Министерство науки и высшего образования Российской Федерации
ФГБОУ ВО «Кубанский государственный технологический университет»

Кафедра информационных систем и программирования

Симоненко Е.А.

# Языки и системы программирования

Методические указания по выполнению лабораторных работ
для студентов всех форм обучения
направления бакалавриата 09.03.03 «Прикладная информатика».

Краснодар
2019

Составитель: ст. преподаватель Симоненко Евгений Анатольевич.

**Языки и системы программирования:** методические указания по выполнению лабораторных работ для студентов всех форм обучения направления 09.03.03 Прикладная информатика. / Сост. Е.А. Симоненко; Кубан. гос. технол. ун-т. Кафедра информационных систем и программирования. – Краснодар: КубГТУ, 2019. – 68 с. Режим доступа: http://moodle.kubstu.ru (по паролю).по паролю).

Методические указания составлены для студентов направления бакалавриата 09.03.03 – «Прикладная информатика» Кубанского государственного технологического университета. В пособии даются методические указания по выполнению лабораторных работ по дисциплине «Языки и системы программирования» в соответствии с утверждённой рабочей программной данной дисциплины учебного плана введённого с 1 сентября 2019 года.

Методические указания могут также использоваться для курсов функционального и логического программирования.

Библиогр.: 22 назв.

Рецензенты:

д-р. техн. наук, профессор каф. ИСП КубГТУ В.Н. Марков;

руководитель отдела телекоммуникаций Краснодарского регионального информационного центра сети «Консультант Плюс», канд. техн. наук. Н.Ф. Григорьев.

## Оглавление

## Введение

Пособие «Языки и системы программирования. Методические указания по выполнению лабораторных работ» / Симоненко Е.А. написано для студентов направления бакалавриата 09.03.03 – «Прикладная информатика» Кубанского государственного технологического университета в соответствии с утверждённой рабочей программной этой дисциплины учебного плана введённого в действие с 1 сентября 2019 года. Пособие вобрало в себя опыт преподавания автором курсов функционального и логического программирования и дополнено новыми темами, связанными с изучением основ теории языков программирования и широкого спектра систем программирования на этих языках.

В частности в данном пособии рассматриваются:

– Программируемый редактор GNU Emacs.

– Язык Haskell версии 98 с элементами из версии 2010. Стандартный Haskell поддерживается пакетами GHC, Hugs, YHC, NHC. Здесь используется компилятор GHC.

– Язык Prolog в его стандартной разновидности. Стандартный Пролог поддерживается пакетами SWI-Prolog, GNU Prolog, YAP (Yet Another Prolog), но не Visual Prolog. Кроме стандартизации и доступности стандартный Пролог больше подходит для обучения чем Visual Prolog в силу большей своей простоты.

Основной литературой, дополняющей лекции, являются книги Романа Душкина (см. раздел «Библиография»), а также «Основные концепции языков программирования» / Р.У. Себеста. Особое внимание следует обратить на книгу [Душкин: 14 эссе], так как она небольшого объёма, в ней рассматриваются многие основные вопросы функционального программирования и программирования на языке Haskell, более того, она распространяется в электронном виде бесплатно. Второй по важности является книга [Душкин: ФП]. В ней изложение материала более пространное и более математизированное, рекомендуется для углубленного изучения функционального программирования. Книги [Душкин: Справочник] и [Душкин: Практика работы] носят вспомогательный характер. В первой в сжатом виде рассматриваются синтаксис и стандартная библиотека языка Haskell. Вторая посвящена средствам разработки на языке Haskell.

В последние годы вышло ещё несколько книг по программированию на языке Haskell. Для начинающих изучать Haskell особенно будет интересна книга [Липовача]. После освоения азов Haskell можно приступить к чтению книг [Мена] и [Бёрд]. В книге [Марлоу] рассматриваются вопросы параллельного и конкурентного программирования на Haskell.

Полный список рекомендованной литературы на русском языке по программированию на Haskell представлен в разделе «Библиография».

В Сети есть прекрасный видеокурс с тестами и заданиями на русском языке: [Stepik: Haskell] и его продолжение [Stepik: Haskell2].

Основной литературой, дополняющей данные указания в части логического программирования, является книга Ивана Братко, переводившаяся на русский язык два раза: 1) Братко И. Программирование на языке Пролог для искусственного интеллекта: пер. с англ. – М.: Мир, 1990. – 560 с.: ил.; 2) Братко И. Алгоритмы искусственного интеллекта на языке PROLOG, 3-е издание.: пер. с англ. – М.: ИД «Вильямс», 2004. – 640 с.: ил. Главное отличие этих двух изданий заключается в том, что в советском издании на русский язык были переведены и примеры кода и программ (в целом для Пролога это нормальное явление).

Для успешного освоения этой дисциплины необходимо предварительное ознакомление студентами с дисциплинами «Математические основы информатики», «Основы математической логики» и «Программирование». В свою очередь эта дисциплина способствует более осмысленному изучению таких дисциплин учебного плана как «Программирование на языке Python», «Веб-технологии».

Порядок выполнения лабораторных работ:

1. Прочтите описание работы.
2. Внимательно ознакомьтесь с теоретическими сведениями. Воспользуйтесь также предлагаемым учебным пособием и рекомендованной литературой. Проработайте этот материал на практике.
3. Проверьте полноту знакомства с теоретическими сведениями и их понимание, ответив устно на контрольные вопросы.
4. Выполните задания, попутно фиксируя результаты и промежуточные шаги выполнения.
5. Оформите отчёт о выполнении лабораторной работы в электронной форме и предоставьте его преподавателю в формате PDF.

## Лабораторная работа №1. Инструментальное программное обеспечение

### Цель работы

Познакомиться с инструментальным программным обеспечением.

### Теоретические сведения

Инструментальное программное обеспечение, или система программирования, включает в себя:

- ассемблеры;
- трансляторы (компиляторы и интерпретаторы);
- компоновщики;
- препроцессоры;
- отладчики;
- текстовые редакторы;
- библиотеки подпрограмм;
- редакторы графического интерфейса;
- парсеры (разборщики);
- профилировщики;
- генераторы документации;
- средства автоматизации разработки программ;
- интегрированные среды разработки;
- SDK (пакеты для разработки программ);
- средства автоматизированного покрытия кода;
- средства анализа покрытия кода тестами;
- системы управления версиями;
- средства непрерывной интеграции;
- средства управления проектами;
- системы отслеживания ошибок.

Непосредственно к средствам разработки программ относятся из перечисленных:

- ассемблеры;
- трансляторы;

- компоновщики;
- препроцессоры;
- отладчики;
- библиотеки подпрограмм.

Не обходятся также обычно без текстового редактора для написания программ.

**Контрольные вопросы**

1. Что такое инструментальное программное обеспечение?
2. Перечислите виды инструментального ПО и объясните их назначение.
3. Какие виды относятся к разработке программ, а какие к проектированию и сопровождению?

**Задание**

Для выполнения задания используйте встроенные в операционную систему средства помощи, а также интерфейс пользователя компьютера.

1. Выясните, какое инструментальное программное обеспечение установлено на Вашем компьютере?
2. Выясните, средства для каких языков программирования установлены на Вашем компьютере?
3. Выясните, какие из установленных средств являются универсальными?

## Лабораторная работа №2. Основы работы с текстовым редактором GNU Emacs

### Цель работы

Научиться работать в среде программируемого текстового редактора GNU Emacs.

### Теоретические сведения

Текстовый редактор GNU Emacs имеет, по сравнению с традиционными редакторами, такими как Visual Studio Code, Atom, Notepad++, несколько иные концептуальные особенности. В частности, окно редактора разделяется на буферы. Буферы могут быть редактируемыми или только для чтения. В каждом окне редактора, называемом фреймом, присутствует эхо-область, через который происходит информирование пользователя. Также предоставляется возможность вводить команды для вызова различных встроенных или дополнительных функций программы в так называемом минибуфере. Внизу каждого буфера отображается строка режима с различной служебной информацией. Также может быть включён показ главного меню редактора, через которое можно получить доступ ко многим функциям редактора.

Ниже приведены основные команды GNU Emacs:

- Alt-x: переход в минибуфер для ввода команды.
- list-packages: список установленных пакетов, при запуске происходит обновление списка из Интернета.
- dired: навигатор по файловой системе.
- Alt-w: копирование выделенного фрагмента.
- Ctrl-y: вставка фрагмента.
- Ctrl-w: вырезание фрагмента.
- Ctrl-x-u: отмена последнего действия редактирования.
- Ctrl-x Ctrl-f: открытие существующего или создание нового файла.
- Ctrl-x Ctrl-s: сохранение изменений в файле.
- Ctrl-s: поиск в буфере.
- Ctrl-x-o: переключение на следующий видимый буфер.
- Ctrl-x-2: делит текущее окно на два по горизонтали.
- Ctrl-x-3: делит текущее окно на два по вертикали.
- Ctrl-x-b: показывает список буферов в минибуфере.

- Ctrl-x Ctrl-b: показывает список буферов в текущем окне.
- Ctrl-x-0: закрывает текущее окно.
- Ctrl-g: прекращает выполнение команды.
- Ctrl-Shift-Arrow: переключение между окнами. Arrow означает одну из четырёх клавиш со стрелками.

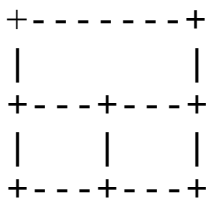
Более подробно возможности редактора GNU Emacs можно изучить по многочисленным руководствам и учебникам. На русском языке можно почитать здесь: http://alexott.net/ru/emacs/emacs-manual/

**Контрольные вопросы**

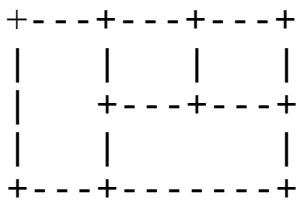
1. Какие отличительные черты GNU Emacs?
2. Как ввести команду редактора?
3. Как вывести список установленных пакетов?
4. Как обновить установленные пакеты?
5. Как установить требуемый пакет?
6. Как отобразить домашнюю директорию?
7. Как создать новый или открыть существующий файл?
8. Какие команды управления окнами GNU Emacs вы запомнили?

**Задание**

1. Выясните, как запустить GNU Emacs в Вашей операционной системе, и запустите его.
2. Ознакомьтесь с интерфейсом редакторы GNU Emacs. Идентифицируйте основные элементы окна редактора.
3. Отобразите список установленных пакетов.
4. Найдите пакет haskell-mode. Посмотрите информацию об этом пакете.
5. Отобразите содержимое домашнего каталога.
6. Откройте файл .profile.
7. Создайте новый файл.
8. Скопируйте содержимое .profile в буфер нового файла.
9. Сохраните этот файл.
10. Разбейте текущее окно по схеме:



11. Разбейте текущее окно по схеме:



## Лабораторная работа №3. Основы программирования на языке Emacs Lisp

### Цель работы

Познакомиться и освоить на практике язык программирования Emacs Lisp.

### Теоретические основы

Для экспериментов с Emacs Lisp удобно воспользоваться специальным буфером под названием ***scratch***. Для запуска написанного в нём кода нужно перейти на позицию за скобкой, ограничивающей Lisp-выражение, и вызвать команду Ctrl-x Ctrl-e (eval-last-sexp).

Основой Lips является список. При этом список может быть вычислен. Если нам нужен список из первых четырёх чисел, то мы напишем так:

```
'(1 2 3 4)
```

Однако, если мы запишем так:

```
(1 2 3 4)
```

то интерпретатор попытается вычислить этот список, а для этого он возьмёт первый элемент списка и будет рассматривать его как имя функции. Очевидно, что такой список вычислить нельзя, но можно вычислить вот такой:

```
(+ 1 2 3 4)
```

В результате получим число 10.

Создание переменной:

```
(set 'x 1)
```

Печать значения:

```
(print "Hello!")
```

Для вычисления последовательности выражений предназначена функция progn:

```
(progn
  (setq y 2)
  (print (* x y)))
```

Функция setq создаёт переменную с локальной областью видимости.

Собственные функции создаются с помощью defun:

```
(defun print-hello ()
  "Printing Hello!"
```

```
  (print "Hello!"))

(print-hello)
```

**Контрольные вопросы**

1. Что такое Emacs Lisp?
2. Охарактеризуйте Emacs Lisp.
3. Как запустить код на Emacs Lisp в среде GNU Emacs?
4. В какой форме записываются выражения на языке Emacs Lisp?
5. Как посмотреть результаты выполнения кода Emacs Lisp?
6. Как создать переменную?
7. Как создать переменную с локальной областью видимости?
8. Как создать функцию?

**Задание**

1. Напишите функцию, вычисляющую корни квадратного выражения.
2. Напишите функцию, считающую количество чётных чисел числового списка.
3. Напишите функцию, считающую количество чисел гетерогенного списка.

## Лабораторная работа №4. Программное управление редактором GNU Emacs

### Цель работы

Познакомиться с основами программного управления редактором GNU Emacs с использованием языка Emacs Lisp.

### Теоретические сведения

После освоения основ программирования на языке Emacs Lisp можно приступить к основам управления GNU Emacs с помощью этого языка.

Печать в минибуфере:

```
(message "Hey")
```

Отображаемые в минибуфере сообщения также сохраняются в буфере ***Messages***.

Для создания функции, которую можно вызывать как и другие команды GNU Emacs, нужно в её теле вызвать interactive:

```
(defun print-hello ()
  "Printing Hello!"
  (interactive)
  (message "Hello!"))
```

Чтобы узнать имя буфера, нужно вычислить функцию buffer-name; а чтобы узнать имя файла, связанного с буфером – buffer-file-name.

Больше информации о программном управлении GNU Emacs можно получить по ссылке: http://alexott.net/ru/emacs/elisp-intro/elisp-intro-ru_4.html

### Контрольные вопросы

1. Как выводить сообщения в минибуфер GNU Emacs?
2. Где можно увидеть все ранее выводившиеся сообщения в текущей сессии?

### Задание

1. Напишите программу, которая может вызываться из минибуфера, и которая выводит в эхо-область имя текущего буфера и имя файла, связанного с этим буфером.

# Лабораторная работа № 5. Средства разработки на языке Haskell

## Цель работы

Изучить основные элементы языка Haskell и научиться использовать средства разработки на этом языке.

## Теоретические основы

Язык программирования Haskell относится к функциональным языкам, которые, в свою очередь, относятся к декларативным языкам. В программах на этом языке описываются в первую очередь *функции*, а также *типы* и *классы*. Существует большое количество средств разработки на Haskell. Это компиляторы, среды разработки, текстовые редакторы с поддержкой языка Haskell, интерактивные интерпретаторы и отладчики.

### *Средства разработки на языке Haskell*

Основным пакетом для разработчиков на языке Haskell сегодня считается *Haskell Platform* (http://hackage.haskell.org/platform/).

Из других пакетов языка Haskell следует также назвать *Hugs* (http://www.haskell.org/hugs/), *NHC* (http://www.haskell.org/nhc98/) и *York Haskell Compiler* (http://www.haskell.org/haskellwiki/Yhc). Эти реализации также как и Haskell Platform являются свободно-распространяемыми, но менее развиты.

*Haskell Platform*

Данное пособие ориентируется на пакет *Haskell Platform*. Его основные характеристики:

- свободное распространение (лицензия GNU GPL);
- соответствие стандарту;
- богатая библиотека;
- развитый инструментарий, включающий в себя, в частности, компилятор, отладчик и командную строку для интерактивной работы;
- менеджер пакетов *Cabal*;
- основан на средствах разработки *Glasgow Haskell Compiler* (GHC).

Haskell Platform в среде Windows устанавливается как и большинство других программ для этой операционной системы с помощью простой программы установки. В среде Linux и BSD для установки Haskell и дополнительных библиотек лучше воспользоваться родным менеджером пакетов.

Для интерактивной работы с Haskell нужно запустить программу GHCi либо через главное меню Windows, либо зайдя в каталог bin каталога установки Haskell Platform, либо воспользоваться тем, что после установки появляется соответствующая ассоциация с файлами, у которых расширение имени *hs*. После запуска GHCi и загрузки программы можно будет увидеть консольное окно со следующим текстом:

```
GHCi, version 7.0.2: http://www.haskell.org/ghc/ :? for help
Loading package ghc-prim ... linking ... done
Loading package integer-gmp ... linking ... done
Loading package base ... linking ... done
Loading package ffi-1.0 ... linking ... done
[1 of 1] Compiling Main ( C:\Temp\HaskellExamples\hello.hs, intepreted )
Ok, modules loaded: Main.
*Main>
```

Для выполнения функции нужно ввести её имя и её параметры, после чего нажать Enter.

*GNU Emacs*

Для редактирования программы на языке Haskell можно воспользоваться любым текстовым редактором, но желательно таким, чтобы в нём была хотя бы синтаксическая подсветка кода программы. Для Windows подойдёт редактор Notepad++ или Komodo Edit, под Linux – Gedit, Vim и много других.

Одним из лучшего выбора является продвинутый редактор *GNU Emacs*. Это свободно-распространяемое ПО, и его можно загрузить по ссылке: http://ftp.gnu.org/gnu/emacs/windows/ . Для установки в среде Windows полученный архив нужно распаковать в подходящую папку. Для добавления GNU Emacs в меню «Пуск» можно запустить программу *addpm.exe* из каталога *bin* папки установки GNU Emacs. Редактор можно также запускать напрямую, вызывая программу *runemacs.exe* из того же каталога bin. В ОС Linux GNU Emacs устанавливается стандартным для используемого дистрибутива образом.

В базовой поставке GNU Emacs не поддерживает Haskell. Для получения этой поддержки в среде Windows (в Linux поддержку можно установить с помощью стандартного менеджера пакетов) необходимо загрузить пакет *Haskell Mode* по ссылке: http://projects.haskell.org/haskellmode-emacs/ , распаковать его в папку *Application Data\.emacs.d* домашнего каталога пользователя Windows, попутно переименовав распакованную папку в *haskell-mode*, и отредактировать в этой же папке файл init.el, добавив в него следующие строки:

```
(load "~/.emacs.d/haskell-mode/haskell-mode")
(load "~/.emacs.d/haskell-mode/haskell-site-file")
(add-to-list 'auto-mode-alist '("\\.hs\\'" . haskell-mode))
(add-to-list 'auto-mode-alist '("\\.lhs\\'" . literate-haskell-mode))
(add-hook 'haskell-mode-hook 'turn-on-haskell-doc-mode)
(add-hook 'haskell-mode-hook 'turn-on-haskell-indentation)
```

Возможно читатель захочет сделать дополнительные настройки:

```
(setq-default truncate-lines t)
(setq line-number-mode t)
(setq column-number-mode t)
(set-keyboard-coding-system 'utf-8)
(prefer-coding-system 'utf-8)
(setq-default indent-tabs-mode nil)
(setq default-tab-width 4)
```

Первая настройка отменяет заворачивание строк при выходе их за пределы окна редактора. Вторая и третья включают отображение номеров строк и колонок. Четвёртая и пятая включают поддержку кодировки UTF-8. Предпоследняя настраивает редактор на замену отступов, сделанных с помощью табуляции, на пробелы. Последняя устанавливает ширину табуляции. На снимке с экрана ниже (рис. 1) изображён GNU Emacs с программой hello.hs:

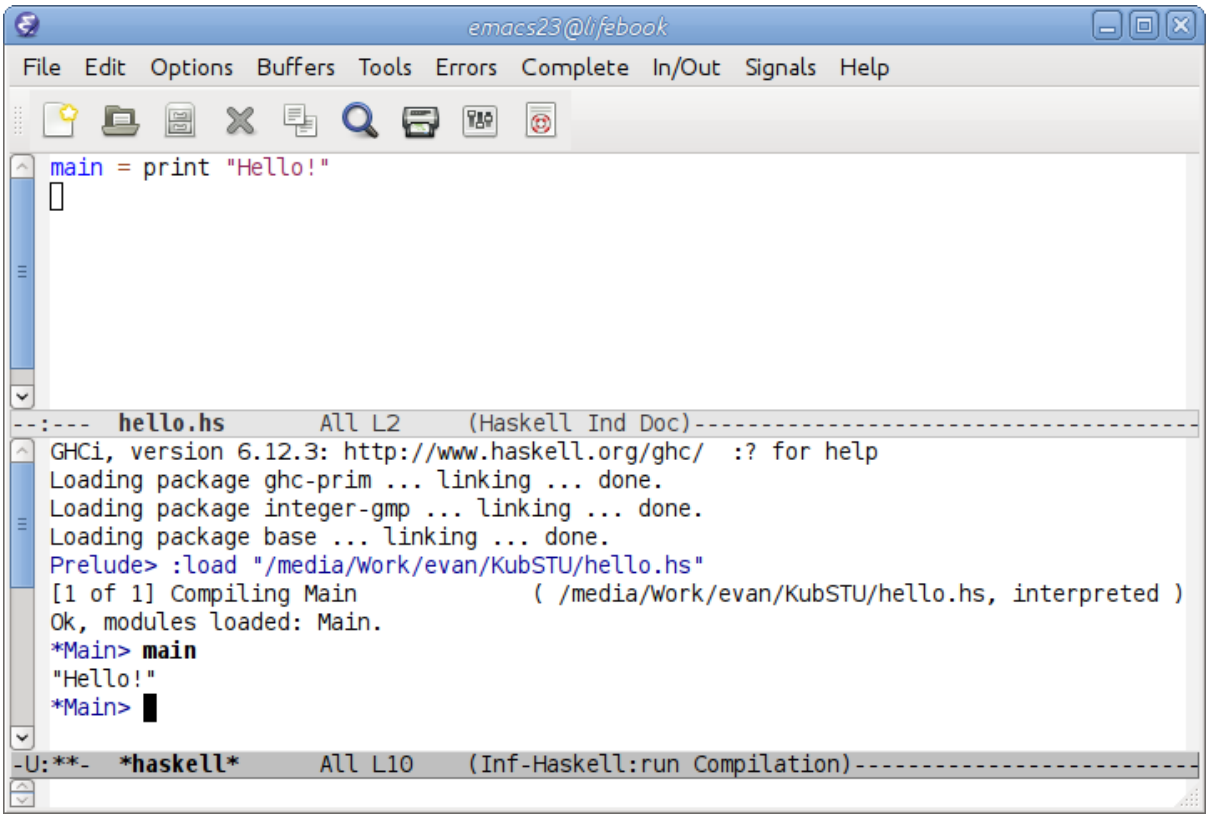



*Рисунок 1: GNU Emacs и Haskell*

В верхней половине окна (в так называемом *буфере*) находится редактируемая программа, во втором буфере запущен интерпретатор программ Haskell (GHCi), в котором загружен модуль *Main* из редактируемого файла hello.hs и выведен результат вычисления функции *main*.

Интерфейс собственно редактора достаточно стандартен.

Для запуска GHCi в среде GNU Emacs, находясь в окне редактирования буфера с программой на языке Haskell, нужно воспользоваться меню Haskell Mode *Haskell → Start interpreter* или сочетанием клавиш *Ctrl + C + B*, а для загрузки редактируемого модуля – *Haskell → Load file* или сочетанием клавиш *Ctrl + C + L*.

После загрузки модуля Вы должны увидеть сообщение вида:

```
[1 of 1] Compiling Main          ( /media/Work/evan/KubSTU/hello.hs,
interpreted )
Ok, modules loaded: Main.
```

Для запуска программы (функции) нужно перейти в окно интерпретатора и ввести имя функции:

```
*Main> main
```

В нашем примере в результате будет выведено сообщение “Hello”.

*Leksah*

Для программирования на языке Haskell также можно использовать свободно-распространяемую IDE Leksah (http://www.leksah.org/). Она написана на Haskell и для Haskell. Существует оригинальное руководство, с помощью которого можно освоить эту среду. Как минимум она, так же как и GNU Emacs, позволяет редактировать модули и запускать их в GHCi.

**Задание**

1. Освойте средства разработки для языка Haskell на примере программы «Hello, World!».
2. Изучите руководство и освойте на практике haskell-mode для GNU Emacs.[1]

1 Оригинальное руководство haskell-mode по ссылке: http://haskell.github.io/haskell-mode/manual/latest/

# Лабораторная работа № 6. Основы программирования на языке Haskell

## Цель работы

Изучить основы программирования на языке Haskell, включая структуру программы, функции, инструкции, потоковый ввод-вывод и рекурсию.

## Теоретические основы

Рассматриваются основы программирования на языке Haskell, включая структуру программы, функции, инструкции, потоковый ввод-вывод и рекурсию.

### *Синтаксис языка Haskell*

Программа на Haskell представляет из себя модуль *Main* с функцией *main* (точкой входа). Создавать модуль Main имеет смысл, если код будет компилироваться в исполняемую программу. В остальных случаях можно обойтись и без них.

Уже из примера “Hello, World!” можно увидеть синтаксис описания *функций*, который в общем виде выглядит так:

```
название = описание
```

Так как в Haskell применяется статическая типизация, то подобное описание полагается на механизм автоматического вывода типов языка Haskell.

Кроме того, функции, не объявленные в каком-либо модуле, считаются объявленными в модуле Main.

Полный код нашего примера должен был выглядеть так:

```
module Main where


main :: IO ()
main = print "Hello!"
```

Описание модуля начинается со слова *module*, далее идёт название модуля и слово *where*. После чего идёт описание функций, типов и классов. В нашем случае описывается функция main. Для объявления типа функции используется специальная конструкция:

```
название :: тип
```

Функция main должна иметь тип *IO ()*, который будет рассмотрен позже.

Символьные данные (строки) в языке Haskell должны заключаться в двойные кавычки «”». Одиночные кавычки «'» используются для обрамления отдельных символов.

Однострочный комментарий начинается с пары символов «тире» «--». Блочный комментарий обрамляется символами «{-» и «-}» (фигурная скобка и тире).

Для выполнения вычислений в программе на Haskell нужно использовать операцию присваивания значения переменной <- или объявлять функции.

```
X <- 10
Y = 5 * 2
Z = X + Y
```

Haskell поддерживает все основные операции: + (сложение), – (вычитание), * (умножение), ** (возведение в степень), / (деление), 'div' (целочисленное деление), 'mod' (остаток от деления), /= (проверка на неравенство), < (меньше), > (больше), <= (меньше или равно), >= (больше или равно). Есть также большое количество арифметических функций.

***Ввод-вывод***

В примере «Hello, World!» мы уже видели как выводить на консоль строковые данные. Для этого предназначена встроенная функция *print*. Эта же функция умеет выводить также числа и содержимое переменных.

```
print 10
let x = 10
print x
```

Для чтения данных с клавиатуры можно использовать функцию *getLine*.

```
let y x = x * x
x <- getLine
print $ y x
```

Символ $ позволяет избежать использования круглых скобок для задания порядка вычисления функций:

```
print (y x)
```

***Рекурсия***

*Рекурсия* в программах на языке Haskell является единственным механизмом для реализации повторения.

*Факториал*

Как известно, факториал числа *N* это произведение целых чисел от 1 до *N*:

$$n!=\prod_{i=1}^{n} i$$

Если расписать формулу факториала, то получим следующее:

$$n!=1\cdot 2\cdot ...\cdot (n-1)\cdot n$$

Очевидно, что

$$1!=1$$

Для числа 0 факториал определяется равным 1:

$$0!=1$$

Посмотрим ещё раз на определение факториала и перепишем формулу в таком виде:

$$n!=n\cdot (n-1)\cdot ...\cdot 2\cdot 1$$

Из этой формулы хорошо видно, что

$$n!=n\cdot (n-1)!$$

Таким образом мы получили рекурсивное определение факториала (*рекуррентную формулу*):

$$n!=\begin{cases} 1, n=0 \\ n\cdot (n-1),\ n>0 \end{cases}$$

Запишем это на Haskell:

```
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Применённый здесь вид рекурсии называется *нисходящей рекурсией*. Но вычисление факториала для больших *N* можно ускорить, если применить *восходящую рекурсию*. Смысл её в том, чтобы как и при итеративном методе вычисления факториала начать с наименьшего значения для факториала, то есть с 1, и увеличивая на каждом шаге на 1, дойти до исходного N.

```
factorial2 n = f n 0 1
f n n2 f2 = if n == n2 then f2 else f n (n2 + 1) (n2 + 1) * f2
```

Факториал для любых *N* мы будем вычислять с помощью вспомогательной функции *f*. Эта функция будет вычисляться до тех, пор пока *n2* не сравняется с *n*, при этом начальным значением для n2 будет 0, а для факториала – 1. При каждом вызове функции f n2 будет увеличиваться на 1, а значение факториала f2 – домножаться на это увеличивающееся n2. Можно сказать, что функция f вычисляет факториал итеративно.

На современном компьютере оба метода работают со скоростью на глаз неразличимой.

*Числа Фибоначчи*

Последовательность чисел Фибоначчи была исследована на Западе Леонардо Пизанским (Фибоначчи) в его труде «Liber Abaci» (1202). Фибоначчи рассматривал развитие идеализированной (биологически нереальной) популяции кроликов. Подробности читай по ссылке: http://ru.wikipedia.org/wiki/Числа_Фибоначчи .

Определение *числа Фибоначчи* для натурального $N$ следующее:

$$f(n)=\begin{cases} 1, n=1,2 \\ f(n-1)+f(n-1), n>2 \end{cases}$$

На языке Haskell это запишется так:

```
fib 1 = 1
fib 2 = 1
fib n = fib (n - 2) + fib (n - 1)
```

У этого метода есть существенный недостаток. Если внимательно посмотреть на код, то мы заметим, что одно и тоже число Фибоначчи считается многократно. И при N = 100 программа зависает на неопределённое время. Очевидно, что если бы наша программа запоминала предыдущие вычисленные числа Фибоначчи, то могла бы заработать значительно быстрее.

**Задание**

1. Напишите на языке Haskell программу, которая вычисляет сумму натуральных чисел от 1 до $n$: $s(n)=\sum_{i=1}^{n} i$ с помощью нисходящей и восходящей рекурсии.
2. Напишите на языке Haskell программу, вычисляющую биномиальные коэффициенты: $C_n^k=\frac{n!}{k!\cdot(n-k)!}$ с помощью нисходящей и восходящей рекурсии.
3. Напишите на языке Haskell программу, которая вычисляет корни квадратного уравнения по его коэффициентам. Модифицируйте эту программу так, чтобы в ней осуществлялась проверка на существование корней уравнения.

## Лабораторная работа № 7. Обработка списков на языке Haskell

### Цель работы

Изучить синтаксис списков в языке Haskell, операцию разделения списка на голову и хвост и основные операции над списками: вычисление длины списка, поиск элемента в списке, добавление элемента в начало и конец списка, вставка элемента в список, удаление элемента из списка.

### Теоретические основы

*Список* это одна из самых часто используемых структур данных. Для него характерен перебор составляющих его элементов от начала до конца.

В языке Haskell элементы списка перечисляются через запятую и заключаются в квадратные скобки “[” и “]”:

```
[1, 2, 4, 8, 16]
['A', 'B', 'C']
```

Список без элементов называется *пустым списком* и обозначается так:

```
[]
```

Единственной встроенной операцией над списками является операция выделения нескольких первых элементов списка по отдельности и всех остальных в другой список. Чаще всего встречается выделение *головы* списка и его *хвоста*. Обозначается эта операция символом двоеточие “:”:

```
(h : t]
(x : y : other)
```

Рассмотрим основные операции над списками.

#### *Длина списка*

Пример вычисления длины списка (количества элементов) на языке Haskell фундаментален, и, разобравшись с ним, можно будет легко разобраться и с остальными операциями над списками на этом языке.

Очевидно, что для того, чтобы узнать сколько элементов у списка, нужно их перебрать по одному. «Проблема» в том, что в Haskell нет циклов и нет операции для списков типа «для каждого» (foreach). Как известно, вместо циклов в программах на Haskell используется рекурсия. Как применить рекурсию к данной задаче? Во-первых, нужно задать правило окончания рекурсии. Так как для списков у нас есть только операция отделения головы от хвоста, то в процессе рекурсии список будет раз за разом сокращаться, и в итоге станет пустым. Очевидно, что длина пустого

списка, то есть его количество элементов, равна нулю. Обозначим правило для подсчёта количества элементов списка как *list_length* и запишем последний факт на Haskell:

```
list_length [] = 0
```

Во-вторых, исчерпывая исходный список, мы уменьшаем его длину на 1 каждый раз, значит длина исходного списка равна длине хвоста списка плюс 1. Это же правило мы применяем и для хвоста. Таким образом мы дойдём до пустого списка. После чего пойдёт раскрутка рекурсии в обратном направлении. Запишем это на Haskell:

```
list_length (_:xs) = 1 + list_length xs
```

Разберём как это работает на конкретном примере. Пусть дан список [1,2,3]. Он не пуст, и значит Haskell для него будет применять второе правило. Сначала произойдёт отделение головы этого списка (так как нам не важно её значение, то мы используем переменную «_»), и хвост *xs* получит значение [2,3], а длина будет равна длине хвоста плюс 1. Так как хвост не пуст, то действия повторятся теперь уже над ним, и мы получим последовательно списки [3] и []. На последнем списке, так как он пуст, сработает первое правило, и для него функция вернёт значение 0. После чего пойдёт раскрутка стека рекурсии, и длина станет равна предыдущему значению плюс 1, то есть 2, затем длина станет равна 2 плюс 1, то есть 3. Стек рекурсии опустошится и работа *list_length* закончится с ответом 3.

Схематически это можно представить так (в функциональном стиле):

```
length([1,2,3]) =
     length([2,3]) + 1 =
          (length([3]) + 1) + 1 =
               ((length([]) + 1) + 1) + 1 =
               ((0 + 1) + 1) + 1 =
          (1 + 1) + 1 =
     2 + 1 =
3
```

***Принадлежность элемента списку***

Проверка на принадлежность списку некоторого значения реализуется проще чем вычисление длины списка. Очевидно, что пустому списку не принадлежит ни один элемент. Запишем это:

```
is_element _ [] = False
```

Очевидно также, что мы легко можем определить совпадает ли заданное значение с головой списка. Если искомый элемент не совпал с

головой, то мы должны посмотреть нет ли его в голове хвоста, что на Haskell как обычно запишется рекурсивно:

```
is_element x (h:xs) = if x == h then True else is_element x xs
```

Здесь мы применили ветвление, так как Haskell не позволяет использовать в образцах входных параметров одинаковые обозначения.

Значения *True* и *False* определены для типа *Bool*.

### *Поиск позиции элемента в списке*

Смысл операции поиска позиции элемента в списке в том, чтобы найти номер элемента в списке совпавший с указанным значением. Алгоритм работы этой операции похож на алгоритм определения принадлежности списку указанного значения, только при переборе элементов списка мы ещё будем добавлять каждый раз добавлять 1 номеру элемента в списке. Пусть нумерация начинается с 0, а в случае, если элемента с таким значением в списке нет, будем генерировать ошибку с помощью функции *error*.

Мы уже выяснили, что в пустом списке мы не найдём ни одного элемента:

```
find_by_value _ [] = error "Not found!"
```

Если значение совпало со значением головы списка, то позиция равна 0. В противном случае будем увеличивать искомый номер позиции на 1. Запишем это на Haskell:

```
find_by_value x (h:xs) = if x == h then 0 else 1 + find_by_value x xs
```

В процессе исчерпывания списка мы либо придём к тому, что сработает второе правило, то есть элемент всё-таки найдётся, либо список исчерпается до пустого, и сработает первое правило, и мы получим в качестве результата ошибку. Подумайте над следующим вопросом: если искомых элементов в списке несколько, то сколько найдёт наша программа? Один? Если один, то какой по счёту? Или найдёт все? Что нужно изменить, что программа искала только один или все?

### *Сумма элементов числового списка*

Положим, что сумма элементов пустого списка равна 0:

```
list_sum [] = 0
```

А сумму элементов непустого списка можно вычислять рекурсивно:

```
list_sum (x:xs) = x + list_sum xs
```

### *Поиск наименьшего элемента в списке*

Имеет смысл рассматривать непустые списки. В списке из одного элемента наименьшим является этот единственный элемент, что на Haskell записывается так:

```
list_min [x] = x
```

Для списков из двух или более элементов функцию определим так:

```
list_min (x:xs) = min x m where m = list_min xs
```

В этой функции отделяется голова и ищется наименьший в хвосте списка, после чего выясняется, что является меньше – голова или наименьший из хвоста. Здесь применена конструкция *where*, предназначенная для создания локальных определений.

Поиск наибольшего элемента в списке оставляется для самостоятельного решения.

### *Добавление элементов в список*

Добавлять элементы можно в начало списка, в конец списка, а также в заданную своим номером позицию. В случае упорядоченных списков также можно добавлять элементы без нарушения упорядоченности.

Рассмотрим сначала добавление элемента в начало списка. Эта операция реализуется очень просто, благодаря тому, что мы имеем простой доступ к голове списка.

```
insert_first x l = (x:l)
```

Добавление в конец списка реализуется стандартным рекурсивным методом. Сначала рассматриваем простейший случай. Им является добавление элемента в конец пустого списка:

```
insert_last x [] = [x]
```

Если же список не пуст, то мы должны добраться до конца списка, рекурсивно его исчерпывая:

```
insert_last x (h:xs) = (h:insert_last x xs)
```

Добавление элемента в заданную позицию и в упорядоченный список оставляется для самостоятельного решения.

### *Удаление элемента из списка*

Так же как и с добавлением элементов удалять элементы можно из начала списка, из конца списка, а также из заданной позиции. В случае упорядоченных списков также можно удалять элементы без нарушения упорядоченности.

Рассмотрим здесь удаление элемента из указанной позиции. Как обычно легче всего работать с головой списка и с пустым списком:

```
remove_by_index _ [] = []
remove_by_index 0 (x:xs) = xs
```

Если удаляемый элемент не первый и список не пуст, то нужно удалять элемент из хвоста списка, при этом, естественно, номер позиции будет уменьшаться на 1.

```
remove_by_index i (x:xs) = (x:remove_by_index (i - 1) xs)
```

Остальные операции по удалению элементов из списка выносятся на самостоятельное решение.

**Задание**

Реализуйте на языке Haskell следующие операции над списками:

1. печать элементов списка;
2. получение значения элемента списка по его номеру в списке;
3. произведение элементов числового списка;
4. поиск наибольшего элемента в списке;
5. проверка на упорядоченность списка;
6. вставка заданного значения в указанную позицию списка;
7. вставка заданного значения в упорядоченный список;
8. замена головы и последнего элемента списка на указанное значение;
9. замена элемента с указанной позицией на заданное значение;
10. удаление элемента из списка с указанным значением.

## Лабораторная работа № 8. Разработка структур данных на языке Haskell

### Цель работы

Изучить реализацию на языке Haskell структур данных, в частности, списков, деревьев и графов.

### Теоретические основы

#### *Связный список*

Рассмотрим для начала простую структуру данных связный список. В Haskell уже присутствует встроенная реализация со специальным синтаксисом, однако, мы можем, использую стандартные средства Haskell, написать собственную реализацию списков, хотя и без специальной синтаксической поддержки.

Итак, сначала объявим подходящий тип данных:

```
data LinkedList a = Nil | Cons a (LinkedList a)
  deriving Show
```

Наш список может быть либо пустым, либо состоять из двух частей: первого элемента и хвоста, также являющегося списком.

Также здесь мы пользуемся возможностью GHC автоматические создавать экземпляры класса типов Show. Это потребуется для проверки в GHCi получаемых списков.

Несколько примеров списков:

```
let l1 = Nil
let l2 = Cons 1 Nil
let l3 = Cons 2 (Cons 1 Nil)
let l4 = Cons 3 l3
```

Напишем функцию, проверяющую список на пустоту (отсутствие элементов):

```
empty :: LinkedList a -> Bool
empty Nil = True
empty (Cons _ _) = False
```

Для добавления ещё одного элемента в список удобно иметь специально предназначенную для этого функцию:

```
insert :: LinkedList a -> a -> LinkedList a
insert l x = Cons x l
```

В данном случае функция добавления элемента в список получилась очень простой и почти бесполезной, но для более сложно устроенных структур это будет уже не так.

Длина списка:

```
listLength :: LinkedList a -> Int
listLength Nil = 0
listLength (Cons _ t) = listLength t + 1
```

Получение головы списка:

```
head :: LinkedList a -> Maybe a
head Nil = Nothing
head (Cons h _) = Just h
```

В отличие от стандартного списка из Prelude, наша реализация получения первого элемента списка возвращает тип Maybe, что позволяет писать код в чисто функциональном стиле, без обработки исключений.

Получение хвоста списка:

```
tail :: LinkedList a -> Maybe (LinkedList a)
tail Nil = Nothing
tail (Cons _ t) = Just t
```

Наша функция tail также отличается от стандартной тем, что возвращает тип Maybe.

***Бинарное дерево поиска***

Рассмотрим теперь пример более сложной структуры данных, бинарное дерево поиска. Определим соответствующий тип данных:

```
data BST a = BSTNil | BSTNode a (BST a) (BST a)
  deriving Show
```

Это определение похоже на определение для связного списка за тем различием, что связей теперь две. Посмотрим на несколько примеров таких деревьев:

```
let t1 = BSTNil
let t2 = BSTNode 4 BSTNil BSTNil
let t3 = BSTNode 4 (BSTNode 2 BSTNil (BSTNode 3 BSTNil BSTNil)) (BSTNode 6
BSTNil BSTNil)
```

(К сожалению, данный подход никак не ограничивает нас в создании некорректных деревьев.)

Проверка дерева на пустоту элементарна:

```
empty :: BST a -> Bool
empty BSTNil = True
empty _ = False
```

Если у нас есть готовое дерево, то можно эффективно искать хранящиеся в нём элементы:

```
contains :: (Ord a, Eq a) => BST a -> a -> Bool

contains BSTNil _ = False
```

```
contains (BSTNode x l r) v =
  v == x || (v < x && contains l v) || (v > x && contains r v)
```

Обратите внимание, что эта функция налагает на тип значений ограничения: Ord, так как функция сравнивает значения на больше-меньше, и Eq, так как проверяет равенство.

Реализуем вставку значения в дерево:

```
insertNode :: (Ord a, Eq a) => BST a -> a -> BST a
insertNode BSTNil v = BSTNode v BSTNil BSTNil
insertNode t@(BSTNode x l r) v
  | v == x = t
  | v < x = BSTNode x (insertNode l v) r
  | v > x = BSTNode x l (insertNode r v)
```

Вставка значения происходит в соответствии с принципом хранения значений в бинарном дереве поиска: меньшие значения помещаются в левые ветви, а большие в правые; если значение уже есть в дереве, то процесс вставки завершается.

Удобно формировать дерево не вручную, а по списку значений. Простейший вариант такой функции ниже:

```
fromList :: (Ord a, Eq a) => [a] -> BST a
fromList [] = BSTNil
fromList xs = fromList' BSTNil xs
  where
    fromList' t [] = t
    fromList' t (x:xs') = fromList' (insertNode t x) xs'
```

Эта функция берёт значения из исходного списка по очереди и добавляет их в формируемое дерево.

Посмотрим теперь, как можно удалять значения из дерева. Так как дерево должно оставаться бинарным деревом поиска, то реализация не вполне тривиальна:

```
deleteNode :: (Ord a, Eq a) => BST a -> a -> BST a
deleteNode BSTNil _ = BSTNil
deleteNode t@(BSTNode x BSTNil BSTNil) v =
  if v == x then BSTNil else t
deleteNode (BSTNode x l r) v
  | v == x = let mv = minValue r in (BSTNode mv l (deleteNode r mv))
  | v < x = BSTNode x (deleteNode l v) r
  | v > x = BSTNode x l (deleteNode r v)
```

Здесь используется функция поиска минимального значения в дереве:

```
minValue :: (Ord a, Eq a) => BST a -> a
minValue BSTNil = error "The minimum value cannot be calculated for an empty tree."
minValue (BSTNode x BSTNil _) = x
minValue (BSTNode _ l _) = minValue l
```

Обратите внимание, что для пустого дерева эта функция генерирует исключение.

### *Граф*

Стандартная библиотека Haskell уже содержит реализацию для графов Data.Graph. Однако, эта реализация весьма ограничена по своим возможностям и эффективности. Для начала рекомендуется ознакомиться с этой реализацией, после чего подумать над написанием собственной.

### Контрольные вопросы

1. Для чего нужен класс типов Show?
2. Возможно ли формирование некоторых структур данных вручную?
3. Почему в предложенной реализации списка некоторые функции возвращают тип Maybe?

### Задание

1. Изучите описанную в теоретических сведениях реализацию связных списков и деревьев и протестируйте приведённый код этой реализации.
2. Добавьте к реализации LinkedList следующие операции:
    1. вычисление суммы всех элементов списка;
    2. поиск наименьшего и наибольшего значения в списке;
    3. удаление заданного значения из списка;
    4. удаление дубликатов в списке.
3. Добавьте к реализации BST следующие операции:
    1. определение высоты дерева;
    2. вычисление суммы всех узлов дерева, если тип хранимых значений целочисленный;
    3. вычисление количества листьев дерева;
    4. вычисление количества промежуточных узлов дерева;

5. вычисление наибольшего значения в дереве (в отличие от minValue эта функция должная возвращать тип Maybe a).

4. [*] Разработайте реализацию списка упорядоченных значений. Учтите, что добавление и удаление элементов, а также слияние таких списков сохраняет упорядоченность его значений.
5. Изучите стандартную реализацию графов в языке Haskell Data.Graph.
6. [*] Разработайте реализацию структуры данных на Haskell для графов. Предусмотрите основные операции по работе с графом (добавление и удаление вершин, добавление и удаление рёбер) и вычисление основных характеристик графа (как-то: количество вершин и рёбер, поиск висячих вершин, вычисление степеней вершин, вычисление расстояний между вершинами и т. п.).

## Лабораторная работа № 9. Разработка алгоритмов на языке Haskell

### Цель работы

Изучить разработку алгоритмов на языке Haskell на примере операций над списками: объединение списков, разбиение списка, обращение списка, циклический сдвиг элементов списка, перестановка элементов списка, обмен элементов списка, сортировка списков.

### Теоретические основы

#### *Объединение двух списков*

Под объединением двух списков будем понимать список, состоящий из всех элементов сначала первого списка, затем – второго. Реализацию операции объединения удобно разбить на два определения, первое для объединения пустого списка с любым другим, второе – для произвольных списков:

```
list_concat [] l = l
list_concat (x:xs) l = (x:list_concat xs l)
```

Второе правило основано на том, что нам легко отделять голову списка, а затем назад её присоединять. Фактически здесь мы разбиваем первый список на отдельные элементы, а затем добавляем их, начиная с последнего, в начало второго:

```
list_concat( [1,2,3], [4,5] ) =
     [1 | list_concat( [2,3], [4,5] )] =
          [1 | [2 | list_concat( [3], [4,5] )]] =
               [1 | [2 | [3 | list_concat( [], [4,5] )]]] =
               [1 | [2 | [3 | [4,5]]]] =
          [1 | [2 | [3,4,5]]] =
     [1 | [2,3,4,5]] =
[1,2,3,4,5]
```

#### *Разбиение списка на два*

Рассмотрим здесь разбиение списка по первому найденному элементу с заданным значением. Например, список [1,2,3,4] по элементу со значением 2 будет разбит на списки [1] и [3,4].

Проще всего разбивать пустые списки, так как ответом всегда будет два пустых списка. Также просто разбивать список, голова которого совпадает с заданным значением, тогда ответом будет пустой список и хвост

исходного списка. Если же заданное значение не совпало с головой списка, то отложим её в сторону и попытаемся разбить по этому значению хвост исходного списка. После разбиения хвоста добавим голову в начало первого результирующего списка:

```
list_split [] _ = ([],[])
list_split (x:xs) v
  | v == x = ([],xs)
  | otherwise = let (l,r) = list_split xs v in ((x:l),r)
```

***Циклический сдвиг элементов списка***

Рассмотрим сначала циклический сдвиг списка влево на один элемент. Например, для списка [1,2,3] ответом будет [2,3,1]. То есть голова списка удаляется и добавляется в конец исходного списка. Записывается это очень просто:

```
list_shift_1 [] = []
list_shift_1 (x:xs) = insert_last x xs
```

Теперь рассмотрим сдвиг списка влево на большее количество позиций. Очевидно, что сдвиг на N позиций можно свести к N сдвигам на одну позицию. Так как итерации нам недоступны, то воспользуемся рекурсией, будем сдвигать список на одни элемент, уменьшать количество сдвигов и вновь вызывать сдвиг на один элемент:

```
list_shift i l = list_shift (i-1) (list_shift_1 l)
```

К несчастью одного определения как обычно недостаточно. Очевидно, что отсутствует определение для последнего сдвига, то есть когда передаваемое количество сдвигов сократится до нуля. Добавим это определение:

```
list_shift 0 l = l
```

***Обращение списка***

Обращение списка (reverse) это изменение порядка его элементов на противоположный, например, для списка [1,2,3] обратным будет [3,2,1]. Проще всего обращать пустой список и список, состоящий из одного элемента. Запишем это:

```
list_reverse [] = []
list_reverse [x] = [x]
```

Для более длинных списков будем поступать следующим образом: последовательно отделяем голову и добавляем её в конец:

```
list_reverse (x:xs) = insert_last x (list_reverse xs)
```

Можно предложить более быструю реализацию с использованием параметра-аккумулятора, который будет накапливать отделяемые от списков головы:

```
list_reverse' xs = list_reverse'' [] xs
  where
    list_reverse'' acc [] = acc
    list_reverse'' acc (x:xs) = list_reverse'' (x:acc) xs
```

Это определение заменяет основное определение на определение с аккумулятором. Аккумулятор получает в качестве начального значения пустой список. Функция с аккумулятором выполняет основную работу: отделяет голову списка и ставит её в начало списка-аккумулятора. Останов этого процесса происходит, когда исходный список опустошается. Посмотрим на коротком примере как это работает:

[1,2,3] : [] => [2,3] : [1 | []] => [3] : [2 | [1]] => [] : [3 | [2,1]] => R := [3,2,1]

Сортировка списка это его упорядочивание, то есть перестановка его элементов так, чтобы они удовлетворяли попарно отношению упорядоченности.

Рассмотрим три разных алгоритма сортировки: пузырьковую сортировку, быструю сортировку и метод вставки.

***Пузырьковая сортировка***

Алгоритм пузырьковой сортировки основан на многократной перестановке элементов списка пока они не станут попарно удовлетворять отношению упорядоченности. Например:

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

Основную часть алгоритма пузырьковой сортировки реализуем в функции *swap*, которая обменивает значениями два соседних элемента списка, если они удовлетворяют отношению *p*:

```
swap :: (Ord a) => [a] -> (a -> a -> Bool) -> [a]
swap [] _ = []
swap [x] _ = [x]
swap l@[x,y] p = if p x y then [y,x] else l
swap (x:xs@(y:t)) p = if p x y then (y:swap (x:t) p) else (x:swap xs p)
```

Как обычно проще всего оперировать первыми элементами списка. Функция *swap* производит перестановку первых двух элементов, если отношение *p* истинно. Если первые два элемента переставлять не надо, то *swap* вызывается для хвоста исходного списка. То есть *swap* ищет в списке два соседних элемента, которые нужно переставить.

В этих двух правилах используются так называемые именованные образцы. Для присвоения имени образцу нужно поместить его перед образцом, соединив с ним символом @.

Для того, чтобы список стал отсортированным необходимо вызывать *swap* многократно, по крайней мере до тех пор пока список не станет отсортированным. Для этого напишем функцию *bublesort*:

```
bubble_sort xs p = bubble_sort' xs (length xs) p where
  bubble_sort' xs i p
    | i == 0 = xs
    | otherwise = bubble_sort' (swap xs p) (i - 1) p
```

### *Быстрая сортировка*

Алгоритм быстрой сортировки основан на разделении исходного списка на два. В первый из списков попадают все элементы исходного списка, которые меньше первого, во второй — все, которые больше его. Такому разделению подвергаются и два полученный списка. Делается это до тех пор, пока списки не станут пустыми. После чего осуществляется обратный процесс — процесс объединения. Например:

[3,1,2,5,4] →

[1,2] – 3 – [5,4] →

([] – 1 – [2]) – 3 – ([4] – 5 – []) →

([] – 1 – ([] – 2 – [])) – 3 – (([] – 4 – []) – 5 – []) →

([] – 1 – [2]) – 3 – ([4] – 5 – []) →

[1,2] – 3 – [4,5] →

[1,2,3,4,5]

Собственно функция *quicksort* для выполнения быстрой сортировки:

```
quick_sort [] p = []
quick_sort (x:xs) p = quick_sort [y | y <-xs, p x y] p
 ++ [x]
 ++ quick_sort [y | y <- xs, not $ p x y] p
```

### *Метод вставки*

Метод вставки работает следующим образом: сначала исходных список редуцируется до пустого посредством поочерёдного отбрасывания его элементов, начиная с первого. После чего происходит восстановление списка, но уже со вставкой ранее отброшенного элемента в такую позицию, чтобы получался отсортированный список. Например:

[3,1,2,5,4] →

[3 | [1,2,5,4]] →

[3 | [1 | [2,5,4]]] →

[3 | [1 | [2 | [5,4]]]] →

[3 | [1 | [2 | [5 | [4]]]]] →

[3 | [1 | [2 | [5 | [4 | []]]]]] →

[3 | [1 | [2 | [5 | [4]]]]] →

[3 | [1 | [2 | [4,5]]]] →

[3 | [1 | [2,4,5]]] →

[3 | [1,2,4,5]] →

[1,2,3,4,5]

Определим сначала функцию для операции вставки элемента с сохранением порядка:

```
insert_ord x [] = [x]
insert_ord x l@(y:ys) = if gt x y then (y:insert_sort x ys) else (x:l)
```

Затем собственно метод вставки:

```
insertsort [] = []
insertsort (x:xs) = insert_ord x $  insertsort xs
```

**Контрольные вопросы**

1. Перечислите основные подходы к реализации алгоритмов на языке Haskell.
2. Как помогает реализация с аккумулятором?
3. Как реализовать итерации по счётчику?

**Задание**

1. Реализуйте на языке Haskell следующие операции над списками:
    1. разбиение списка на два по элементу с указанным индексом;
    2. циклический сдвиг элементов списка вправо;
    3. перестановка элементов списка;
    4. обмен значениями двух элементов списка с указанными индексами;
    5. слияние двух упорядоченных списков;
    6. проверка является ли один список частью другого списка.
2. [*] Напишите реализацию quick_sort без использования генератора списков и с распределением элементов на два списка за один проход.

## Лабораторная работа № 10. Средства разработки на языке Prolog

### Цель работы

Изучить основные элементы стандартного Prolog и научиться использоваться средства разработки на языке Prolog с использованием SWI-Prolog.

### Теоретические основы

Язык программирования Пролог относится к декларативным языкам. В программах на этом языке описываются объекты и отношения между ними. Программа на Прологе состоит из предложений (*предикатов*), которые описывают *факты* и *правила*, а также *вопросы*, то есть то, ради чего, собственно, и пишется программа на языке Пролог. Программист на Прологе не пишет как искать ответы на вопросы, этим занимается система Пролог, которая содержит механизм поиска ответа на вопросы.

#### *Средства разработки на языке Пролог*

Данное пособие ориентируется на пакет SWI-Prolog (http://www.swi-prolog.org/). Его основные характеристики:

- свободное распространение (лицензия GNU GPL);
- соответствие стандарту;
- богатая библиотека;
- развитый инструментарий, включающий в себя, в частности, компилятор, отладчик и командную строку для интерактивной работы;
- также имеется удобный текстовый редактор с наглядной подсветкой.

Из других пакетов языка Пролог следует также назвать GNU Prolog (http://www.gprolog.org/) и Yet Another Prolog (http://www.dcc.fc.up.pt/~vsc/Yap/). Эти реализации также являются свободно-распространяемыми, но менее развиты, чем SWI-Prolog.

SWI-Prolog в среде Windows устанавливается как и большинство других программ для этой операционной системы с помощью простой программы установки. Если SWI-Prolog устанавливается пользователем без прав администратора, то необходимо программе установки указать каталог, в который данный пользователь может записывать. Желательно также задать расширение файлов программ Пролог .pro вместо .pl, чтобы не было возможного конфликта с Perl. В дальнейшем выбранное расширение файлов будет иметь значение при работе со встроенным текстовым редактором. Для установки SWI-Prolog может потребоваться библиотека Visual Studio 2008 SP1 Redistributable, которую можно получить с сайта

Microsoft по ссылке: http://www.microsoft.com/downloads/ru-ru/details.aspx?FamilyID=A5C84275-3B97-4AB7-A40D-3802B2AF5FC2 .

В Ubuntu установить SWI-Prolog можно из репозитория пакетов:

```
sudo apt install swi-prolog
```

Основным средством разработки на SWI-Prolog является программа swipl. Через неё можно:

- запустить интерактивную консольную оболочку (REPL, Read-Eval-Print Loop);
- скомпилировать программу;
- запустить программу;
- вызвать редактор для изменения загруженной программы.

Рассмотрим эти возможности по порядку.

Если swipl вызвать непосредственно из командной строки, то она запустит интерактивный режим REPL, в котором можно построчно вводить программу и запускать её на выполнение. Этот режим очень полезен как в начале знакомства с Пролог, так и в процессе написания программ, когда требуется выяснить, как работает та или иная возможность и провести с ней эксперимент. Окно командной строки с REPL в Linux выглядит примерно вот так:

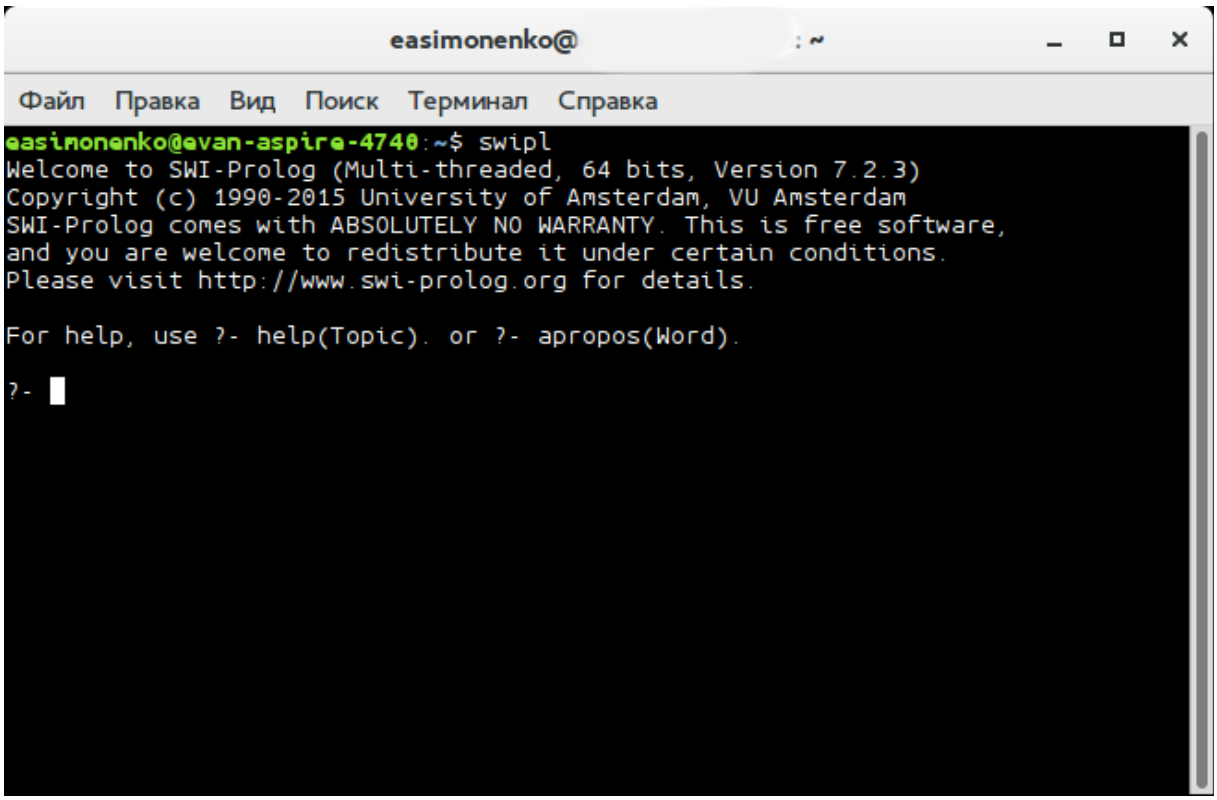



*Рисунок 2: SWI-Prolog REPL в Linux*

Для просмотра опций командной строки нужно задать опцию --help:

```
swipl --help
```

Для компиляции программы на Пролог задайте опцию -c:

```
swipl -c hello.pro
```

В результате будет получен исполняемый файл с именем a.out. Чаще всего нужно другое имя, которое можно задать опцией -o:

```
swipl -o hello -c hello.pro
```

Для запуска программы на Пролог без предварительной компиляции нужно воспользоваться опциями -t и -s:

```
swipl -t run -s hello.pro
```

Запуск REPL с загрузкой программы можно сделать так:

```
swipl -s hello.pro
```

Для запуска редактора для загруженной программы нужно вызвать предикат edit:

```
edit.
```

После небольшой паузы появится окно редактора кода на SWI-Prolog. Как выглядит окно редактора можно увидеть на рисунке 3.

В Windows для работы со SWI-Prolog можно запустить программу swipl-win.exe либо через главное меню Windows, либо зайдя в каталог bin каталога установки SWI-Prolog. Главное окно SWI-Prolog выглядит приблизительно так как изображено на снимке экрана ниже (рис.1).

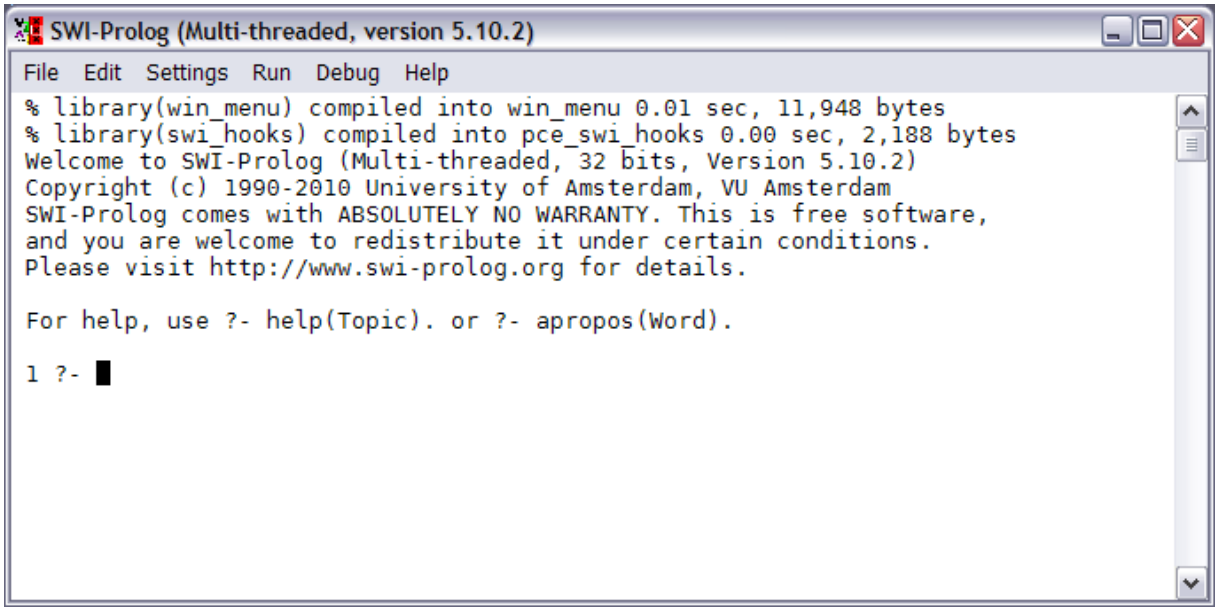



*Рисунок 3: Главное окно SWI-Prolog*

Для редактирования новой программы следует воспользоваться меню File → New. После ввода имени нового файла появится окно редактора. Введём в нём текст программы “Hello, World!”:

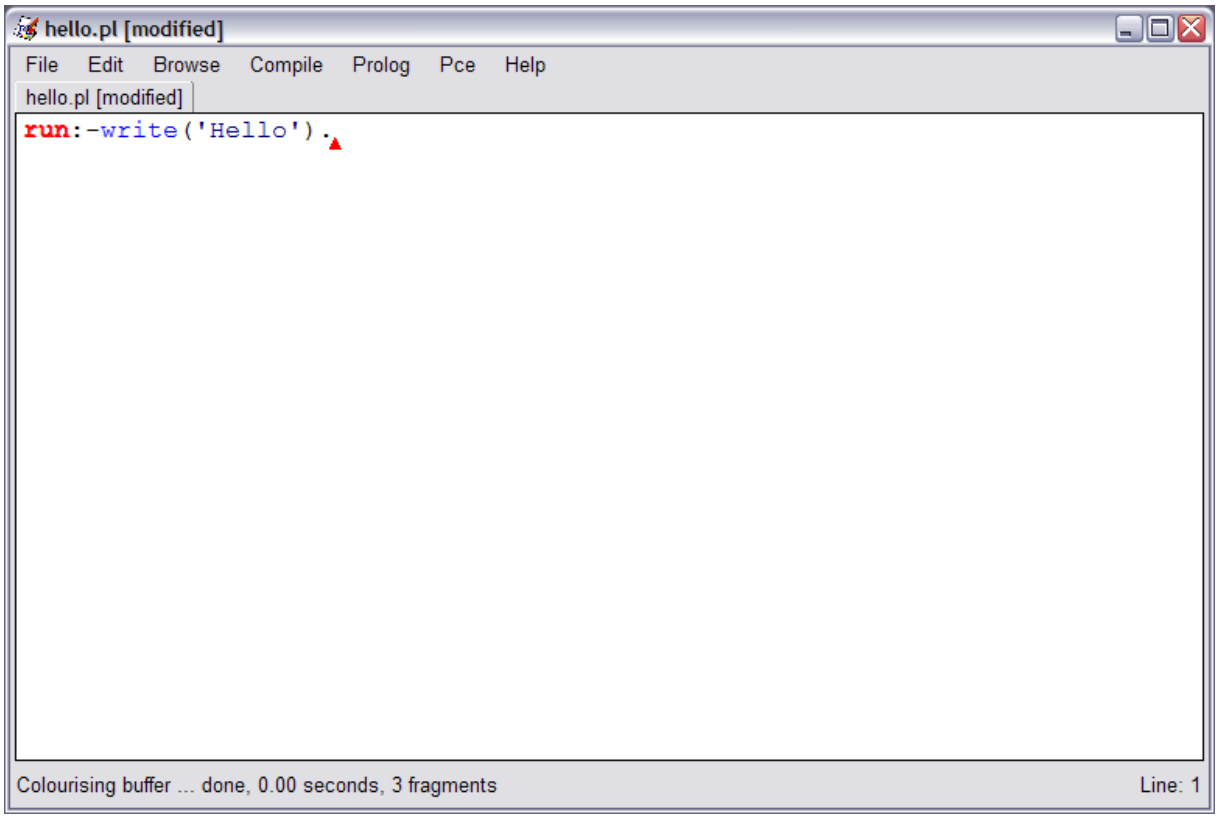



*Рисунок 4: Редактор программы*

Для сохранения программы можно использовать стандартное сочетание клавиш Ctrl-S.

Для проверки программы на отсутствие синтаксических ошибок, а также для компиляции, нужно выбрать меню Compile → Make. В случае обнаружения ошибок или предупреждений появится соответствующее окно. Если же ошибок нет, то в строке статуса появится сообщение “Make done”.

Для запуска программы нужно обратиться к меню File → Consult главного окна SWI-Prolog и указать запускаемую программу (или через меню редактора Compile → Consult selection). В главном окне появится сообщение вида «hello.pl compiled 0.00 sec, 588 bytes». Собственно для запуска программы нужно ввести следующий запрос:

```
run.
```

В результате в главное окно будет выведено сообщение Hello и строчка “true.”, говорящая о том, что данный запрос «истинен» и вывод ответов закончен.

Для просмотра справки используется встроенный предикат *help*.

```
help(assert).
```

**Контрольные вопросы**

1. Назовите разные виды реализации Prolog.
2. Какие средства разработки включает в себя SWI-Prolog?
3. Как запустить программу на SWI-Prolog?
4. Как скомпилировать программу на SWI-Prolog?
5. Как получить справочную информацию в среде SWI-Prolog?

**Задание**

1. Установите SWI-Prolog на свой компьютер.
2. Создайте простую программу в редакторе SWI-Prolog.
3. Скомпилируйте эту программу с помощью SWI-Prolog.
4. Запустите эту программу с помощью SWI-Prolog.
5. Откройте программу в интерактивной оболочке SWI-Prolog.

## Лабораторная работа № 11. Основы программирования на языке Prolog

### Цель работы

Изучить основные элементы языка Пролог, относящиеся к выражениям, потоковому вводу-выводу, базам данных и рекурсии.

### Теоретические основы

#### *Синтаксис языка Пролог*

Уже из примера “Hello, World!” можно увидеть синтаксис описания *предикатов* (предложений программного кода), который в общем виде выглядит так:

```
название:-описание.
```

Название предиката и его описание разделяет два символа «:-», которые можно читать как «это». Каждый предикат должен завершаться знаком «.» (точка). Название предиката должно начинаться со строчной (маленькой) буквы (названия, начинающиеся с прописной (большой) буквы, принадлежат переменным).

Предикаты делятся на *факты* и *правила*.

Символьные данные (строки) в языке Пролог должны заключаться в одинарные кавычки «'». Кавычки можно опускать, если текст не содержит пробелов и знаков операций. Двойные кавычки «”» превращают строку в список составляющих её символов.

Однострочный комментарий начинается с символа процента «%». Блочный комментарий обрамляется символами «/*» и «*/» (как в C++).

#### *Пример «Родственные отношения»*

Рассмотрим классический пример программы на Прологе, которая описывает родственные отношения (см. также книгу И. Братко).

Пусть генеалогическое дерево некоторого рода (семьи) выглядит так как изображено на рисунке ниже. Тогда на Прологе факты, что кто-то кому-то является родителем будут записаны следующим образом:

```
parent('pam', 'bob').
parent('tom', 'bob').
parent('tom', 'liz').
parent('kate', 'liz').
parent('bob', 'ann').
parent('mary', 'ann').
```

```
parent('bob', 'pat').
parent('dick', 'jim').
parent('ann', 'jim').
parent('jack', 'joli').
parent('pat', 'joli').
```

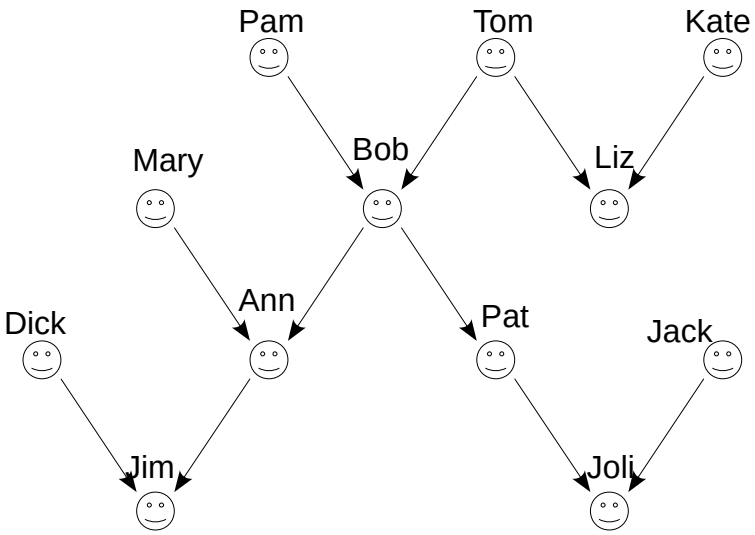



*Рисунок 5: Генеалогическое дерево рода*

Как следует читать данные факты? Например, “parent('pam', 'bob').” означает, что pam – родитель для bob.

Внесите эти предикаты в новую программу (например, под названием family) и загрузите её в систему Пролог.

Что с этим можно делать? Можно убедиться в том, что, например, Pam является родителем Bob'a:

```
parent('pam','bob').
```

В ответ должны получить “true.”.

Или проверить, не является ли Pam родителем для Liz:

```
parent('pam','liz').
```

Ответом должно быть “false.”.

В других системах Пролог вместо true и false будет выводиться yes и no, соответственно.

Далее, можно посмотреть все сведения о том, кто кому является родителем. Делается это следующим образом:

```
parent(X,Y).
```

Для системы Пролог *X* и *Y* являются объектами, которые она будет пытаться конкретизировать, исходя из того, что она знает о *parent*, так,

чтобы запрос был истинен. Ответы система будет выдавать по одному. Чтобы посмотреть следующий ответ нужно нажимать клавишу «;» (точка с запятой), для завершения вывода ответов нужно нажать клавишу «Ввод». Обратите внимание на то, что имена переменных должны начинаться с большой буквы или с символа подчёркивания «_», в противном случае система Пролог будет воспринимать эти имена просто как текст.

Какие ещё вопросы мы можем задать системе Пролог? Мы можем узнать, кто является родителем конкретного члена рода, например, для Bob'a:

```
parent(X,'bob').
```

Также можем узнать, кто дети Bob'a:

```
parent('bob',Y).
```

Мы даже можем узнать вообще, есть ли у кого-либо дети:

```
parent('liz',_).
```

В ответ на данный запрос мы должны получить ответ «false.». Символ подчёркивания «_» говорит системе Пролог о том, что нам всё равно какими данными будет конкретизирован этот объект.

Соответственно, теперь мы можем посмотреть имена тех, для кого известны их родители:

```
parent(_,Y).
```

Можно задавать системе Пролог и более сложные вопросы, например, мы можем узнать внуков Bob'a. Чтобы это выяснить мы можем поступить так: выяснить кто у Bob'a дети, а затем узнать их детей:

```
parent('bob,Y).
parent('ann',Z).
parent('pat,Z).
```

Очевидно, что такая рутинная работа должна выполняться самой системой Пролог. Для этого мы должны объединить два разных запроса в один, используя операцию *конъюнкции* (логическое И), которая в Пролог обозначается символом «,» (запятая).

```
parent('bob',Y),parent(Y,Z).
```

В этом запросе Y будет конкретизирован так, чтобы подходить для Bob, после чего Z будет конкретизирован подходящим значением для уже конкретизированного Y.

Аналогично можем  поинтересоваться, кто у Ann дедушка и бабушка:

```
parent(X,Y),parent(Y,'ann').
```

Итак, мы научились предоставлять системе Пролог факты и задавать ей вопросы.

Теперь научимся составлять правила. Начнём с простого примера. Так как мы знаем, кто кому является родителем, то мы можем отсюда также узнать кто кому является ребёнком. Что мы и делали в одном из примеров выше. Очевидно, что если X является родителем для Y, то Y является ребёнком для X. (Ребёнка в данном контексте лучше называть отпрыском, так как у 80 летнего дедушки ребёнком может быть 60 летний.) Запишем это на Пролог:

```
offspring(Y,X):-parent(X,Y).
```

Правило *offspring* мы определили через отношение *parent*. На «естественном» языке это означает следующее: Y является отпрыском для X, *если* X является родителем для Y.

Определив это правило мы теперь можем узнавать сведения об отпрысках используя его, а не отношение parent.

Для того, чтобы добавление к программе family было загружено в систему Пролог, нужно вновь скомпилировать программу.

Более сложные правила составляются с использованием *конъюнкции* (логическое И) и *дизъюнкции* (логическое ИЛИ). Выше мы выясняли кто для кого является дедушкой или бабушкой, а также внуком или внучкой. Давайте составим соответствующее правило. Назовём его grandparent:

```
grandparent(X, Y):-parent(X, Z),parent(Z,Y).
```

Расшифровывается это правило так: X является дедушкой или бабушкой для Y, если X является родителем для некоего Z, который является родителем для Y. Другими словами в генеалогическом дереве между X и Y есть общий непосредственно к ним примыкающий Z.

Выясним теперь кто является отцом. Отцом является мужчина, у которого есть ребёнок. Таким образом, нам не хватает данных о том, кто является мужчиной. Соответственно, для определения, кто является мамой, потребуется знать кто является женщиной.

```
male('tom').
male('dick').
male('bob').
male('jim').
male('jack').
female('pam').
female('kate').
female('mary').
```

```
female('liz').
female('ann').
female('pat').
female('joli').
```

Итак, X является отцом для Y, если X – мужчина и X – родитель для Y. Аналогично формулируется для мамы.

```
father(X,Y):-male(X),parent(X,Y).
mother(X,Y):-female(X),parent(X,Y).
```

Давайте теперь сформулируем отношение брат и сестра. X является братом для Y, если он мужчина и для X и Y имеется хотя бы один общий родитель. (Аналогично для отношения сестра.)

```
brother(X,Y):-male(X),parent(Z,X),parent(Z,Y).
sister(X,Y):-female(X),parent(Z,X),parent(Z,Y).
```

Давайте сделаем запрос:

```
sister(X,'pat').
```

Что мы видим? Оказывается Pat является не только сестрой для Ann, но и сестрой для самой себя! Для решения этой проблемы нужно указать системе Пролог, что X и Y должны быть различны. Делается это с помощью операции проверки на неравенство \=. (В других разновидностях Пролог эта операция может обозначаться по другому.)

```
brother(X,Y):-male(X),parent(Z,X),parent(Z,Y),X\=Y.
sister(X,Y):-female(X),parent(Z,X),parent(Z,Y),X\=Y.
```

Итак, с помощью программы family мы можем узнавать кто чей родитель, а также кто чей дедушка или бабушка. А как быть с предками более дальними? С прадедушками, прапрадедушками или так далее? Не будем же мы для каждого такого случая писать соответствующее правило, да и всё проблематичней это будет с каждым разом. На самом деле всё просто: X является для Y предком, если он является предком для родителя Y. Как это записать на Прологе? Ответ: с использованием рекурсии.

Ближайший предок является родителем:

```
predecessor(X,Y):-parent(X,Y).
```

Для тех же X кто отстоит дальше от Y правило запишется так:

```
predecessor(X,Y):-predecessor(X,Z),parent(Z,Y).
```

Проверим:

```
predecessor('pam','joli').
predecessor('bob','joli').
```

```
predecessor('ann','joli').
predecessor('ann', 'pat').
```

Первые два запроса срабатывают правильно, а на последний система отвечает “ERROR: Out of local stack”. Это говорит о том, что правило было сформулировано неверно, и в случае, если X не является предком для Y, то происходит бесконечная рекурсия. Переформулируем второе правило:

```
predecessor(X,Y):-parent(X,Z),predecessor(Z,Y).
```

Это правило говорит о том, что X является родителем некоего Z, который для X также является предком. Другими словами, Z – промежуточное звено между X и Y.

Наконец определим правило для выяснения кто кому является родственником. Во-первых, родственниками являются предки и потомки, например, сын и мать, бабушка и внук. Во-вторых, родственниками являются братья и сёстры в том числе двоюродные, троюродные и так далее, что в терминах предков означает, что у них общий предок. И, в-третьих, родственниками считаются те у кого общие потомки, например, муж и жена. Для определения первого правила будем использовать *дизъюнкцию* (логическое ИЛИ), которая обозначается символом «;» (точка с запятой).

```
relative(X,Y):-predecessor(X,Y);predecessor(Y,X).
relative(X,Y):-predecessor(Z,X),predecessor(Z,Y),X\=Y.
relative(X,Y):-predecessor(X,Z),predecessor(Y,Z),X\=Y.
```

К сожалению система Пролог при попытке просмотреть всех родственников некоего лица выдаёт повторяющиеся значения.

### *Выражения*

Для выполнения вычислений в программе на Пролог нужно использовать операцию присваивания значения переменной *is*.

```
X is 10.
Y is 5 * 2.
Z is X + Y.
```

Пролог поддерживает все основные арифметические операции: + (сложение), – (вычитание), * (умножение), ** или ^ (возведение в степень), / (вещественное деление), // (целочисленное деление), << (сдвиг влево), >> (сдвиг вправо), mod и rem (остаток от деления), /\ (битовое И), \/ (битовое ИЛИ), xor (битовое Исключающее ИЛИ), \= (проверка на неравенство), < (меньше), > (больше), =< (меньше или равно), >= (больше или равно). (Здесь перечислены не все операции.)

***Ввод-вывод***

В примере «Hello, World!» мы уже видели как выводить на консоль строковые данные. Для этого предназначен встроенный предикат *write*. Этот же предикат умеет выводить также и числа и содержимое переменных.

```
write(10).
X is 4 << 4, write(X).
```

Для чтения данных с клавиатуры можно использовать предикат *read*.

```
read(X), Y is X * X, write(X).
```

Если ввести число, то будет посчитан и выведен на консоль квадрат этого числа.

***Базы данных***

Для просмотра всех загруженных нами в систему Пролог отношений и правил можно использовать встроенный предикат *listing*. Если до вызова этого предиката мы никакой программы не загружали, то *listing* выдаст только *true*. Загрузите программу в систему Пролог программу «Родственные отношения» и перезапустите *listing*.

По сути программа «Родственные отношения» это база данных (БД). И её можно в процессе эксплуатации пополнять. Для этого предназначен встроенный предикат *assert*. Он предназначен для добавления в конец базы данных ещё одного предиката, и тем самым эквивалентен предикату *assertz*. Для добавления предиката в начало БД предназначен предикат *asserta*.

```
assert(parent(kate,liz)).
assert(female(kate)).
```

Удалять предикаты можно с помощью встроенного предиката *retract*.

```
retract(male(X)).
```

К сожалению, система Пролог не вносит изменения в БД, загруженную как часть кода программы, в чём прилежный читатель мог убедиться.

***Рекурсия***

Рекурсию мы уже затронули, когда рассматривали программу «Родственные отношения». Посмотрим теперь как можно использовать рекурсию при вычислениях.

*Факториал*

Как известно, факториал числа *N* это произведение целых чисел от 1 до *N*:

$$n!=\prod_{i=1}^{n} i$$

Если расписать формулу факториала, то получим следующее:

$$n!=1\cdot 2\cdot ...\cdot (n-1)\cdot n$$

Очевидно, что

$$1!=1$$

Для числа 0 факториал определяется равным 1:

$$0!=1$$

Посмотрим ещё раз на определение факториала и перепишем формулу в таком виде:

$$n!=n\cdot (n-1)\cdot ...\cdot 2\cdot 1$$

Из этой формулы хорошо видно, что

$$n!=n\cdot (n-1)!$$

Таким образом мы получили рекурсивное определение факториала (*рекуррентную формулу*):

$$n!=\begin{cases} 1, n=0 \\ n\cdot (n-1)!, n>0 \end{cases}$$

Запишем это на Пролог:

```
factorial(0,1).
factorial(N,F):-N1 is N - 1, factorial(N1,F1), F is N * F1.
```

Применённый здесь вид рекурсии называется *нисходящей рекурсией*. Но вычисление факториала для больших *N* можно ускорить, если применить *восходящую рекурсию*. Смысл её в том, чтобы как и при итеративном методе вычисления факториала начать с наименьшего значения для факториала, то есть с 1, и увеличивая на каждом шаге на 1, дойти до исходного N.

```
factorial2(N,F):-f(N,F,0,1).
f(N,F,N,F).
f(N,F,N1,F1):-N2 is N1 + 1, F2 is F1 * N2, f(N,F,N2,F2).
```

Факториал для любых *N* мы будем вычислять с помощью вспомогательного предиката *f*. Первое правило для *f* является граничным, то есть служит для останова вычислений, и сработает, когда *N* достигнет заданного значения. Например, если мы зададим *N* = 0, то это правило сработает сразу, так как будет вызвано *f(1,F,0,1)*, и *F* получит значение 1. Второе правило начнёт выполняться с *N1* = 1 и *F1* = 1. Это правило будет вызываться до тех, пор пока *N1* не сравняется с *N*.

На современном компьютере оба метода работают со скоростью на глаз неотличимой.

*Числа Фибоначчи*

Последовательность чисел Фибоначчи была исследована на Западе Леонардо Пизанским (Фибоначчи) в его труде «Liber Abaci» (1202). Фибоначчи рассматривал развитие идеализированной (биологически нереальной) популяции кроликов. (Подробнее о числах Фибоначчи можно прочитать в [Воробьёв].)

Определение *числа Фибоначчи* для натурального $N$ следующее:

$$f(n)=\begin{cases}1, n=1,2\\ f(n-1)+f(n-2), n>2\end{cases}$$

На языке Пролог это запишется так:

```
fib(1,1).
fib(2,1).
fib(N,F):-N1 is N - 2, N2 is N - 1, fib(N1,F1), fib(N2,F2), F is F1 + F2.
```

У этого метода есть существенный недостаток. Если внимательно посмотреть на код, то мы заметим, что одно и тоже число Фибоначчи считается многократно. И при N = 100 программа зависает на неопределённое время. Очевидно, что если бы наша программа запоминала предыдущие вычисленные числа Фибоначчи, то могла бы заработать значительно быстрее. Мы уже знаем как сохранять предикаты в БД, делается это с помощью *assertz*. Рассмотрим модифицированный код для вычисления чисел Фибоначчи:

```
:-dynamic(stored/1).
memo(G):-stored(G) -> true; G, assertz(stored(G)).
fib2(1,1).
fib2(2,1).
fib2(N,F):-
     N1 is N - 1, memo(fib2(N1,F1)),
     N2 is N - 2, memo(fib2(N2,F2)),
     F is F1 + F2.
```

В первой строке мы подгружаем предикат *stored*. Затем определяем предикат *memo*, который мы будем использовать для получения чисел Фибоначчи. Работает он так: сначала полученный *memo* предикат проверяется не сохранён ли он. Если сохранён, то возвращается сохранённый ответ, в противном случае происходит вычисление числа Фибоначчи и сохранение результата в БД.

На том же N = 100 эта программа не только не зависает, но и выдаёт ответ практически мгновенно.

**Задание**

1. Дополните программу family правилами для следующих родственных отношений:
   1. сын и дочь;
   2. дедушка и бабушка;
   3. внук и внучка;
   4. дядя и тётя;
   5. муж и жена;
   6. свёкор и свекровь;
   7. тесть и тёща;
   8. отчим и мачеха;
   9. свояк и свояченица;
   10. двоюродный брат и сестра.
2. Напишите на языке Пролог программу, которая вычисляет сумму натуральных чисел от 1 до $n$: $s(n)=\sum_{i=1}^{n} i$ с помощью нисходящей и восходящей рекурсии.
3. Напишите на языке Пролог программу, вычисляющую биномиальные коэффициенты: $C_n^k=\frac{n!}{k!\cdot(n-k)!}$ с помощью нисходящей и восходящей рекурсии.
4. Напишите на языке Пролог программу, которая вычисляет корни квадратного уравнения по его коэффициентам. Модифицируйте эту программу так, чтобы в ней осуществлялась проверка на существование корней уравнения.

## Лабораторная работа № 12. Обработка списков на языке Prolog

### Цель работы

Изучить синтаксис списков в языке Пролог, операцию разделения списка на голову и хвост и основные операции над списками: вычисление длины списка, поиск элемента в списке, добавление элемента в начало и конец списка, вставка элемента в список, удаление элемента из списка.

### Теоретические основы

*Список* это одна из самых часто используемых структур данных. Для него характерен перебор составляющих его элементов от начала до конца.

В языке Пролог элементы списка перечисляются через запятую и заключаются в квадратные скобки “[” и “]”:

```
[1, 2, 4, 8, 16]
['A', 'B', 'C']
```

Список без элементов называется *пустым списком* и обозначается так:

```
[]
```

Единственной встроенной операцией над списками является операция выделения нескольких первых элементов списка по отдельности и всех остальных в другой список. Чаще всего встречается выделение *головы* списка и его *хвоста*. Обозначается эта операция символом вертикальная черта “|”:

```
[Head | Tail]
[X, Y | Other]
```

Рассмотрим основные операции над списками.

#### *Длина списка*

Пример вычисления длины списка (количества элементов) на языке Пролог фундаментален, и, разобравшись с ним, можно будет легко разобраться и с остальными операциями над списками на этом языке.

Очевидно, что для того, чтобы узнать сколько элементов у списка, нужно их перебрать по одному. «Проблема» в том, что в Прологе нет циклов и нет операции для списков типа «для каждого» (foreach). Как известно, вместо циклов в программах на Прологе используется рекурсия. Как применить рекурсию к данной задаче? Во-первых, нужно задать правило окончания рекурсии. Так как для списков у нас есть только операция отделения головы от хвоста, то в процессе рекурсии список будет раз за

разом сокращаться, и в итоге станет пустым. Очевидно, что длина пустого списка, то есть его количество элементов, равна нулю. Обозначим правило для подсчёта количества элементов списка как *list_length* и запишем последний факт на Прологе:

```
list_length([],0).
```

Во-вторых, исчерпывая исходный список, мы уменьшаем его длину на 1 каждый раз, значит длина исходного списка равна длине хвоста списка плюс 1. Это же правило мы применяем и для хвоста. Таким образом мы дойдём до пустого списка. После чего пойдёт раскрутка рекурсии в обратном направлении. Запишем это на Пролог:

```
list_length([_|T],L):-list_length(T,L1),L is L1+1.
```

Разберём как это работает на конкретном примере. Пусть дан список [1,2,3]. Он не пуст, и значит Пролог для него будет применять второе правило. Сначала произойдёт отделение головы этого списка (так как нам не важно её значение, то мы используем переменную «_»), и хвост *T* получит значение [2,3], а длина *L* будет равна длине хвоста *L1* плюс 1. Так как хвост не пуст, то действия повторятся теперь уже над ним, и мы получим последовательно списки [3] и []. На последнем списке, так как он пуст, сработает первое правило, и *L1* для него получит значение 0. После чего пойдёт раскрутка стека рекурсии, и переменная *L1* станет равна предыдущему значению плюс 1, то есть 2, затем *L1* станет равна 2 плюс 1, то есть 3. Стек рекурсии опустошится и работа *list_length* закончится с ответом 3 в переменной *L*.

Схематически это можно представить так (в функциональном стиле):

```
length([1,2,3]) =
     length([2,3]) + 1 =
          (length([3]) + 1) + 1 =
               ((length([]) + 1) + 1) + 1 =
               ((0 + 1) + 1) + 1 =
          (1 + 1) + 1 =
     2 + 1 =
3
```

### *Принадлежность элемента списку*

Проверка на принадлежность списку некоторого значения реализуется проще чем вычисление длины списка. Очевидно, что мы легко можем определить совпадает ли заданное значение с головой списка. Обозначим соответствующее правило как is_element и запишем его:

```
is_element(X,[X|_]).
```

Хвост списка мы обозначили «_», так как нам всё-равно каков он, искомый элемент мы уже нашли.

Далее, если искомый элемент не совпал с головой, то мы должны посмотреть нет ли его в голове хвоста, что на Прологе как обычно запишется рекурсивно:

```
is_element(X,[_|T]):-is_element(X,T).
```

Теперь нам всё-равно какова голова списка, так как теперь мы ищем элемент в хвосте списка.

***Поиск позиции элемента в списке***

Смысл операции поиска позиции элемента в списке в том, чтобы найти номер элемента в списке совпавший с указанным значением. Алгоритм работы этой операции похож на алгоритм определения принадлежности списку указанного значения, только при переборе элементов списка мы ещё будем добавлять каждый раз добавлять 1 номеру элемента в списке. Пусть нумерация начинается с 1, а 0 означает, что элемента с таким значением в списке нет. Тогда, если значение совпало со значением головы списка, то позиция равна 1. Запишем это на Пролог:

```
find_by_value(H,[H|_],1).
```

Очевидно также, что если список пуст, то мы никогда в нём ничего не найдём, то есть результат поиска для любого заданного значения 0. Запишем и это на Пролог:

```
find_by_value(_,[],0):-fail,!.
```

Думаю, что в пояснении здесь нуждаются только *fail* и *!*. *fail* это что-то вроде указания системе Пролог на то, что результатом срабатывания этого правила будем «ложь» (false), а восклицательный знак служит признаком того, что дальше выполнять это правило не нужно, можно остановиться на полученном результате.

И, наконец, правило для общего случая:

```
find_by_value(X,[_|T],I):-find_by_value(X,T,I1),I is I1+1.
```

В процессе исчерпывания списка мы либо придём к тому, что сработает первое правило, то есть элемент всё-таки найдётся, либо список исчерпается до пустого, и сработает второе правило, и мы получим в качестве результата *false*. Подумайте над следующим вопросом: если искомых элементов в списке несколько, то сколько найдёт наша программа? Один? Если один, то какой по счёту? Или найдёт все? Что нужно изменить, что программа искала только один или все?

### *Сумма элементов числового списка*

Положим, что сумма элементов пустого списка равна 0.

```
list_sum([],0).
```

А сумму элементов непустого списка можно вычислять рекурсивно:

```
list_sum([H|T],S):-list_sum(T,S1),S is H + S1.
```

### *Поиск наименьшего элемента в списке*

Имеет смысл рассматривать непустые списки. Определим сначала правило для определения наименьшего среди двух чисел:

```
min(X,Y,X):-X<=Y,!.
min(_,Y,Y).
```

Это пример правила альтернативного оператору *if* императивных языков программирования. Первое определение сработает, если X <= Y, в противном случае сработает второе определение.

Вернёмся к спискам. В списке из одного элемента наименьшим является этот единственный элемент, что на Прологе записывается так:

```
list_min([H],H):-!.
```

Отсечение здесь нужно, чтобы остановить исчерпывание списка на списке из одного элемента.

При поиске наименьшего в списке из двух и более элементов будем пользоваться тем, что мы умеем определять наименьшее из двух чисел:

```
list_min([X,Y|T],Min):-list_min([Y|T],Min1),min(X,Min1,Min).
```

По этому правилу отделяется голова и ищется наименьший в хвосте списка, после чего выясняется, что является меньше – голова или наименьший из хвоста.

Поиск наибольшего элемента в списке оставляется для самостоятельного решения.

### *Добавление элементов в список*

Добавлять элементы можно в начало списка, в конец списка, а также в заданную своим номером позицию. В случае упорядоченных списков также можно добавлять элементы без нарушения упорядоченности.

Рассмотрим сначала добавление элемента в начало списка. Эта операция реализуется очень просто, благодаря тому, что мы имеем простой доступ к голове списка.

```
insert_first(X,L,[X|L]).
```

Добавление в конец списка реализуется стандартным рекурсивным методом. Сначала рассматриваем простейший случай. Им является добавление элемента в конец пустого списка.

```
insert_last(X,[],[X]).
```

Если же список не пуст, то мы должны добраться до конца списка, рекурсивно его исчерпывая:

```
insert_last(X,[H|T],R):-insert_last(X,T,L1),insert_first(H,L1,R).
```

Воспользовавшись тем фактом, что результирующий список *R* это [*H*|*L*], где *L* – результат вставки *X* в *T*, мы можем упростить правило *insert_last* следующим образом:

```
insert_last(X,[H|T],[H|L]):-insert_last(X,T,L).
```

Это стандартная техника, которую можно использовать для многих других операций со списками.

С правилом *insert_last* можно провести интересный эксперимент: можно посмотреть, что будет, если на вход *insert_last* подать результирующий список, а вставляемый элемент и список, в который он должен быть вставлен, задать переменными, например, так:

```
insert_last(X, S, [1,2,3,4]).
```

В ответ должны будем получить, что *X* равен *4*, а *S* – [*1*,*2*,*3*]! То есть Пролог умеет находить решение по ответу, если конечно правило для этого годится.

Добавление элемента в заданную позицию и в упорядоченный список оставляется для самостоятельного решения.

### *Удаление элемента из списка*

Так же как и с добавлением элементов удалять элементы можно из начала списка, из конца списка, а также из заданной позиции. В случае упорядоченных списков также можно удалять элементы без нарушения упорядоченности.

Рассмотрим здесь удаление элемента по указанной позиции. Как обычно легче всего работать с головой списка и с пустым списком:

```
remove_by_index(1,[_|T],T).
remove_by_index(_,[],[]).
```

Если удаляемый элемент не первый и список не пуст, то нужно удалять элемент из хвоста списка, при этом, естественно, номер позиции будет уменьшаться на 1.

```
remove_by_index(I,[H|T],[H|L]):-I1 is I-1,remove_by_index(I1,T,L).
```

Остальные операции по удалению элементов из списка выносятся на самостоятельное решение.

**Задание**

Реализуйте на языке Пролог следующие операции над списками:

1. печать элементов списка;
2. получение значения элемента списка по его номеру в списке;
3. произведение элементов числового списка;
4. поиск наибольшего элемента в списке;
5. проверка на упорядоченность списка;
6. вставка заданного значения в указанную позицию списка;
7. вставка заданного значения в упорядоченный список;
8. замена головы и последнего элемента списка на указанное значение;
9. замена элемента с указанной позицией на заданное значение;
10. удаление элемента из списка с указанным значением.

## Лабораторная работа № 13. Разработка алгоритмов на языке Prolog

### Цель работы

Изучить основы разработки алгоритмов на языке Prolog на примере базовых алгоритмов над списками: объединение списков, разбиение списка, обращение списка, циклический сдвиг элементов списка, перестановка элементов списка, обмен элементов списка, различные виды сортировок.

### Теоретические основы

#### *Объединение двух списков*

Под объединением двух списков будем понимать список, состоящий из всех элементов сначала первого списка, затем – второго. Реализацию операции объединения удобно разбить на два правила, первое для объединения пустого списка с любым другим, второе – для произвольных списков:

```
list_concat([],L2,L2).
list_concat([H|L1],L2,[H|L3]):-list_concat(L1,L2,L3).
```

Второе правило основано на том, что нам легко отделять голову списка, а затем назад её присоединять. Фактически здесь мы разбиваем первый список на отдельные элементы, а затем добавляем их, начиная с последнего, в начало второго:

```
list_concat( [1,2,3], [4,5] ) =
     [1 | list_concat( [2,3], [4,5] )] =
          [1 | [2 | list_concat( [3], [4,5] )]] =
               [1 | [2 | [3 | list_concat( [], [4,5] )]]] =
               [1 | [2 | [3 | [4,5]]]] =
          [1 | [2 | [3,4,5]]] =
     [1 | [2,3,4,5]] =
[1,2,3,4,5]
```

#### *Разбиение списка на два*

Разсмотрим здесь разбиение списка по первому найденному элементу с заданным значением. Например, список [1,2,3,4] по элементу со значением 2 будет разбит на списки [1] и [3,4].

Проще всего разбивать пустые списки, так как ответом всегда будет два пустых списка. Также просто разбивать список, голова которого совпа-

дает с заданным значением, тогда ответом будет пустой список и хвост исходного списка. Напишем это:

```
split_by_value(_,[],[],[]).
split_by_value(H,[H|T],[],T).
```

Если же заданное значение не совпало с головой списка, то отложим её в сторону и попытаемся разбить по этому значению хвост исходного списка. После разбиения хвоста добавим голову в начало первого результирующего списка:

```
split_by_value(X,[H|T],[H|R1],R2):-split_by_value(X,T,R1,R2).
```

***Циклический сдвиг элементов списка***

Рассмотрим сначала циклический сдвиг списка влево на один элемент. Например, для списка [1,2,3] ответом будет [2,3,1]. То есть голова списка удаляется и добавляется в конец исходного списка. Записывается это очень просто:

```
list_shift_1([H|T],R):-insert_last(H,T,R).
```

Теперь рассмотрим сдвиг списка влево на большее количество позиций. Очевидно, что сдвиг на N позиций можно свести к N сдвигам на одну позицию. Так как итерации нам недоступны, то воспользуемся рекурсией, будем сдвигать список на одни элемент, уменьшать количество сдвигов и вновь вызывать сдвиг на один элемент:

```
list_shift([H|T],I,R):-I1 is I-1,insert_last(H,T,L1),list_shift(L1,I1,R).
```

К несчастью одного правила как обычно недостаточно. Очевидно, что отсутствует правило для последнего сдвига, то есть когда передаваемое количество сдвигов сократиться до одного. Добавим это правило:

```
list_shift(L,1,R):-list_shift_1(L,R).
```

***Обращение списка***

Обращение списка (reverse) это изменение порядка его элементов на противоположный, например, для списка [1,2,3] обратным будет [3,2,1]. Проще всего обращать пустой список и список, состоящий из одного элемента. Запишем это:

```
list_reverse([],[]).
list_reverse([H],[H]).
```

Для более длинных списков будем поступать следующим образом: последовательно отделяем голову и добавляем её в конец:

```
list_reverse([H|T],L):-list_reverse(T,L1),insert_last(H,L1,L).
```

Можно предложить более быструю реализацию с использованием параметра-аккумулятора, который будет накапливать отделяемые от списков головы:

```
list_reverse2(S,R):-list_reverse2(S,[],R).
list_reverse2([H|T],Acc,R):-list_reverse2(T,[H|Acc],R).
list_reverse2([],R,R).
```

Первое правило заменяет основное правило на правило с аккумулятором. Аккумулятор получает в качестве начального значения пустой список. Второе правило выполняет основную работу: отделяет голову списка и ставит её в начало списка-аккумулятора. Третье правило служит для останова этого процесса. Посмотрим на коротком примере как это работает:

[1,2,3] : [] => [2,3] : [1 | []] => [3] : [2 | [1]] => [] : [3 | [2,1]] => R := [3,2,1]

Сортировка списка это его упорядочивание, то есть перестановка его элементов так, чтобы они удовлетворяли попарно отношению упорядоченности. Определим это отношение так:

```
gt(X,Y):-X > Y.
```

Рассмотрим три разных алгоритма сортировки: пузырьковую сортировку, быструю сортировку и метод вставки.

***Пузырьковая сортировка***

Алгоритм пузырьковой сортировки основан на многократной перестановке элементов списка пока они не станут попарно удовлетворять отношению упорядоченности. Например:

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

Основную часть алгоритма быстрой сортировки реализуем в правиле *swap()*, которое обменивает значениями два соседних элемента списка, если они удовлетворяют отношению *gt()*:

```
swap([X,Y|T],[Y,X|T]):-gt(X,Y).
swap([Z|T],[Z|T1]):-swap(T,T1).
```

Как обычно проще всего оперировать первыми элементами списка. *swap()* производит перестановку первых двух элементов, если отношение *gt()* истинно, то есть $X > Y$. Если первые два элемента переставлять не надо, то *swap()* вызывается для хвоста исходного списка. То есть *swap()* ищет в списке два соседних элемента, которые нужно переставить. Если таких элементов не было найдено, то *swap()* возвращает *false*, что может служить признаком окончания сортировки.

Для того, чтобы список стал отсортированным необходимо вызывать *swap()* многократно, по крайней мере до тех пор пока список не станет отсортированным. Для этого напишем правило *bublesort()*:

```
bublesort(L,S):-swap(L,L1),!,bublesort(L1,S).
bublesort(S,S).
```

Первое правило *bublesort()* работает до тех пор, пока *swap()* не вернёт *false*, после чего сработает второе определение, которое и вернёт нам отсортированный список.

***Быстрая сортировка***

Алгоритм быстрой сортировки основан на разделении исходного списка на два. В первый из списков попадают все элементы исходного списка, которые меньше первого, во второй — все, которые больше его. Такому разделению подвергаются и два полученный списка. Делается это до тех пор, пока списки не станут пустыми. После чего осуществляется обратный процесс — процесс объединения. Например:

[3,1,2,5,4] →

[1,2] – 3 – [5,4] →

([] – 1 – [2]) – 3 – ([4] – 5 – []) →

([] – 1 – ([] – 2 – [])) – 3 – (([] – 4 – []) – 5 – []) →

([] – 1 – [2]) – 3 – ([4] – 5 – []) →

[1,2] – 3 – [4,5] →

[1,2,3,4,5]

Напишем сначала правило для операции разделения списка на два:

```
split(_,[],[],[]).
split(X,[Y|T],[Y|Small],Big):-gt(X,Y),!,split(X,T,Small,Big).
split(X,[Y|T],Small,[Y|Big]):-split(X,T,Small,Big).
```

Теперь собственно правило для выполнения быстрой сортировки:

```
quicksort([],[]).
quicksort([H|T],S):-
split(H,T,Small,Big),quicksort(Small,SS),quicksort(Big,SB),
     concat_lists(SS,[H|SB],S).
```

Потребуется также определить правило для объединения списков:

```
concat_lists([],L,L).
concat_lists([H|T1],T2,[H|T3]):-concat_lists(T1,T2,T3).
```

### *Метод вставки*

Метод вставки работает следующим образом: сначала исходных список редуцируется до пустого посредством поочерёдного отбрасывания его элементов, начиная с первого. После чего происходит восстановление списка, но уже со вставкой ранее отброшенного элемента в такую позицию, чтобы получался отсортированный список. Например:

[3,1,2,5,4] →

[3 | [1,2,5,4]] →

[3 | [1 | [2,5,4]]] →

[3 | [1 | [2 | [5,4]]]] →

[3 | [1 | [2 | [5 | [4]]]]] →

[3 | [1 | [2 | [5 | [4 | []]]]] →

[3 | [1 | [2 | [5 | [4]]]]] →

[3 | [1 | [2 | [4,5]]]] →

[3 | [1 | [2,4,5]]] →

[3 | [1,2,4,5]] →

[1,2,3,4,5]

Определим сначала правило для операции вставки элемента в подходящую позицию:

```
insert(X,[Y|T],[Y|T1]):-gt(X,Y),insert(X,T,T1).
insert(X,T,[X|T]).
```

Затем собственно метод вставки:

```
insertsort([],[]).
insertsort([H|T],S):-insertsort(T,ST),insert(H,ST,S).
```

**Задание**

Реализуйте на языке Пролог следующие операции над списками:

1. разбиение списка на два по элементу с указанным индексом;
2. циклический сдвиг элементов списка вправо;
3. перестановка элементов списка;
4. обмен значениями двух элементов списка с указанными индексами;
5. слияние двух упорядоченных списков;
6. проверка, является ли один список частью другого списка.

## Список литературы

Ниже приведена краткая библиография и список ссылок на ресурсы Internet. Списки книг приведены в алфавитном порядке, а не в порядке предпочтения.

### Основная библиография


1. [Бёрд] Бёрд Р. Жемчужины проектирования алгоритмов. Функциональный подход. - Пер. с англ. - М.: ДМК Пресс, 2013. - 330 с.
2. [Братко: 1990] Братко И. Программирование на языке Пролог для искусственного интеллекта: пер. с англ. – М.: Мир, 1990. – 560 с.: ил.
3. [Братко: 2004] Братко И. Алгоритмы искусственного интеллекта на языке PROLOG, 3-е издание.: пер. с англ. – М.: ИД «Вильямс», 2004. – 640 с.: ил.
4. [Душкин: ФП] Душкин Р. В. Функциональное программирование на языке Haskell. – М.: ДМК Пресс, 2007. – 608 с., ил.
5. [Душкин: Справочник] Душкин Р. В. Справочник по языку Haskell. – М.: ДМК Пресс, 2008. – 544 с., ил.
6. [Душкин: Практика работы] Душкин Р. В. Практика работы на языке Haskell. – М.: ДМК Пресс, 2009. – 288 с., ил.
7. [Душкин: 14 эссе] Душкин Р. В. 14 занимательных эссе о языке Haskell и функциональном программировании. – М.: ДМК Пресс, 2011. – 140 с., ил.
8. [Клоксин, Меллиш] Клоксин У., Меллиш К. Программирование на языке Пролог: пер. с англ. – М.: Мир, 1987. – 336 с.: ил.
9. [Липовача] Липовача М. Изучай Haskell во имя добра! - Пер. с англ. - М.: ДМК Пресс, 2012. - 490 с.
10. [Марков] Марков В.Н. Современное логическое программирование на языке Visual Prolog 7.5: учебник. – СПб.: БХВ-Петербург, 2015. – 544 с.
11. [Марлоу] Марлоу С. Параллельное и конкурентное программирование на языке Haskell: Пер. с англ. -- М.: ДМК Пресс, 2014. -- 372 с.
12. [Мена] Мена Алехандро С. Изучаем Haskell. -- СПб.: Питер, 2015. -- 464 с.
13. [Себеста] Себеста Роберт У. Основные концепции языков программирования. – 5-е изд. – Пер. с англ. – М.: Вильямс, 2001. – 672 с.
14. [Стобо] Стобо Дж. Язык программирования Пролог: пер. с англ. – М.: Радио и связь, 1993. – 368 с.: ил.

15. [Тейт] Тейт Б. Семь языков за семь недель. Практическое руководство по изучению языков программирования. – Пер. с англ. – М.: ДМК-Пресс, 2017. – 384 с.

16. [Хоггер] Хоггер К. Введение в логическое программирование: пер. с англ. – М.: Мир, 1988. – 348 с.

**Дополнительная библиография**

1. [Ахо и др.: 2007] Ахо А., Хопкрофт Дж., Ульман Дж. Структуры данных и алгоритмы: Пер. с англ. – М.: Вильямс, 2007. – 400 с.
2. [Болтянский, Савин] Болтянский В.Г., Савин А.П. Беседы о математике. Книга 1. Дискретные объекты. – М.: ФИМА, МЦНМО, 2002. – 368 с.
3. [Воробьёв] Воробьёв Н.Н. Числа Фибоначчи, 4-е изд. – М.: Наука, 1978. – 144 с.
4. [Ландо] Ландо С.К. Введение в дискретную математику. – М.: МЦНМО, 2012. – 265 с.
5. [Макконнелл] Макконнелл Дж. Анализ алгоритмов. Активный обучающий подход. – 3-е изд. – М.: Техносфера, 2009. – 416 с.
6. [Окулов: ДМ] Окулов С.М. Дискретная математика. Теория и практика решения задач по информатике: учебное пособие. – М.: БИНОМ. Лаборатория знаний, 2008. – 422 с.
7. [Окулов] Окулов С. М. Программирование в алгоритмах. – 3-е изд. – М.: БИНОМ, 2007. – 383 с., ил.
8. [Скиена] Скиена С. Алгоритмы. Руководство по разработке. – 2-е изд.: пер. с англ. – СПб.: БХВ-Петербург, 2011. – 720 с.
9. [Шень] Шень А. Программирование: теоремы и задачи. – 2-е изд. – М.: МЦНМО, 2004. – 296 с.

**Internet-ссылки**

1. [Stepik: Haskell] Функциональное программирование на языке Haskell – https://stepik.org/course/75/
2. [Stepik: Haskell2] Функциональное программирование на языке Haskell (часть 2) – https://stepik.org/course/693/
3. Шевченко Д. О Haskell по-человечески (для обыкновенных программистов) – https://www.ohaskell.guide/
4. Практика функционального программирования – http://fprog.ru/
5. Haskell Platform – https://www.haskell.org/platform/
6. Stackage – https://www.stackage.org

7. Haskell Cabal – https://www.haskell.org/cabal/
8. Atom – https://atom.io
9. Leksah – http://www.leksah.org
10. GNU Emacs – https://www.gnu.org/software/emacs/
11. Haskell-mode for Emacs – https://github.com/haskell/haskell-mode
12. Elm - https://elm-lang.org
13. Русскоязычное сообщество разработчиков на языке Elm - https://vk.com/elm_lang_ru

ЯЗЫКИ И СИСТЕМЫ ПРОГРАММИРОВАНИЯ

Методические указания по выполнению лабораторных работ
для студентов всех форм обучения

Составитель: Симоненко Евгений Анатольевич

Авторская редакция

Компьютерная верстка Симоненко Евгений Анатольевич

ФГБОУ ВО «Кубанский государственный технологический университет»

350072, г. Краснодар, ул. Московская, 2, кор. А