МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ

РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное образовательное учреждение высшего профессионального образования

«Чувашский государственный университет им. И.Н.Ульянова»

АЛАТЫРСКИЙ ФИЛИАЛ

Факультет управления и экономики
Кафедра высшей математики и информационных технологий

ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

Специальность «Математическое обеспечение и администрирование информационных систем»

Методические указания к контрольной работе

Преподаватель: доц. Федоров Р.В.

2010-2011 учебный год

Алатырь 2010

Основной целью дисциплины является формирование у будущих специалистов практических навыков по основам функционального программирования, развитие умения работы с персональным компьютером на высоком пользовательском уровне, обучение работе с научно-технической литературой и технической документацией по элементам функционального программирования.

Задачи изучения дисциплины – изложение основных принципов функционального программирования, их основных применений в современном программировании, дать студенту ориентиры в дальнейшем углубленном изучении отдельных вопросов в специализированных разделах математической логики и функционального программирования. Студенты должны освоить основные понятия и методы, включенные в каждый раздел. Допускается исключение отдельных тем и сокращение их содержания в каждом разделе.

Дисциплина «Функциональное программирование» является основным источником информации в области функционального подхода к программированию и служит основой для дальнейшей профессиональной специализации в этой области.

Перечень дисциплин, знание которых необходимо для изучения данной дисциплины:

№ п/п	Наименование дисциплины	Разделы (темы), усвоение которых необходимо для изучения данной дисциплины
1	Математический анализ	Весь курс
2	Функциональный анализ	Весь курс
3	Математическая логика	Весь курс
4	Структуры и алгоритмы компьютерной обработки данных	Весь курс
5	Информатика	Весь курс
6	Программирование	Весь курс

В результате изучения дисциплины студент должен:

а) знать

- особенности задач искусственного интеллекта и роль функционального программирования как методологии решения этих задач;

- тенденции и перспективы развития инструментальных средств функционального программирования;

- основы теории и практики лямбда-исчисления;

б) уметь:

- разрабатывать программные приложения для решения поставленных задач на функциональном языке программирования;

-разрабатывать алгоритмы решения задач для функционального программирования.

в) иметь представление:

- о современном этапе развития функционального программирования;

- об основах лямбда-исчисления;
- об основах объектно-ориентированного программирования в функциональном программировании.

ОБЩИЕ УКАЗАНИЯ

Задание выполняется по вариантам, определяемым по соответствующему алгоритму преподавателем совместно со студентом

Для написания контрольной работы можно использовать один или несколько источников по данной теме. При анализе источников следует обратить внимание на связь материала источника с материалом лекций.

Текст печатается на листах формата А4 через 1,5 интервала. Поля стандартные. Объем не менее 12 страниц. В тексте должны быть ссылки на использованную литературу, список которой приводится в конце. Задания выполняются в строгой последовательности: сначала указывается условие, затем ответ.

Контрольную работу необходимо представить в сроки, указанные в учебном графике. Работы, не отвечающие требованиям методических указаний, не засчитываются.

Контрольная работа оформляется в следующем виде:

1. титульный лист;
2. содержание;
3. затем приводятся:

для теоретических заданий – вариант ответа;

для практических заданий – распечатки результатов выполненной работы на компьютере и описание проделанных действий.

4. список использованной литературы

КРАТКИЕ ТЕОРИТИЧЕСКИЕ СВЕДЕНИЯ

Языки функционального программирования

Наиболее известными языками функционального программирования являются:

•**Lisp** (List processor). Считается первым функциональным языком программирования. Нетипизирован. Содержит массу императивных свойств, однако в общем поощряет именно функциональный стиль программирования. При вычислениях использует *вызов-по-значению*. Сейчас наиболее распространён диалект Common Lisp, ушедший от принципов ФП. Язык сложен в изучении, имеет очень непривычный синтаксис. Существует объектно-ориентированный диалект языка — CLOS.

•**Scheme**. Один из многих диалектов языка Lisp, предназначенный для научных исследований в области информатики. При разработке Scheme был сделан упор на элегантность и простоту языка. Благодаря этому язык получился намного меньше, чем базовая версия языка Lisp. Отличается простотой как самого языка, так и стандартной библиотеки функций, хотя несколько проигрывает в универсальности. Кроме того, здесь точнее соблюдаются принципы функционального программирования.

•**ISWIM** (If you See What I Mean). Функциональный язык-прототип. Разработан Ландиным (США) в 60-х годах для демонстрации того, каким может и должен быть язык функционального программирования. Вместе с языком Ландин разработал и специальную виртуальную машину для исполнения программ на языке ISWIM. Эта виртуальная машина, основанная на *вызове-по-значению*, получила название SECD-машины. На синтаксисе языка ISWIM базируется синтаксис многих функциональных языков. На синтаксис ISWIM похож синтаксис ML, особенно Caml.

•**ML** (Meta Language). Целое семейство строгих функциональных языков с развитой полиморфной системой типов и параметризуемыми модулями. ML преподаётся во многих университетах мира (в некоторых даже как первый язык программирования).

•**Standard ML**. Один из первых типизированных языков функционального программирования. Содержит некоторые императивные свойства, такие как ссылки на изменяемые значения, и поэтому не является чистым. При вычислениях использует *вызов-по-значению*. Очень интересная реализация модульности. Мощная полиморфная система типов. Последний стандарт языка — Standard ML-97, существует формальное математическое определение синтаксиса, а также статической и динамической семантик языка.

•**Caml Light** и **Objective Caml**. Как и Standard ML принадлежит к семейству ML. Objective Caml отличается от Caml Light, в основном, поддержкой классического объектно-ориентированного программирования. Objective Caml, также как и Standard ML, является строгим, но имеет некоторую встроенную поддержку отложенных вычислений.

•**Miranda**. Функциональный язык, разработанный Дэвидом Тернером (США), в качестве стандартного функционального языка, использовавшего отложенные вычисления. Имеет строгую полиморфную систему типов. Как и ML этот язык преподаётся во многих университетах. Оказал большое влияние на разработчиков языка Haskell.

•**Haskell**. Один из самых распространенных нестрогих языков функционального программирования. Имеет очень развитую систему типизации. Несколько хуже разработана система модулей. Последний стандарт языка — Haskell-98.

•**Gofer** (GOod For Equational Reasoning). Упрощенный диалект языка Haskell. Предназначен для обучения основам функционального программирования.

•**Clean**. Функциональный язык, специально предназначенный для параллельного и распределенного программирования. По синтаксису напоминает Haskell. Чистый. Использует отложенные вычисления. С компилятором поставляется набор библиотек (I/O libraries), позволяющих программировать графический пользовательский интерфейс под Win32 или MacOS.

•**Erlang**. Язык, разработанный компанией Ericsson, ориентирован на сферу коммуникаций; происходит от Пролога, хотя сейчас похож на него лишь внешне. Имеет лёгкий в освоении синтаксис, богатейшую библиотеку, в том числе: переносимый графический интерфейс, СУБД, распределенные вычисления и многое другое; причем всё это доступно бесплатно. Язык является параллельным изначально — возможность параллельной работы нескольких процессов и их взаимодействия заложена на уровне синтаксиса. Недостаток пока только один: компиляция в байт-код, для которого нужен "тяжёлый" и не слишком быстрый интерпретатор.

•**Рефал**. Функциональный язык, разработанный в СССР ещё в семидесятых годах XX века. В некоторых отношениях близок к Прологу. Крайне эффективен для обработки сложных структур данных типа текстов на естественном языке, XML и т.д. Эта эффективность обусловлена тем, что Рефал является единственным языком, использующим двунаправленные списки — это позволяет сократить объём некоторых программ и ускорить их работу (за счёт отсутствия необходимости в сборке мусора). К сожалению, в последнее время разрабатывается не очень активно. С другой стороны, по языку много документации на русском языке; имеется суперкомпилятор — система оптимизации программ на уровне исходных текстов (т.е. фактически оптимизация алгоритма!).

•**Joy**. Чистый функциональный язык, основанный на комбинаторной логике. Внешне похож на Форт: используется обратная польская запись и стек. Пока что находится в стадии развития, хотя уже имеет неплохую теоретическую основу. Существует даже прототипная реализация. Основное преимущество перед другими языками — линейная запись без переменных, что должно резко облегчить автоматизацию написания, проверки и

оптимизации программ. Помимо недоразвитости, есть еще один серьезный недостаток: как и у Форта, у Joy страдает читабельность.

Интернет-ресурсы по функциональному программированию

- **www.haskell.org** — очень насыщенный сайт, посвященный функциональному программированию в общем и языку Haskell в частности. Содержит различные справочные материалы, список интерпретаторов и компиляторов языка Haskell (в настоящий момент все интерпретаторы и компиляторы бесплатны). Кроме того, имеется обширный список интересных ссылок на ресурсы по теории функционального программирования и другим языкам (Standard ML, Clean).
- **cm.bell-labs.com/cm/cs/what/smlnj** — Standard ML of New Jersey. Очень хороший компилятор. В бесплатный дистрибутив помимо компилятора входят утилиты MLYacc и MLLex и библиотека Standard ML Basis Library. Отдельно можно взять документацию по компилятору и библиотеке.
- **www.harlequin.com/products/ads/ml/** — Harlequin MLWorks, коммерческий компилятор Standard ML. Однако в некоммерческих целях можно бесплатно пользоваться версией с несколько ограниченными возможностями.
- **caml.inria.fr** — институт INRIA. «Домашний» сайт команды разработчиков языков Caml Light и Objective Caml. Можно бесплатно скачать дистрибутив Objective Caml, содержащий интерпретатор, компиляторы байт-кода и машинного кода, Yacc и Lex для Caml, отладчик и профайлер, документацию, примеры. Качество компилированного кода у этого компилятора очень хорошее, по скорости опережает даже Standard ML of New Jersey.
- **www.cs.kun.nl/~clean/** — содержит дистрибутив компилятора с языка Clean. Компилятор коммерческий, но допускается бесплатное использование в некоммерческих целях. Из того, что компилятор коммерческий, следует его качество (очень быстр), наличие среды разработчика, хорошей документации и стандартной библиотеки.

Основы работы с HUGS 98

После запуска ИС HUGS 98 на экране появляется диалоговое окно среды разработчика, автоматически загружается специальный файл предопределений типов и определений стандартных функций на языке Haskell (Prelude.hs) и выводится стандартное приглашение к работе.

Диалоговое окно среды разработчика состоит из главного меню, набора кнопок для быстрого доступа к наиболее часто используемым командам и консоли, где происходит работа с интерпретатором. Необходимо особо отметить, что ИС не позволяет создавать и редактировать файлы с кодами программ, для этого требуется использование любого текстового редактора, поддерживающего обычный стандарт TXT (этим редактором, например, может быть стандартный блокнот Windows).

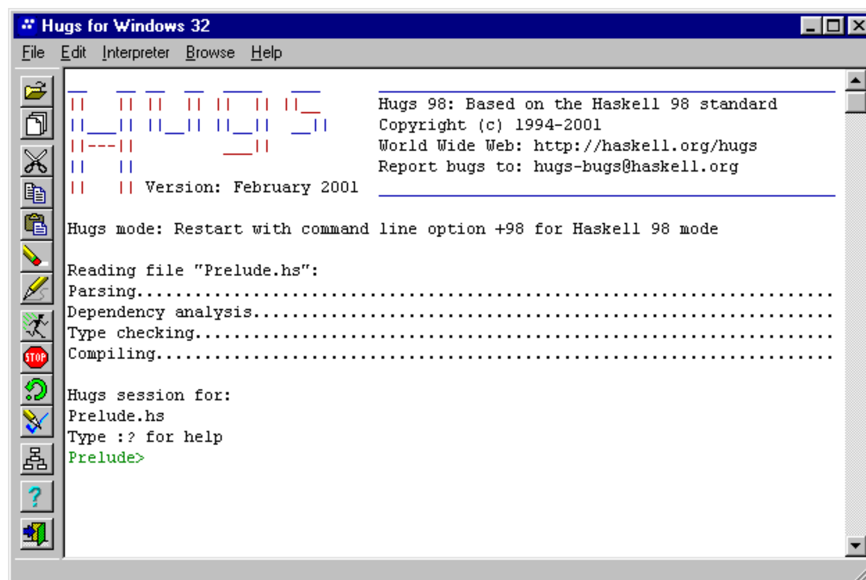


Рис. 1. Главное окно ИС HUGS 98

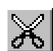
Главное окно HUGS 98 (рис. 1) обеспечивает доступ ко всем командам ИС, используемым для интерпретации и отладки программ. Кроме того, ИС позволяет вызвать на редактирование открытые модули в стандартном окне Notepad, встроенном в операционную систему Windows любой версии.

Панель инструментов HUGS 98

На панели инструментов, находящейся слева на главном диалоговом окне, предоставлены кнопки, при помощи которых можно вызвать наиболее часто используемые в процессе разработки команды (конечно, с точки зрения разработчиков ИС HUGS 98, а не с точки зрения конечного пользователя). Ниже представлены краткие описания всех четырнадцати кнопок, которые можно найти на панели инструментов.

Загрузка модулей из внешних файлов. Позволяет выбрать и открыть файл, из которого загружаются все модули, обнаруживаемые интерпретатором в этом файле.

Вызов менеджера модулей. Менеджер модулей позволяет добавлять, удалять и редактировать загруженные в память ИС программные модули.



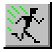
Вырезать выделенный текст. Стандартная функция редактирования текстов. Удаляет из редактора выделенный текст и помещает его в буфер обмена операционной системы.

Скопировать выделенный текст в буфер обмена. Стандартная функция редактирования текстов. Копирует выделенный текст в буфер обмена операционной системы.

Вставить текст из буфера обмена. Стандартная функция редактирования текстов. Вставляет в редактируемый тест содержимое буфера обмена операционной системы.

Очистить выбранный текст. Стандартная функция редактирования текстов. Удаляет из редактора выделенный текст, не помещая его в буфер обмена операционной системы.

Запустить внешний редактор текста. Запускает внешний тектовый редактор, зарегистрированный в операционной системе. Для семейства Windows при нажатии на эту кнопку запускается стандартная программа Notepad.

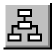


Запуск на выполнение выражения «*main*». Исполняет функцию main в загруженных модулях (конечно, если такая функция обнаружена в модулях). Если функция *main* не обнаружена ни в одном из загруженных мдулей, то выдаётся ошибка: **ERROR — Undefined variable "main"**.

Остановка исполнения программы. Остановка выполнения любой запущенной функции. Используется, например, для прекращения вычисления бесконечного списка.

Перезагрузка всех файлов текущего проекта. Осуществляет перезагрузку всех файлов с целью загрузить в память интерпретатора все сделанные изменения в коде проекта.

Установка параметров интерпретатора. Вывод на экран диалогового окна установки набора параметров интерпретатора языка Haskell. О параметрах интерпретатора подробно описано в приложении В.



Вывод на экран иерархии классов. На экране появляется иерархия классов текущего проекта, показанная в виде множества прямоугольников с названиями (классы) и связей между ними (отношения наследования).

Вызов справки. Вызывает на экран стандартное диалоговое окно справочной информации. Предполагается, что все справочные файлы присутствуют в каталоге, где установлено ИС (эти файлы не входят в стандартную поставку HUGS 98).

Выход из программы. Осуществляет выход из ИС HUGS 98 в операционную систему.

Команды консоли HUGS 98

Консоль ИС HUGS 98 предоставляет небольшой набор служебных конструкций, позволяющих управлять работой ИС. Многие из этих команд дублируют действия кнопок на панели инструментов и некоторые пункты главного меню приложения. Однако в любом случае эти команды могут позволить профессиональным пользователям значительно ускорить процесс разработки.

Каждая команда начинается с символа «двоеточие» — «:». Это сделано для того, чтобы отличить встроенные команды от написанных разработчиками функций. Кроме того, ИС позволяет сокращать каждую команду вплоть до одной буквы, набрав только символ «двоеточие» и собственно первую букву команды.

Всего существует девятнадцать команд, ниже представлено подробное описание каждой из них.

:load [<filenames>]

Загружает программные модули из заданных файлов (имена файлов можно разделить пробелом). Дублирует кнопку загрузки модулей на панели инструментов. Если имена файлов отсутствуют, то происходит выгрузка всех модулей, кроме стандартного (Prelude.hs). При повторном использовании команды все ранее загруженные модули выгружаются из памяти интерпретатора.

:also <filenames>

Подгружает дополнительные модули в текущий проект. Имена файлов должны быть разделены пробелами (если указывается более чем один файл).

:reload

Повторяет последнюю выполненную команду загрузки (:load). Позволяет быстро выполнить перезагрузку модуля в случае, если он редактируется во внешнем текстовом редакторе.

:project <filename>

Загружает и использует файл проекта. Загрузить можно только один файл. Файлы проекта используются для объединения разрозненных файлов с кодом. При повторном использовании команды происходит выгрузка всех файлов (как проектных, так и обычных) из памяти интерпретатора.

:edit [<filename>]

Вызывает внешний текстовый редактор для исправления указанного файла. Если имя файла не указано, то на редактирование вызывается последний файл (загруженный или редактированный). Данная команда дублирует кнопку вызова внешнего текстового редактора на панели инструментов.

:module <module>

Устанавливает заданный модуль в качестве текущего для выполнения функций. Эта команда предназначена, в первую, очередь для разрешения коллизий имён.

<expr>

Запуск заданного выражения на выполнение. Например, команда main запустит на выполнение соответствующую функцию — main, что произведёт дублирование кнопки с панели инструментов.

:type <expr>

Выводит на экран тип заданного выражения. Эта команда используется, главным образом, в отладочных целях для быстрого получения типа создаваемого выражения (переменной, функции, сложного объекта).

:?

Выводит на экран список команд с кратким описанием.

:set [<options>]

Позволяет задать параметры ИС с командной строки. Дублирует действие диалогового окна настройки HUGS 98 (описание которого приведено в приложении B). Все возможные параметры этой команды (<options>) выводятся на экран при выполнении этой команды без каких-либо параметров.

:names [pat]

Выводит на экран список всех имён объектов, которые находятся в текущем (если не задано иное) пространстве имён.

:info <names>

Выводит на экран описание заданных имён объектов. Например, для функций выводит их тип вместе с именем заданной функции.

:browse <modules>

Выводит на экран список всех объектов (функций, переменных, типов), определённых в заданных модулях. Имена модулей должны быть разделены пробелом (в случае, если указано более одного имени модуля).

:find <name>

Вызывает на редактирование модуль, содержащий заданное имя. Если заданного имени нет ни в одном из текущих модулей, то выдаётся собщение об ошибке: **ERROR — No current definition for name "<name>"**.

:!<command>

Выходит в операционную систему и выполняет заданную команду. Необходимо особо отметить, что между символом «восклицательный знак» и именем команды операционной системы должен отсутствовать пробел.

:cd <directory>

Изменяет текущий каталог, с которым работает HUGS 98.

:gc

Принудительно запускает на выполнение процесс сборки мусора. После этого выводит на экран статистику о собранных и восстановленных ячейках памяти.

:version

Выводит на экран информацию о версии установленного интерпретатора языка Haskell и ИС HUGS 98.

:quit

Осуществляет выход в операционную систему. Дублирует кнопку на панели инструментов.

Как работать с Hugs

1. Запуск

В папке Hugs-98 запускаем файл Hugs.exe

2. Вычислим какое нибудь выражение

Введите выражение и нажмите Enter.

Например:

2 + 2 (увидите ответ 4)

sin 1 (увидите ответ 0.841470984807897)

sum [1..1000] (увидите ответ 500500 – это сумма чисел от 1 до 1000)

3. Опишем свою функцию

а. Создаем файл с описанием функции

Создайте текстовый файл с помощью любого текстового редактора (например, Блокнот).

Например, создадим файл cube.hs, содержавший одну строку:
cube x = x * x * x

Замечание: Можно прямо в Hugs ввести команду **:e** ***<имя файла>*** (например, :e e:\test\cube.hs). При этом запустится Блокнот и откроет (а при необходимости и создаст) этот файл.

б. Загрузим файл

Введите команду **:l** ***<имя файла>*** (или выберите пункт *Open* в меню *File* или нажмите верхнюю кнопку на панели инструментов).

Если мы написали описание функции без ошибок, то она загрузится в память, и функцию можно будет использовать.

Например,

:load c:\test\cube.hs (загружается файл с описанием функции cube)
cube 3 (увидите ответ 27)

4. Редактируем файл (например, исправляем ошибки)

Допустим, при загрузке файла были сообщения об ошибках или просто мы хотим что-то изменить в программе.

Введите команду **:e** (или выберите пункт *Text Editor* в меню *Edit* или нажмите кнопку, на которой изображен карандаш, на панели инструментов).

Запустится редактор Блокнот и в нем откроется последний загруженный файл. Внесите нужные вам исправления в программу и закройте редактор. После этого WinHugs автоматически перезагрузит измененный файл.

Замечание: Другой вариант, можно отредактировать файл любым текстовым редактором и ввести команду **:r** (или нажать кнопку, на которой изображена зеленая круглая стрелка, на панели инструментов). WinHugs перезагрузит последний загруженный файл.

5. Выход из программы

Введите команду **:q** (или выберите пункт *Exit* в меню *File* или нажмите самую нижнюю кнопку на панели инструментов).

Параметры ИС HUGS 98

ИС HUGS 98 предоставляет программисту возможность тонко настраивать интерпретатор и саму ИС под ту или иную задачу. Это возможно при помощи изменения настроек (параметров) ИС. На рис. 2 показано состояние настроек, загружаемых по умолчанию (такой набор параметров действует при первоначальной установке HUGS 98).

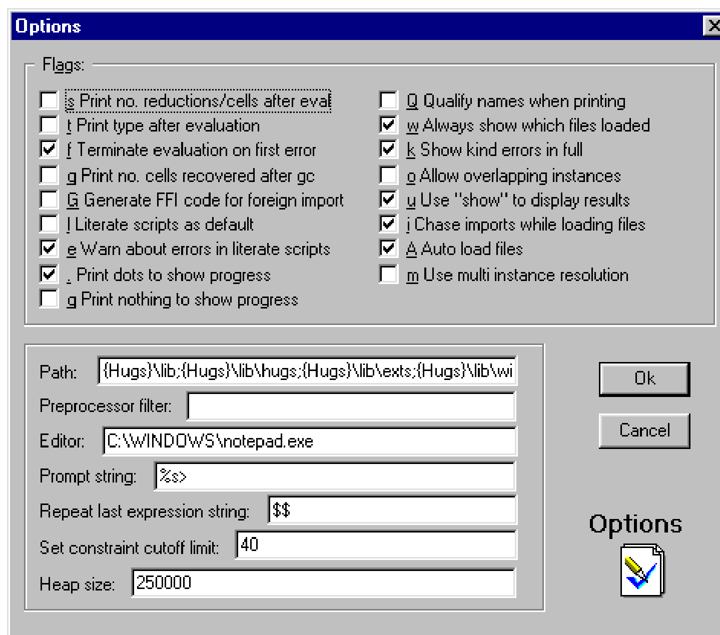


Рис. 2. Диалоговое окно установка параметров ИС

В верхней части представленного диалогового окна находится набор так называемых флагов, значения которых могут быть либо ИСТИНА (флаг установлен), либо ЛОЖЬ (флаг сброшен). Каждый флаг отвечает за тот или иной параметр интерпретатора или самой оболочки.

В нижней части диалогового окна настроек находятся поля ввода внутренних переменных окружения ИС HUGS 98.

Каждый флаг представлен определённой буквой латинского алфавита в верхнем или нижнем ключе. Далее описываются все имеющиеся флаги, обозначаемые соответствующими буквами:

s — распечатывать количество редукций и ячеек памяти после выполнения вычислений;

t — распечатывать тип выражения после его вычисления;

f — прерывать вычисление после первой ошибки;

g — распечатывать количество ячеек памяти, собранный во время сборки мусора;

G — генерация кода FFI для импортированный файлов;

l — оптимизация скриптов по умолчанию;

e — предупреждать об ошибках в оптимизированных скриптах;

. — распечатывать точки для визуализации процесса вычисления;

q — ничего не распечатывать для визуализации процесса вычисления;

Q — квалифицировать имена во время распечатки;

w — всегда показывать названия загруженных файлов;

k — полностью показывать тип и описание ошибок;

o — позволять пересекаться экземплярам классов;

u — использовать функцию «show» для отображения результатов;

i — удалять импортированные файлы при загрузке новых;

A — автоматически загружать файлы;

m — использовать множественную резолюцию экземпляров классов.

«Основы языка Haskell»

Структуры данных и их типы

Одна из базовых единиц любого языка программирования — символ. Символом традиционно называется последовательность букв, цифр и специальных знаков ограниченной или неограниченной длины. В некоторых языках строчные и прописные буквы различаются, в некоторых нет. Так в Lisp'е различия между строчными и заглавными буквами нет, а в Haskell'е есть.

Символы чаще всего выступают в качестве идентификаторов — имен констант, переменных, функций. Значениями же констант, переменных и функций являются типизированные последовательности знаков. Так значением числовой константы не может быть строка из букв и т.п. В функциональных языках существует базовое понятие — атом. В реализациях атомами называются символы и числа, причем числа могут быть трех видов: целые, с фиксированной и с плавающей точкой.

Следующим понятием функционального программирования является список. В абстрактной математической нотации использовались символы [], которые также используются в Haskell'е. Но в Lisp'е используются обычные «круглые» скобки — (). Элементы списка в Lisp'е разделяются пробелами, что не очень наглядно, поэтому в Haskell'е было решено ввести запятую для разделения. Таким образом, список [a, b, c] будет правильно записан в синтаксисе Haskell'а, а в нотацию Lisp'а его необходимо перевести как (a b c). Однако создатели Lisp'а пошли еще дальше в своей изощрённости. Допускается использовать точечную запись для организации пары, поэтому приведенный выше список можно записать как (a.(b.(c.NIL))).

Списочные структуры в Lisp'е и Haskell'е описываются в соответствии с нотацией — заключение одного списка в другой. При этом в нотации Lisp'а сделано послабление, т.к. перед скобкой внутреннего списка можно не ставить пробел.

Типы данных в функциональных языках определяются автоматически. Механизм автоматического определения типа встроен и в Haskell. Однако в некоторых случаях необходимо явно указывать тип, иначе интерпретатор может запутаться в неоднозначности (в большинстве случаев будет выведено сообщение об ошибке или предупреждение). В Haskell'е используется специальный символ — :: (два двоеточия), котрый читается как «имеет тип». Т.е. если написать:

```
5 :: Integer
```

Это будет читаться как «Числовая константа 5 имеет тип Integer (Целое число)».

Однако Haskell поддерживает такую незаурядную вещь, как полиморфные типы, или шаблоны типов. Если, например, записать [a], то это будет обозначать тип «список из атомов любого типа», причем тип атомов должен быть одинаковым на протяжении всего списка. Т.е. списки [1, 2, 3] и ['a', 'b',

‘c’] будут иметь тип [a], а список [1, ‘a’] будет другого типа. В этом случае в записи [a] символ a имеет значение типовой переменной.

Соглашения по именованию

В Haskell’е очень важны соглашения по именованию, ибо они явно входят в синтаксис языка (чего обычно нет в императивных языках). Самое важное соглашение — использование заглавной буквы в начале идентификатора. Имена типов, в том числе и определяемых разработчиком, должны начинаться с заглавной буквы. Имена функций, переменных и констант должны начинаться со строчной буквы. В качестве первого символа идентификатора также возможно использование некоторых специальных знаков, некоторые из которых также влияют на семантику идентификатора.

Определители списков и математические последовательности

Пожалуй, Haskell — это единственный язык программирования, который позволяет просто и быстро конструировать списки, основанные на какой-нибудь простой математической формуле. Этот подход уже был использован при построении функции быстрой сортировки списка методом Хоара (см. пример 3 в лекции 1). Наиболее общий вид определителей списков выглядит так:

```
[ x | x <- xs ]
```

Эта запись может быть прочитана как «Список из всех таких x, взятых из xs». Структура «x ← xs» называется генератором. После такого генератора (он должен быть один и стоять первым в записи определителя списка) может стоять некоторое число выражений охраны, разделённых запятыми. В этом случае выбираются все такие x, значения всех выражений охраны на которых истинно. Т.е. запись:

```
[ x | x <- xs, x > m, x < n ]
```

Можно прочитать как «Список из всех таких x, взятых из xs, что (x больше m) И (x меньше n)».

Другой важной особенностью Haskell’а является простая возможность формирования бесконечных списков и структур данных. Бесконечные списки можно формировать как на основе определителей списков, так и с помощью специальной нотации. Например, ниже показан бесконечный список, состоящий из последовательности натуральных чисел. Второй список представляет бесконечную последовательность нечётных натуральных чисел:

```
[1, 2 ..]
[1, 3 ..]
```

При помощи двух точек можно также определять любую арифметическую прогрессию, как конечную, так и бесконечную. Если последовательность конечна, то в ней задаются первый и последний элементы. Разность арифметической прогрессии вычисляется на основе первого и второго заданного элементов — в приведенных выше примерах разность в первой прогресси равна 1, а во второй — 2. Т.е. чтобы определить список всех нечётных натуральных чисел вплоть до 10, необходимо записать: [1, 3 .. 10]. Результатом будет список [1, 3, 5, 7, 9].

Бесконечные структуры данных можно определять на основе бесконечных

списков, а можно использовать механизм рекурсии. Рекурсия в данном случае используется как обращение к рекурсивным функциям. Третий способ создания бесконечных структур данных состоит в использовании бесконечных типов.

Пример 1. Определение типа для представления двоичных деревьев.

```
data Tree a          = Leaf a
                     | Branch (Tree a) (Tree a)

Branch               :: Tree a -> Tree a -> Tree a
Leaf                 :: a -> Tree a
```

В этом примере показан способ определения бесконечного типа. Видно, что без рекурсии тут не обошлось. Однако если нет необходимости создавать новый тип данных, бесконечную структуру можно получить при помощи функций:

```
ones            = 1 : ones
numbersFrom n = n : numberFrom (n + 1)
squares         = map (^2) (numbersFrom 0)
```

Первая функция определяет бесконечную последовательность, полностью состоящую из единиц. Вторая функция возвращает последовательность целых чисел, начиная с заданного. Третья возвращает бесконечную последовательность квадратов натуральных чисел вместе с нулем.

Вызовы функций

Математическая нотация вызова функции традиционно полагала заключение параметров вызова в скобки. Эту традицию впоследствии переняли практически все императивные языки. Однако в функциональных языках принята иная нотация — имя функции отделяется от её параметров просто пробелом. В Lisp'е вызов функции length с неким параметром L записывается в виде списка: (length L). Такая нотация объясняется тем, что большинство функций в функциональных языках каррированны.

В Haskell'е нет нужды обрамлять вызов функции в виде списка. Например, если определена функция, складывающая два числа:

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

То ее вызов с конкретными параметрами (например, 5 и 7) будет выглядеть как:

```
add 5 7
```

Здесь видно, что нотация Haskell'а наиболее сильно приближена к нотации абстрактного математического языка. Однако Haskell пошел еще дальше Lisp'а в этом вопросе, и в нем есть нотация для описания некаррированных функций, т.е. тип которых нельзя представить в виде $A_1 \to (A_2 \to \ldots (A_n \to B) \ldots)$. И эта нотация, как и в императивных языках программирования, использует круглые скобки:

```
add (x, y) = x + y
```

Можно видеть, что последняя запись — это функция с одним аргументом в строгой нотации Haskell'a. С другой стороны для каррированных функций вполне возможно делать частичное применение. Т.е. при вызове функции двух аргументов передать ей только один. Как показано в предыдущей лекции результатом такого вызова будет также функция. Более чётко этот процесс можно поиллюстрировать на примере функции inc, которая прибавляет единицу к заданному аргументу:

```
inc :: Integer -> Integer
inc = add 1
```

Т.е. в этом случае вызов функции inc с одним параметром просто приведет к вызову функции add с двумя, первый из которых — 1. Это интуитивное понимание понятия частичного применения. Для закрепления понимания можно рассмотреть классический пример — функция map (её определение на абстрактном функциональном языке приведено во второй лекции). Вот определение функции map на Haskell'e:

```
map                 :: (a -> b) -> [a] -> [b]
map f []            = []
map f (x:xs)        = (f x) : (map f xs)
```

Как видно, здесь использована инфиксная запись операции **prefix** — двоеточие, только такая запись используется в нотации Haskell'a для обозначения или конструирования пары. После приведенного выше определения можно произвести следующий вызов:

```
map (add 1) [1, 2, 3, 4]
```

Результатом которого будет список [2, 3, 4, 5].

Использование λ-исчисления

Т.к. функциональная парадигма программирования основана на λ-исчислении, то вполне закономерно, что все функциональные языки поддерживают нотацию для описания λ-абстракций. Haskell не обошел стороной и этот аспект, если есть необходимость в определении какой-либо функции через λ-абстракцию. Кроме того, через λ-абстракции можно определять анонимные функции (например, для единичного вызова). Ниже показан пример, где определены функции add и inc именно при помощи λ-исчисления.

Пример 2. Функции add и inc, определённые через λ-абстракции.

```
add = \x y -> x + y
inc = \x -> x + 1
```

Пример 3. Вызов аномимной функции.

```
cubes       = map (\x -> x * x * x) [0 ..]
```

Пример 3 показывает вызов анонимной функции, возводящей в куб переданный параметр. Результатом выполнения этой инструкции будет бесконечный список кубов целых чисел, начиная с нуля. Необходимо отметить, что в Haskell'e используется упрощенный способ записи λ-выражений, т.к. в точной нотации функцию add правильней было бы на-

писать как:

```
add = \x -> \y -> x + y
```

Остаётся отметить, что тип λ-абстракции определяется абсолютно так же, как и тип функций. Тип λ-выражения вида λx.expr будет выглядеть как $T_1 \rightarrow T_2$, где T_1 — это тип переменной x, а T_2 — тип выражения expr.

Инфиксный способ записи функций

Для некоторых функций возможен инфиксный способ записи, такие функции обычно представляют собой простые бинарные операции. Вот как, например, определены операции конкатенации списков и композиции функций:

Пример 4. Инфиксная операция конкатенации списков.

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

Пример 5. Инфиксная операция композиции функций.

```
(.)        :: (b -> c) -> (a -> b) -> (a -> c)
f . g      = \x -> f (g x)
```

Т.к. инфиксные операции всё-таки являются функциями в смысле Haskell’a, т.е. они каррированы, то имеет смысл обеспечить возможность частичного применения таких функций. Для этих целей имеется специальная запись, которая в Haskell’e носит название «секция»:

```
(x ++)     = \x -> (x ++ y)
(++ y)     = \y -> (x ++ y)
(++)       = \x y -> (x ++ y)
```

Выше показаны три секции, каждая из которых определяет инфиксную операцию конкатенации списков в соответствии с количеством переданных ей аргументов. Использование круглых скобок в записи секций является обязательным.

Если какая-либо функция принимает два параметра, то её также можно записывать в инфиксной форме. Однако если просто записать между параметрами имя функции — это будет ошибкой, т.к. в строгой нотации Haskell’a, это будет просто двойным применением, причем в одном применении не будет хватать одного операнда. Для того чтобы записать функцию в инфиксной форме, её имя необходимо заключить в символы обратного апострофа — `.

Для вновь определённых инфиксных операций возможно определение порядка вычисления. Для этого в Haskell’e есть зарезервированное слово infixr, которое назначает заданной операции степень её значимости (порядок выполнения) в интервале от 0 до 9, при этом 9 объявляется самой сильной степенью значимости (число 10 также входит в этот интервал, именно эту степень имеет операция применения). Вот так определяются степени для определенных в примерах 4 и 5 операций:

```
infixr 5 ++
```

infixr 9 .

Остается отметить, что в Haskell’e все функции являются нестрогими, т.е. все они поддерживают отложенные вычисления. Например, если какая-то функция определена как:

bot = bot

При вызове такой функции произойдет ошибка, и обычно такие ошибки сложно отслеживать. Но если есть некая константная функция, которая определена как:

constant_1 x = 1

То при вызове конструкции (constant_1 bot) никакой ошибки не произойдёт, т.к. значение функции bot в этом случае не вычислялось бы (вычисления отложенные, значение вычисляется только тогда, когда оно действительно требуется). Результатом вычисления естественно будет число 1.

ЗАДАНИЯ К КОНТРОЛЬНОЙ РАБОТЕ

Задание №1

Цель задания. Приобрести навыки работы с интерпретатором языка Haskell. Получить представление об основных типах языка Haskell.

Варианты заданий

Приведите пример (не менее 5 примеров) нетривиальных выражений, принадлежащих следующему типу:

1) ((Char,Integer), String, [Double])
2) [(Double,Bool,(String,Integer))]
3) ([Integer],[Double],[(Bool,Char)])
4) [[[(Integer,Bool)]]]
5) (((Char,Char),Char),[String])
6) (([Double],[Bool]),[Integer])
7) [Integer, (Integer,[Bool])]
8) (Bool,([Bool],[Integer]))
9) [([Bool],[Double])]
10) [([Integer], [Char]),(Integer,[Bool])]

Требование нетривиальности в данном случае означает, что встречающиеся в выражениях списки должны содержать больше одного элемента.

Задание №2

Цель задания. Научиться определять простейшие функции.

Варианты заданий

Определите следующие функции:

1) Функция max3, по трем целым возвращающая наибольшее из них.

2) Функция min3, по трем целым возвращающая наименьшее из них.
3) Функция sort2, по двум целым возвращающая пару, в которой наименьшее из них стоит на первом месте, а наибольшее — на втором.
4) Функция bothTrue : : Bool > Bool > Bool , которая возвращает True тогда и только тогда, когда оба ее аргумента будут равны True. Не используйте при определении функции стандартные логический операции (&&, | | и т.п.).
5) Функция solve2::Double->Double->(Bool,Double) , которая по двум числам, представляющим собой коэффициенты линейного уравнения ax + b = 0, возвращает пару, первый элемент которой равен True, если решение существует и False в противном случае; при этом второй элемент равен либо значению корня, либо 0.0.
6) Функция isParallel, возвращающая True, если два отрезка, концы которых задаются в аргументах функции, параллельны (или лежат на одной прямой). Например, значение выражения isParallel (1,1) (2,2) (2,0)

(4, 2) должно быть равно True, поскольку отрезки (1,1) − (2, 2) и (2, 0) − (4, 2) параллельны.
7) Функция is Included, аргументами которой служат параметры двух окружностей на плоскости (координаты центров и радиусы); функция возвращает True, если вторая окружность целиком содержится внутри первой.
8) Функция isRectangular, принимающая в качестве параметров координаты трех точек на плоскости, и возвращающая True, если образуемый ими треугольник — прямоугольный.
9) Функция is Triangle, определяющая, можно ли их отрезков с заданными длинами x, y и z построить треугольник.
10) Функция isSorted, принимающая на вход три числа и возвращающая True, если они упорядочены по возрастанию или по убыванию.

Задание №3

Цель задания. Научиться определять рекурсивные функции. Получить представление о механизме сопоставления с образцом.
Варианты заданий
Определите функцию, принимающую на вход целое число n и возвращающую список, содержащий n элементов, упорядоченных по возрастанию.
1) Список натуральных чисел.
2) Список нечетных натуральных чисел.
3) Список четных натуральных чисел.
4) Список квадратов натуральных чисел.
5) Список факториалов.
6) Список степеней двойки.
7) Список степеней тройки.
8)Список кубов натуральных чисел
9) Список треугольных чисел.
(*n*-е треугольное число t_n равно количеству одинаковых монет, из которых можно построить равносторонний треугольник, на каждой стороне которого укладывается *n* монет. Нетрудно убедиться, что $t_1 = 1$ и $t_n = n + t_{n-1}$)
10) Список пирамидальных чисел.
(*n-е* пирамидальное число p_n равно количеству одинаковых шаров, из которых можно построить правильную пирамиду с треугольным основанием, на каждой стороне которой укладывается *n* шаров. Нетрудно убедиться, что $p_1 = 1$ и $p_n = t_n + p_{n-1}$)
Построить функции, вычисляющие N-ый элемент следующих рядов.
1. $F(x, n) = x^n$
2. $F(x, n) = x^{n+3/n}$
3. $F(x, n) = (x+x^2)^n$
4. $F(n) = \Sigma_{i = 1,n}\ i$

5. $F(n) = \Sigma_{j=1,n} (\Sigma_{i=1,j}\ i)$
6. $F(n) = \Sigma_{i=1,p}\ n^{-i}$
7. $F(n) = 2^n$
8. $F(n) = 3^n$
9. $F(n) = e^n = \Sigma_{i=0,\infty} (n^i / i!)$
10. $F(n) = n^n$

Задание №4

Цель задания. Приобрести навыки определения функций для обработки списков.

Варианты заданий

Построить следующие бесконечные списки.

1) Список натуральных чисел.
2) Список нечетных натуральных чисел.
3) Список четных натуральных чисел.
4) Список квадратов натуральных чисел.
5) Список факториалов.
6) Список степеней двойки.
7) Список степеней тройки.
8)Список кубов натуральных чисел
9) Список треугольных чисел.

(*n*-е треугольное число t_n равно количеству одинаковых монет, из которых можно построить равносторонний треугольник, на каждой стороне которого укладывается *n* монет. Нетрудно убедиться, что $t_1 = 1$ и $t_n = n + t_{n-1}$)

10) Список пирамидальных чисел.

(*n-е* пирамидальное число p_n равно количеству одинаковых шаров, из которых можно построить правильную пирамиду с треугольным основанием, на каждой стороне которой укладывается *n* шаров. Нетрудно убедиться, что $p_1 = 1$ и $p_n = t_n + p_{n-1}$)

Определите следующие функции:

1) Функция, принимающая на входе список вещественных чисел и вычисляющую их арифметическое среднее. Постарайтесь, чтобы функция осуществляла только один проход по списку.
2) Функция вычленения n-го элемента из заданного списка.
3) Функция сложения элементов двух списков. Возвращает список, составленный из сумм элементов списков - параметров. Учесть, что переданные списки могут быть разной длины.
4) Функция перестановки местами соседних четных и нечетных элементов в заданном списке
5) Функция removeOdd, которая удаляет из заданного списка целых чисел все нечетные числа. Например: removeOdd [1,4,5,6,10] должен возвращать [4,10].
6) Функция removeEmpty, которая удаляет пустые строки из заданного

списка строк. Например:
removeEmpty ["", "Hello", "", "", "World!"] возвращает ["Hello", "World!"].
7) Функция countTrue : [Bool] > Integer , возвращающая количество элементов списка, равных True.
8)Функция makePositive, которая меняет знак всех отрицательных элементов списка чисел, например: makePositive [-1, 0, 5, -10, -20] дает [1,0,5,10,20]
9) Функция delete : : Char > String > String , которая принимает на вход строку и символ и возвращает строку, в которой удалены все вхождения символа. Пример: delete ' l' "Hello world! " должно возвращать "Heo word!".
10) Функция substitute :: Char > Char > String > String которая заменяет в строке указанный символ на заданный. Пример: substitute 'e' 'i' "eigenvalue" возвращает "iiginvalui"

Задание №5

Цель задания. Приобрести навыки обработки списков и строк.
Варианты заданий
Часть I

1. addStars :: [String] -> [String]
   Составить список из строк исходного списка, добавив в начало каждой строки списка символ '*' («звездочка»), если только строка уже не начинается с этого символа.
2. hasLetters :: [String] -> Int
   Определить количество строк в списке, содержащих хотя бы одну букву (буквой будем называть символ, для которого функция isAlpha :: Char -> Bool выдает значение True).
3. hasString :: [String] -> String -> Bool
   Определить, имеется ли в списке, заданном первым аргументом, строка, совпадающая со строкой, заданной вторым аргументом.
4. firstSymbols :: [String] -> String
   Составить строку из первых символов строк-элементов списка (пустые строки пропускать).
5. replaceToFirst :: [String] -> [String]
   По заданному списку строк составить новый список, содержащий строки, составленные из первых символов исходных строк. Пустые строки оставить без изменения.
6. removeLong :: [String] -> Int -> [String]
   По заданному списку строк (первый аргумент) составить новую строку,

содержащую те же строки, кроме строк, длина которых превышает значение, заданное вторым аргументом.

7. truncateList :: [String] -> Int -> [String]
   По заданному списку строк (первый аргумент) составить новую строку, содержащую те же строки, укороченные до длины, заданной вторым аргументом. Строки, длина которых меньше или равна второго аргумента, оставить без изменения.
8. longest :: [String] -> String
   Найти в списке самую длинную строку.
9. shortest :: [String] -> String
   Найти в списке самую короткую строку.
10. bracketsOnly :: [String] -> [String]
    Составить список из тех строк исходного списка, которые начинаются с символа '[' и заканчиваются символом ']'.

Часть II

1. removeStars :: [String] -> [String]
   Составить список из строк исходного списка, в который входят все строки, кроме строк, начинающихся с символа '*' («звездочка»).
2. startsWithDigit :: [String] -> Int
   Определить количество строк в списке, начинающихся с цифры (цифрой будем называть символ, для которого функция isDigit :: Char -> Bool выдает значение True).
3. notEmpty :: [String] -> Int
   Найти количество непустых строк в списке.
4. numLongs :: [String] -> Int -> Int
   Определить количество строк в списке, длина которых превышает значение второго аргумента.
5. allDigits :: [String] -> Int
   Определить количество строк в списке, состоящих только из цифр (цифрой будем называть символ, для которого функция isDigit :: Char -> Bool выдает значение True).
6. onlyLetters :: [String] -> [String]
   Составить список из строк исходного списка, состоящих только из букв (буквой будем называть символ, для которого функция isAlpha :: Char -> Bool выдает значение True).
7. lastSymbols :: [String] -> String
   Составить строку из последних символов строк-элементов списка (пустые строки пропускать).
8. removeShortest :: [String] -> [String]
   Составить список строк из строк исходного списка кроме самой

короткой строки. Если исходный список пустой, то и результат будет пустым списком; если строк с самой маленькой длиной несколько - удалить одну из них (любую).

9. remove :: [String] -> String -> [String]
   По заданному списку строк (первый аргумент) составить новую строку, содержащую те же строки, кроме строк, совпадающих со вторым аргументом.
10. removeLongest :: [String] -> [String]
    Составить список строк из строк исходного списка кроме самой длинной строки. Если исходный список пустой, то и результат будет пустым списком; если строк с самой большой длиной несколько - удалить одну из них (любую).

УЧЕБНО-МЕТОДИЧЕСКОЕ ОБЕСПЕЧЕНИЕ КУРСА

Рекомендуемая (основная) литература.

1. Методические указания к лабораторным работам по курсу "Функциональное программирование".
2. Электронный конспект лекций по курсу "Функциональное программирование".
3. John Harrison, Introduction to Functional Programming, http://www.cl.cam.ac.uk/Teaching/Lectures/funprog-jrh-1996/
4. Hal Daume, Yet Another Haskell Tutorial, http://www.isi.edu/~hdaume/htut/tutorial.pdf
5. Р. В. Душкин, Лабораторный практикум по функциональному программированию «Инструментальное средство HUGS 98 для программирования на языке Haskell», Московский инженерно-физический институт, Москва, 2001
6. School of Computer Science and Engineering
7. Jeroen Fokker, Functional Programming, Department of Computer Science, Utrecht University
8. Главная страница языка Haskell, http://haskell.org

Рекомендуемая (дополнительная) литература.

1. А.Филд, П.Харрисон. Функциональное программирование. М., "Мир", 1993, 637 с.
2. П.Хендерсон. Функциональное программирование. Применение и реализация. М., "Мир", 1983, 349 с.
3. Simon Peyton Jones (editor), Haskell 98 Language and Libraries (The Revised Report)
4. John Hughes, Introduction to Programming in Haskell, www.cs.chalmers.serjmh
5. Paul Hudak, John Peterson, Joseph H. Fasel ,A Gentle Introduction to Haskell 98
6. Cordelia Hall, John Hugs, The little Haskeller Philip Wadler, Monads for functional programming, Department of Computing Science, University of Glasgow
7. All About Monads , http://www.nomaware.com/monads/html/
8. Emery Berger , FP + OOP = Haskell, Department of Computer Science, The University of Texas at Austin
9. Rex Page, Two Dozen Short Lessons in Haskell, a participatory textbook on functional programming, School of Computer Science, University of Oklahoma
10. Damir Medak, Gerhard Navratil, Haskell-Tutorial, Institute for Geoinformation Technical University Vienna