy("h", 10, 1). xy("m", 9,6).*

*xy("n", 11, 3). xy("p", 12, 6).*

*xy("q", 11, 7). xy("s", 13, 2).*

*getXY(M, X, Y) :- xy(M, X, Y), !.*

*cost(No, NoFilho) = C:-*

*if getXY(No, XN, YN), getXY(NoFilho, XF, YF)*

*then*

*C = math::sqrt(((XN-XF)\*(XN-XF)) + ((YN - YF)\*(YN - YF)))*

*else*

*C= 0.0*

*end if.*

*hn(N) = HN :- theGoal(S),*

*if getXY(S, XS, YS), getXY(N, XN, YN)*

*then HN= math::sqrt(((XN-XS)\*(XN-XS)) + ((YN - YS)\*(YN - YS)))*

*else HN= 0.0 end if.*

*search([T|Queue],S):-*

*if goalReached(T) then S= T*

*else Extension= [E || toDaughter(T,E)],*

*NewQueue= list::append(Queue,Extension),*

*BestFirst= list::sortBy(cmp, NewQueue),*

*search(BestFirst, S)*

*end if.*

*toDaughter(t(\_F,G,[r(B,N)|Path]),t(F1,G1,*

*[r(Op, Child),r(B,N)|Path])):-*

*op(N, Child),*

*Op= string::format("%s to %s", N, Child),*

*not\_in\_circle(Child, Path),*

*G1 = G + cost(N, Child), F1 = G1 + hn(Child).*

*prtSolution( t(\_,\_,T)):-*

*foreach X= list::getMember\_nd(list::reverse(T)) do*

*stdio::write(X), stdio::nl*

*end foreach.*

*solve():- if init(E), search([t(hn(E),0,[r("root",E)])],S)*

*then prtSolution(S)*

*else stdio::write("No solution") end if.*

*run():- console::init(), solve().*

*end implement main /\*heureka\*/*

*goal mainExe::run(main::run).*

## **Теоретические исследования, оборудование, материалы и наглядные пособия**

Объектом исследования данной лабораторной работы являются деревья в среде Visual Prolog.

Для выполнения работы необходимы ПК и соответствующее программное обеспечение:

* MS WINDOWS;
* VisualProlog;
* MS Office (для оформления отчета).

## **Задание на работу (рабочее задание)**

* 1. Создайте предикат, удаляющий из двоичного справочника максимальный элемент левого поддерева.
  2. Создайте предикат, находящий максимальное из значений, находящихся в вершинах дерева.
  3. Создайте предикат, проверяющий, что дерево является двоичным справочником.
  4. Создайте предикат, переписывающий дерево в двоичный справочник.
  5. Создайте предикат, который будет находить среднеарифметическое значений, находящихся в вершинах дерева.
  6. Создайте предикат, который будет находить среднеарифметическое значений, находящихся в листьях дерева.
  7. Создайте предикат, находящий сумму положительных чисел, расположенных в вершинах дерева.
  8. Создайте предикат, находящий произведение отрицательных чисел, расположенных в вершинах дерева.
  9. Создайте предикат, подсчитывающий количество всех вершин данного дерева, значения которых принадлежат заданному диапазону.
  10. Создайте предикат, подсчитывающий количество всех вершин данного дерева заданной высоты.
  11. Создайте предикат, выводящий значения находящиеся в вершинах заданной высоты.
  12. Создайте предикат, проверяющий, является ли одно дерево поддеревом второго.
  13. Создайте предикат, выводящий дерево на экран, расположив дерево обычным образом: корень находится сверху, а листья снизу.
  14. Создайте предикат, выводящий дерево на экран, расположив дерево следующим образом: корень находится слева, а листья справа.
  15. Создайте предикат, удаляющий из двоичного справочника минимальный элемент левого поддерева.
  16. Создайте предикат поиска минимального из значений, находящихся в вершинах дерева.
  17. Создайте предикат, удаляющий из двоичного справочника минимальный элемент правого поддерева.
  18. Создайте предикат, удаляющий из двоичного справочника максимальный элемент правого поддерева.
  19. Создайте предикат, который будет находить среднегеометрическое значений, находящихся в вершинах дерева.
  20. Создайте предикат, который будет находить среднегеометрическое значений, находящихся в листьях дерева.
  21. Создайте предикат, находящий сумму отрицательных чисел, расположенных в вершинах дерева.
  22. Создайте предикат, находящий произведение четных чисел, расположенных в вершинах дерева.
  23. Создайте предикат, подсчитывающий количество всех вершин данного дерева, значения которых не принадлежат заданному диапазону.
  24. Создайте предикат, подсчитывающий количество всех вершин заданного диапазона.
  25. Создайте предикат, подсчитывающий количество всех вершин, не попадающих в заданный диапазон.
  26. Создайте предикат, находящий сумму нечетных чисел, расположенных в вершинах дерева.
  27. Создайте предикат, находящий произведение нечетных чисел, расположенных в вершинах дерева.

## **Ход работы (порядок выполнения работы)**

Для выполнения лабораторной работы необходимо:

* 1. Изучить теоретическую справку;
  2. Написать программу на зыке Пролог по индивидуальному заданию;
  3. Оформить отчет;
  4. Защитить работу преподавателю.

## **Содержание отчета**

В отчете должны присутствовать следующие пункты:

* + 1. Постановка задачи в терминах логики предикатов
    2. Текст программы на зыке Пролог по индивидуальному заданию.

# **Файлы**

## **Цель и задачи работы**

Получить навыки работы с внешними файлами в среде Visual Prolog.

## **Теоретические положения**

Класс «**file»** используется для чтения файла или записи в файл.

Часто требуется прочитать текст из файла, обработать его, а затем записать его обратно. Наилучший подход к решению данной задачи — прочитать весь файл в строку, а затем использовать мощный класс **string** для её обработки. Пролог имеет два предиката для работы такого типа:

Листинг 8.1

*readString : (string FileName, boolean IsUnicodeFile)*

*-> string String procedure (i,o).*

*writeString : (string Filename, string Source, boolean IsUnicodeFile)*

*procedure (i,i,i).*

Предикат **readString** считывает файл в строку.

Предикат **writeString**перезаписывает строку в файл.

Оба предиката имеют версии, не требующие указания, имеет ли файл формат Unicode.

Пример использования этих предикатов.

Листинг 8.2

*implement main*

*open core*

*constants*

*className = "main".*

*classVersion = "filetest".*

*clauses*

*classInfo(className, classVersion).*

*clauses*

*run():-*

*console::init(),*

*Str= file::readString("test.txt", Unicode),*

*stdio::write(Str),*

*S\_Upper= string::toUpperCase(Str),*

*file::writeString("upper.txt", S\_Upper, Unicode),*

*succeed().*

*end implement main*

*goal mainExe::run(main::run).*

В данном примере из файла считывается строка, затем ее символы преобразуются в символы верхнего регистра и новая строка записывается в другой файл.

В случае, если потребуется до записать в файл строку, то необходимо воспользоваться предикатом **appendString**.

Листинг 8.3

*appendString : (string Filename, string Source, boolean IsUnicodeFile)*

*procedure (i,i,i).*

## **Теоретические исследования, оборудование, материалы и наглядные пособия**

Объектом исследования данной лабораторной работы являются внешние файлы в среде Пролог.

Для выполнения работы необходимы ПК и соответствующее программное обеспечение:

* MS WINDOWS;
* VisualProlog;
* MS Office (для оформления отчета).

## **Задание на работу (рабочее задание)**

* + 1. Напишите предикат, который будет открывать файл на запись, если файл существует, и выводить соответствующее сообщение, если он отсутствует.
    2. Напишите предикат, который будет открывать файл на чтение и запись, если файл существует, и выводить соответствующее сообщение, если файл отсутствует.
    3. Напишите предикат, который будет открывать файл на дозапись, если файл существует, и выводить соответствующее сообщение, если он отсутствует.
    4. Создайте предикат, осуществляющий переписывание из одного файла, содержащего числа, в другой файл только тех чисел, которые являются четными.
    5. Создайте предикат, вычисляющий количество отрицательных чисел в файле.
    6. Создайте предикат, вычисляющий сумму чисел, хранящихся в файле.
    7. Создайте предикат, вычисляющий количество чисел, меньших среднего арифметического значения всех чисел в файле.
    8. Создайте предикат, формирующий из текста, хранящегося в файле, список слов, в которых имеются повторяющиеся символы.
    9. Создайте предикат, дополняющий все строки, хранящиеся в файле, символом "\*" до самой длинной строки.
    10. Создайте предикат, вычисляющий количество чисел в текстовом файле.
    11. Создайте предикат, вычисляющий количество слов в текстовом файле.
    12. Создайте предикат, осуществляющий переписывание из одного файла, содержащего текст, в другой файл только числа.
    13. Создайте предикат, формирующий из текста, хранящегося в файле, список слов, в которых нет повторяющихся символов.

## **Ход работы (порядок выполнения работы)**

Для выполнения лабораторной работы необходимо:

1. Изучить теоретическую справку;
2. Написать программу на зыке Пролог по индивидуальному заданию;
3. Оформить отчет;
4. Защитить работу преподавателю.

## **Содержание отчета**

В отчете должны присутствовать следующие пункты:

* + 1. Постановка задачи в терминах логики предикатов
    2. Текст программы на зыке Пролог по индивидуальному заданию.

# **Haskell: работа со списками**

## **Цель и задачи работы**

Получить навыки программирования на языке Haskell.

## **Теоретические положения**

Haskell — это язык программирования, то есть такой язык, на котором люди могут выражать то, как должны работать компьютеры. Это очень похоже на написание кулинарного рецепта: вы пишете рецепт, а компьютер готовит по нему.

Прежде всего вам нужен компилятор Haskell. Компилятор — это программа, которая берёт исходный текст и переводит его в машинный код: второй, более примитивный язык, который могут понимать только компьютеры (вы тоже можете, но чтение машинного языка — крайне сложное занятие). Другой способ объяснить это: компилятор Haskell берёт ваш рецепт и создаёт по нему программу.

В любом случае, для того, чтобы начать изучать Haskell, загрузите и установите [платформу Haskell](http://hackage.haskell.org/platform/). Она содержит «Glasgow Haskell Compiler» (или GHC), а также всё другое, что вам понадобится,

Рассмотрим GHCi, входящий в состав пакета GHC. Буква 'i' означает 'interactive' (англ. интерактивный), что и будет написано в заставке программы, когда вы её запустите. В командной строке (если вы работаете под Windows, кликните "Пуск", затем "Выполнить", наберите 'cmd' и нажмите клавишу Enter) наберите ghci и нажмите Enter.

На экране вы должны увидеть что-то похожее на это:

\_\_\_ \_\_\_ \_

/ \_ \ /\ /\/ \_\_(\_)

/ /\_\// /\_/ / / | | GHC Interactive, version 6.6, for Haskell 98.

/ /\_\\/ \_\_ / /\_\_\_| | http://www.haskell.org/ghc/

\\_\_\_\_/\/ /\_/\\_\_\_\_/|\_| Type :? for help.

Loading package base ... linking ... done.

Prelude>

Сначала идёт заставка GHCi, затем сообщение, что загружается пакет "base". Это означает, что вы сможете пользоваться встроенными модулями и функциями, которые идут вместе с GHC. И наконец, строка Prelude>, известная как "приглашение". Именно сюда надо вводить команды, а GHCi будет их вычислять.

Давайте попробуем выполнить простые арифметические операции:

Prelude> 2 + 2

4

Prelude> 5 \* 4 + 3

23

Prelude> 2 ^ 5

32

Операторы означают то же, что и в других языках: + - сложение, \* - умножение, и ^ - возведение в степень.

GHCi — очень мощная среда разработки. По ходу курса мы научимся загружать исходный код и вычислять различные его части.

Для длинных вычислений, а также для написания программ на Haskell, нам нужно отслеживать промежуточные результаты.

Промежуточные результаты хранятся в переменных. Например, рассмотрим следующее вычисление:

ghci> 3.1416 \* 5^2

78.53999999999999

Результат — площадь круга с радиусом 5. Это очень неудобно — каждый раз печатать или запоминать последовательность {\displaystyle \pi \approx 3.1416}. На самом деле, вся цель програмирования заключается в том, чтобы поручить машине бездумное повторение и механическое запоминание. Именно поэтому в Haskell определена переменная под названием pi, которая содержит в себе более дюжины цифр из записи числа pi:

ghci> pi

3.141592653589793

ghci> pi \* 5^2

78.53981633974483

Другими словами, как переменная pi, так и её значение 3.141592653589793 одинаково успешно могут быть использованы в вычисленииях, они взаимозаменяемы.

Создайте новый файл с именем Varfun.hs в вашем любимом текстовом редакторе (расширение «hs» означает «Haskell») и вставьте туда следующее определение:

r **=** 5.0

Убедитесь в отсутствии пробельных символов перед r, так как Haskell чувствителен к этому.

Теперь, перейдите в каталог с сохранённым файлом, запустите GHCi и воспользуйтесь командой :load (или, для краткости можно использовать :l):

Prelude> :load Varfun.hs

Compiling Main ( Varfun.hs, interpreted )

Ok, modules loaded: Main.

\*Main>

Загрузка исходного файла Haskell сделает доступными из командной строки GHCi все содержащиеся там определения.

Если GHCi выдаёт ошибку «Could not find module 'Varfun.hs'», возможно, вы забыли перейти в каталог с исходным файлом. Воспользуйтесь командой :cd для того, чтобы сменить текущий каталог на тот, где находится файл «Varfun.hs».

Prelude> :cd c:\myDirectory

Prelude> :load Varfun.hs

Compiling Main ( Varfun.hs, interpreted )

Ok, modules loaded: Main.

\*Main>

Теперь вы можете использовать в вычислениях недавно определённую переменную r.

\*Main> r

5

\*Main> pi \* r^2

78.53981633974483

Итак, чтобы вычислить площадь круга с радиусом 5, мы просто определяем r = 5.0 и затем вписываем известную формулу {\displaystyle \pi \*r^{2}} для вычисления площади круга. Теперь нет необходимости каждый раз писать числа, и это очень удобно!

Давайте добавим другое определение. Измените содержимое исходного файла на следующее:

r **=** 5

area **=** pi \* r ^ 2

Сохраните файл и выполните команду :reload в GHCi, чтобы загрузить туда новое содержимое файла. (Также вместо :reload, для краткости можно исполбзовать просто :r)

\*Main> :reload

Compiling Main ( Varfun.hs, interpreted )

Ok, modules loaded: Main.

\*Main>

Теперь нам доступны две переменные: r и area:

\*Main> area

78.53981633974483

\*Main> area / r

15.707963267948966

Если вы уже знакомы с императивными языками программирования (например, C или Python), вы уже могли заметить, что в Haskell переменные весьма отличаются от того, что вам о них известно. Теперь мы объясним суть и причины этих отличий.

В отличие от императивных языков, переменные в Haskell не изменяются. Однажды определённые, они никогда не изменяют своих значений: они неизменяемые (англ. Immutable).

Представьте, что у нас несколько окружностей с разными радиусами, и нужно вычислить их площадь. Например, чтобы рассчитать площадь круга с радиусом 3, мы объявим новые переменные r2 и area2, и запишем их в наш файл.

r **=** 5

area **=** pi\*r^2

r2 **=** 3

area2 **=** pi\*r2^2

Конечно, это не удобно, так как мы повторяем формулы, абсолютно идентичные друг другу. Гораздо лучше записать эту формулу один раз, а затем применять её для разных радиусов. Функции, как раз позволяют сделать так.

Функции определяют для конкретного значения аргумента (или параметра) конкретное результирующее значение. Функции в Хаскеле определяются просто и подобно определению переменных. Для того, чтобы определить функцию, слева от знака «=» мы пишем имя функции, затем через пробел локальную переменную (или несколько переменных)которая и будет использована в функции, а справа от знака «=» мы пишем выражение, определяющее результирующее значение. Например, для определения площади круга с известным радиусом определим функцию area, зависящую от параметра r

area r **=** pi \* r^2

Теперь мы можем использовать любые значения параметра, для вычисления площади нужного круга. Напишите это определение в своём файле и попробуйте следующее:

\*Main> area 5

78.53981633974483

\*Main> area 3

28.274333882308138

\*Main> area 17

907.9202768874502

Как вы видите, мы можем использовать разные радиусы для расчёта площади соответствующих кругов.

Круглые скобки «()» используются, чтобы сгруппировать значения. Так, например

area (5+3)

интерпретируется так: сначала сложить 5 и 3, затем вычислить функцию от получившейся суммы; в то время как

area 5 + 3

означает: вычислить значение функции от 5, а затем, к получившемуся результату прибавить 3.

У функции также может быть много параметров, записываются они также через пробел. Например, объявим функцию для вычисления площади прямоугольника:

areaRect a b **=** a \* b

И воспользуемся ею:

\*Main> areaRect 6 10

60

Очевидно, что можно использовать и большее количество переменных, для расчёта функции по какой-либо формуле, они также будут записываться через пробел.

Для задания функций также можно использовать уже заданные функции (обратите внимание на необходимость использования скобок во второй строчке, справа от знака «=»: double — функция одной переменной, поэтому отсутствие скобок выглядело бы как желание использовать doubleкак функцию двух переменных (одна из которых — также является функцией)):

double x **=** 2\*x

quadruple x **=** double (double x)

Иногда, писать одну и ту же формулу внутри одной функции не хочется, и при этом она может нигде больше не использоваться, а занимать хорошее, короткое имя не хочется, и писать длинное имя внутри функции каждый раз тоже. Для этого используются локальные переменные, которые будут доступны только внутри функции, в которой они объявлены. Для этой цели используется оператор where. Перед этим оператором и всеми локальными переменными внутри функции отбиваются четырьмя пробелами. Так компилятор отличает локальные переменные от глобальных. Например, для вычисления площади треугольника по формуле Герона:

heron a b c **=** sqrt (s\*(s-a)\*(s-b)\*(s-c))

**where**

s **=** (a+b+c) / 2

s — это половина периметра писать эту формулу каждый раз утомительно. Также нельзя определять s, как глобальную переменную,

heron a b c **=** sqrt (s\*(s-a)\*(s-b)\*(s-c)) *-- Это работать не будет.*

s **=** (a+b+c) / 2

это не будет работать, так как переменные a, b и c доступны только в правой части функции heron (то есть они локальны) и не будут определены для переменной s.

В следующем примере рассмотрено использование нескольких функций с локальными переменными:

areaTriangleTrig a b c **=** c \* height / 2 *-- use trigonometry*

**where**

cosa **=** (b^2 + c^2 - a^2) / (2\*b\*c)

sina **=** sqrt (1 - cosa^2)

height **=** b\*sina

areaTriangleHeron a b c **=** result *-- use Heron's formula*

**where**

result **=** sqrt (s\*(s-a)\*(s-b)\*(s-c))

s **=** (a+b+c)/2

Также, глобальные переменные не будут влиять на локальные, например:

area r **=** pi \* r^2

r **=** 0

Если вы воспользуетесь этой функцией, она не будет возвращать значение «0», потому что r внутри функции area, и вне её — это две разные, независимые переменные.

Однако, стоит заметить, что если внутри функции не определена одноимённая локальная переменная, то за неё принимается глобальная. Например:

z **=** 4

zat a **=** z\*a

Если набрать это в исходном файле, затем в GHCi ввести zat 2, то интерпретатор вернёт «8». Это также позволяет говорить о переменных, как о функциях от 0 переменных.

В Haskell используются две фундаментальные структуры, предназначенные для работы с множественными значениями: списки и кортежи. И те, и другие собирают несколько значений в одно.

Создадим несколько списков в GHCI:

ghci> let numbers = [1,2,3,4]

ghci> let truths = [True, False, False]

ghci> let strings = ["here", "are", "some", "strings"]

Элементы списка записываются в квадратных скобках и разделяются запятыми. Единственное ограничение - все элементы списка должны быть одного типа. Попытка определить список с элементами различных типов приводит к ошибке типов:

ghci> let mixed = [True, "bonjour"]

<interactive>:1:19:

Couldn't match `Bool' against `[Char]'

Expected type: Bool

Inferred type: [Char]

In the list element: "bonjour"

In the definition of `mixed': mixed = [True, "bonjour"]

Вдобавок к определению списка путём перечисления его элементов, вы можете построить список шаг за шагом при помощи оператора (:). Этот оператор называется "cons". Такое название он получил в наследство от LISP-программистов, назвавших этот оператор так от слова "constructor". Cons используется для того, чтобы присоединить к списку элемент:

ghci> let numbers = [1,2,3,4]

ghci> numbers

[1,2,3,4]

ghci> 0:numbers

[0,1,2,3,4]

Когда вы присоединяете элемент к списку (something:someList), вы получаете новый список. Таким образом вы можете присоединить сколько угодно элементов. Обратите внимание, что элемент добавляется слева.

ghci> 1:0:numbers

[1,0,1,2,3,4]

ghci> 2:1:0:numbers

[2,1,0,1,2,3,4]

ghci> 5:4:3:2:1:0:numbers

[5,4,3,2,1,0,1,2,3,4]

На самом деле, в Haskell все списки строятся путём присоединения элементов к пустому списку, []. Запись с квадратными скобками и запятыми - всего лишь синтаксический сахар. Так что [1,2,3,4,5] - абсолютно то же самое, что 1:2:3:4:5:[]

Однако нужно принять во внимание следующее: тогда как True:False:[] - рабочий код, True:False вовсе не рабочий:

ghci> True:False

<interactive>:1:5:

Couldn't match `[Bool]' against `Bool'

Expected type: [Bool]

Inferred type: Bool

In the second argument of `(:)', namely `False'

In the definition of `it': it = True : False

True:False приводит к знакомому сообщению об ошибке. в нём говорится, что оператор присоединения (:) (который на самом деле является всего лишь функцией) ожидает список в качестве второго аргумента, но вместо списка был получен Bool. (:) умеет присоединять значения к спискам, но не значения к значениям.

Cтроки в Haskell - всего лишь списки символов. Это означает, что со строками можно работать так же, как со списками. К примеру, вместо того, чтобы напрямую задать строку последовательностью символов в кавычках, мы также можем задать её с помощью последовательного присоединения символов, либо используя оператор (:) и завершая последовательность пустым списком, либо используя запись с квадратными скобками:

ghci>"hey" == ['h','e','y']

True

ghci>"hey" == 'h':'e':'y':[]

True

Кавычки - это тоже синтаксический сахар.

Списки могут содержать что угодно — если всё это что угодно одного и того же типа. Списки подходят под что угодно, так что они могут содержаться в списках.

Стоит отметить, что тип списка, к примеру, целых чисел отличен от типа целых чисел. То есть, Int и [Int] - это разные типы. Рассмотрим правила задания типа списка:

[Int] -- Список целых чисел

[Char] -- Список символов

[String] -- Список строк

То есть тип списка так и указывается, в квадратных скобках. Например, список списков строк имеет такой тип:

[[String]] -- Список списков строк

Модель очень проста:

[ [String] ]

│ Тип │

└ данных ┘

│ Тип │

│ списка │

└─ этих данных ─┘

Списки различных типов не могут быть присоединены друг к другу, но пустой список можно присоединить к любому списку. Например, []:[[1, 2], [1, 2, 3]] даст [[], [1, 2], [1, 2, 3]], а [1]:[[1, 2], [1, 2, 3]] даст [[1], [1, 2], [1, 2, 3]], но ['a']:[[1, 2], [1, 2, 3]] уже вызовет сообщение об ошибке.

Списки списков позволяют описать некоторые виды сложных структур данных (к примеру, матриц). Они также прекрасно демонстрируют мощь системы типов Haskell. Программисты постоянно путаются при работе со списками списков, и наличие ограничений системы типов помогает не допускать ошибок.

.

## **Теоретические исследования, оборудование, материалы и наглядные пособия**

Объектом исследования данной лабораторной работы является программирование простых функций в Haskell.

Для выполнения работы необходимы ПК и соответствующее программное обеспечение:

* MS WINDOWS;
* Haskell;
* MS Office (для оформления отчета).

## **Задание на работу (рабочее задание)**

1. Создайте функцию, заменяющую в исходном списке первое вхождение заданного значения другим.
2. Создайте функцию, заменяющую в исходном списке все вхождения заданного значения другим.
3. Создайте функцию, порождающую по заданному натуральному числу N список, состоящий из натуральных чисел от 1 до N (по возрастанию).
4. Создайте функцию, порождающую по заданному натуральному числу N список, состоящий из натуральных чисел от N до 1 (по убыванию).
5. Создайте функцию, порождающую по заданному натуральному числу N список, состоящий из N случайных натуральных чисел из промежутка от 1 до 100.
6. Создайте функцию, порождающую по заданным числам N, M, K список, состоящий из N случайных натуральных чисел из промежутка от M до K.
7. Создайте функцию, порождающую по заданным числам M, K список, состоящий из случайного количества случайных чисел из промежутка от M до K.
8. Создайте функцию, который увеличивает элементы исходного списка на величину K.
9. Создайте функцию, переводящую список цифр от 0 до 9 в список соответствующих им названий (строк).
10. Создайте функцию, переводящую список цифр от 0 до 9 в список соответствующих им римских чисел.
11. Создайте функцию, переводящую список римских чисел в список соответствующих им арабских чисел (диапазон от 1 до 20).
12. Создайте функцию, удваивающую значения элементов списка.
13. Создайте функцию, преобразующую список, элементами которого являются числа, в список, элементы которого неотрицательны.
14. Создайте функцию, преобразующую исходный список в список позиций отрицательных элементов.
15. Создайте функцию, удаляющую из исходного списка элементы с четными номерами.
16. Создайте функцию, который разделит исходный список из целых чисел на два списка: список положительных чисел и список отрицательных чисел.
17. Создайте функцию, разделяющую исходный список на два подсписка. В первый из них должны попасть элементы с нечетными номерами, во второй - элементы с четными номерами.
18. Создайте функцию, вычисляющую по списку и числу, подсписок исходного списка, начинающийся с элемента с указанным номером.
19. Создайте функцию, осуществляющую удаление указанного количества последних элементов исходного списка.
20. Создайте функцию, осуществляющую разделение исходного списка на два подсписка. В первый из них должно попасть указанное количество элементов из начала списка, во второй - оставшиеся элементы.
21. Создайте функцию, осуществляющую разделение исходного списка на два подсписка. В первый из них должно попасть указанное количество элементов с конца списка, во второй - оставшиеся элементы.
22. Создайте функцию, находящую предпоследний элемент списка.
23. Создайте функцию, удаляющую предпоследний элемент списка.
24. Создайте функцию, заменяющую в исходном списке два подряд идущих одинаковых элемента одним.
25. Создайте функцию, удаляющую в исходном списке все повторные вхождения элементов.
26. Создайте функцию, осуществляющую перестановку двух элементов списка с заданными номерами.
27. Создайте функцию, генерирующую все перестановки элементов списка, указанного в качестве первого аргумента функции.
28. Создайте функцию, осуществляющую циклический сдвиг элементов списка на один влево (вправо).
29. Создайте функцию, осуществляющую циклический сдвиг элементов списка на заданное количество шагов влево (вправо).
30. Создайте функцию, осуществляющую поэлементное перемножение соответствующих элементов двух исходных списков.

## **Ход работы (порядок выполнения работы)**

Для выполнения лабораторной работы необходимо:

1. Изучить теоретическую справку;
2. Написать программу на зыке Haskell по индивидуальному заданию;
3. Оформить отчет;
4. Защитить работу преподавателю.

## **Содержание отчета**

В отчете должны присутствовать следующие пункты:

1. Постановка задачи в терминах описания функций
2. Текст программы на зыке Haskell по индивидуальному заданию.

# **Haskell**

## **Цель и задачи работы**

Получить навыки программирования на языке Haskell.

## **Теоретические положения**

Мы начинаем писать настоящий код. А для этого нам понадобится окно во внешний мир. Откроем модуль app/Main.hs, найдём функцию main и напишем в ней следующее:

main :: IO ()

main = putStrLn ”Hi, real world!”

Стандартная функция putStrLn выводит строку на консоль. А если говорить строже, функция putStrLn применяется к значению типа String и делает так, чтобы мы увидели это значение в нашем терминале.

Выбирать внутри функции приходится очень часто. Существует несколько способов задания условной конструкции. Вот базовый вариант:

if CONDITION then EXPR1 else EXPR2

где CONDITION — логическое выражение, дающее ложь или истину, EXPR1 — выражение, используемое в случае True, EXPR2 — выражение, используемое в случае False.

Пример:

checkLocalhost :: String -> String

checkLocalhost ip =

-- True или False?

if ip == ”127.0.0.1” || ip == ”0.0.0.0”

-- Если True - идёт туда...

then ”It’s a localhost!”

-- А если False - сюда...

else ”No, it’s not a localhost.”

Функция checkLocalhost применяется к единственному аргументу типа String и возвращает другое значение типа String. В качестве аргумента выступает строкасодержащая IP-адрес, а функция проверяет, не лежит ли в ней localhost. Оператор || — стандартый оператор логического «ИЛИ», а оператор == — стандартный оператор проверки на равенство. Итак, если строка ip равна 127.0.0.1 или 0.0.0.0, значит в ней localhost, и мы возвращаем первое выражение, то есть строку It’s a localhost!, в противном случае возвращаем второе выражение, строку No, it’s not a localhost.

Круглые скобки включают выражение типа String по схеме:

main :: IO ()

main = putStrLn (checkLocalhost ”173.194.22.100”)

└─── выражение типа String ───┘

То есть функция putStrLn видит не применение функции checkLocalhost к строке, а просто выражение типа String. Если бы мы опустили скобки и написали так:

main :: IO ()

main = putStrLn checkLocalhost ”173.194.22.100”

произошла бы ошибка компиляции, и это вполне ожидаемо: функция putStrLn применяется к одному аргументу, а тут их получается два:

Вспомним о равенстве в определении:

checkLocalhost ip =

if ip == ”127.0.0.1” || ip == ”0.0.0.0”

then ”It’s a localhost!”

else ”No, it’s not a localhost.”

То, что слева от знака равенства, равно тому, что справа. А раз так, эти два кода эквивалентны:

main :: IO ()

main = putStrLn (checkLocalhost ”173.194.22.100”)

main :: IO ()

main =

putStrLn (if ”173.194.22.100” == ”127.0.0.1” ||

”173.194.22.100” == ”0.0.0.0”

then ”It’s a localhost!”

else ”No, it’s not a localhost.”)

Мы просто заменили применение функции checkLocalhost её внутренним выражением, подставив вместо аргумента ip конкретную строку 173.194.22.100. В итоге, в зависимости от истинности или ложности проверок на равенство, эта условная конструкция будет также заменена одним из двух выражений. В этом и заключается идея: возвращаемое функцией значение — это её последнее, итоговое выражение. То есть если выражение:

”173.194.22.100” == ”127.0.0.1” ||

”173.194.22.100” == ”0.0.0.0”

даст нам результат True, то мы переходим к выражению из логической ветви then.

Если же оно даст нам False — мы переходим к выражению из логической ветви else.

Это даёт нам право утверждать, что условная конструкция вида:

if True

then ”It’s a localhost!”

else ”No, it’s not a localhost.”

может быть заменена на первое нередуцируемое выражение, строку It’s a localhost!, а условную конструкцию вида:

if False

then ”It’s a localhost!”

else ”No, it’s not a localhost.”

можно спокойно заменить вторым нередуцируемым выражением, строкой No, it’s not a localhost.. Поэтому код:

main :: IO ()

main = putStrLn (checkLocalhost ”0.0.0.0”)

эквивалентен коду:

main :: IO ()

main = putStrLn ”It’s a localhost!”

Аналогично, код:

main :: IO ()

main = putStrLn (checkLocalhost ”173.194.22.100”)

есть ни что иное, как:

main :: IO ()

main = putStrLn ”No, it’s not a localhost.”

Каким бы сложным ни было логическое ветвление внутри функции checkLocalhost, в конечном итоге оно вернёт/вычислит какое-то одно итоговое выражение. Именно поэтому из функции в Haskell нельзя выйти в произвольном месте, как это принято в императивных языках, ведь она не является набором инструкций, она — выражение, состоящее из других выражений.

## **Теоретические исследования, оборудование, материалы и наглядные пособия**

Объектом исследования данной лабораторной работы является программирование простых функций в Haskell.

Для выполнения работы необходимы ПК и соответствующее программное обеспечение:

* MS WINDOWS;
* Haskell;
* MS Office (для оформления отчета).

## **Задание на работу (рабочее задание)**

1. Решить СЛАУ размерностью не выше 5 методом Гаусса.
2. Решить СЛАУ размерностью не выше 5 методом Крамера.
3. Найти производную функции, заданной таблично.
4. Найти интеграл функции  методом трапеции.
5. Найти интеграл функции  методом левых прямоугольников.
6. Найти интеграл функции  методом правых прямоугольников.
7. Найти интеграл функции  методом центральных прямоугольников.
8. Найти корень уравнения  методом дихотомии.
9. Найти корень уравнения  методом секущих.
10. Найти корень уравнения  методом Ньютона.
11. Найти определитель матрицы методом Гаусса.
12. Найти интеграл функции  методом трапеции.
13. Найти интеграл функции  методом левых прямоугольников.
14. Найти интеграл функции  методом правых прямоугольников.
15. Найти интеграл функции  методом центральных прямоугольников.
16. Найти корень уравнения  методом дихотомии.
17. Найти корень уравнения  методом секущих.
18. Найти корень уравнения  методом Ньютона.
19. Найти интеграл функции  методом трапеции.
20. Найти интеграл функции  методом левых прямоугольников.
21. Найти интеграл функции  методом правых прямоугольников.
22. Найти интеграл функции  методом центральных прямоугольников.
23. Найти корень уравнения  методом дихотомии.
24. Найти корень уравнения методом секущих.
25. Найти корень уравнения  методом Ньютона.

## **Ход работы (порядок выполнения работы)**

Для выполнения лабораторной работы необходимо:

1. Изучить теоретическую справку;
2. Написать программу на зыке Haskell по индивидуальному заданию;
3. Оформить отчет;
4. Защитить работу преподавателю.

## **Содержание отчета**

В отчете должны присутствовать следующие пункты:

1. Постановка задачи в терминах описания функций
2. Текст программы на зыке Haskell по индивидуальному заданию.

## **Список используемой литературы**

1. Адаменко А., Кучуков А. Логическое программирование и Visual Prolog.- СПб, 2003
2. Цуканова Н.И., Дмитриева Т.А. Логическое программирование на языке Visual Prolog. Учебное пособие для вузов. - М.: Горячая линия-Телеком, 2008. - 144 с.
3. Юхтенко В. Visual Prolog. Программирование приложений.
4. Patrick Blackburn, Johan Bos, Kristina Striegnitz: Learn Prolog Now! College Publications, 2006.
5. Марков В.Н. Современное логическое программирование на языке Visual Prolog 7.5: учебник. - СПб.: БХВ-Петербург, 2016. - 544 с.
6. Visual Prolog [Электронный ресурс]: Visual Prolog – электрон. текстовые дан. – WiKi. Visual Prolog, 2018. – режим доступа: <https://wikiru.visual-prolog.com/index.php?title=Заглавная_страница>