Федеральное государственное бюджетное образовательное учреждение
высшего образования
«ПЕРМСКИЙ ГОСУДАРСТВЕННЫЙ
ГУМАНИТАРНО-ПЕДАГОГИЧЕСКИЙ УНИВЕРСИТЕТ»

ФАКУЛЬТЕТ ИНФОРМАТИКИ И ЭКОНОМИКИ

Кафедра информатики и вычислительной техники

Выпускная квалификационная работа

Дидактическое обеспечение раздела "Функциональное программирование" в дисциплине "Языки программирования"

Работу выполнил
студент группы М1221
направления подготовки
44.04.01 Педагогическое образование,
магистерская программа
«Информатика и ИКТ»
Иванов Сергей Владимирович

_____

«Допущен к защите в ГЭК»
Зав. кафедрой информатики и ВТ
Шестаков А.П.

_____

«____» июня 2017 г.

Руководитель:
к.п.н., доцент, зав. кафедрой
информатики и ВТ
Шестаков Александр Петрович

ПЕРМЬ
2017

Оглавление

Введение

В большинстве вузов на направлениях, связанных с информационными технологиями, не изучается другие парадигмы, кроме императивной и объектно-ориентированной, а если изучаются, то им уделяется слишком мало внимания, что в определенной степени сужает кругозор и программистские компетенции обучающихся.

Необходимо изучать несколько парадигм программирования по нескольким причинам: улучшается понимание конкретного языка, расширяется активный запас полезных конструкций, появляется возможность более обоснованно выбрать язык для решения конкретной задачи, облегчается освоение нового языка [12].

Кроме того, главной задачей обучения в педагогическом вузе является подготовка студентов для их дальнейшей работы в школе. Исходя из этого, отбор содержания курса информатики в педагогическом вузе должен производиться с учетом концепции школьного курса информатики и его целей.

Причины для изучения функциональной парадигмы.

Первая — это противоречие между многообразием парадигм программирования и однобокостью в изучении языка одной парадигмы программирования.

Вторая состоит в недостаточной математической подготовке будущих учителей информатики (в аспекте изучения математических оснований парадигм программирования и состоянием практики обучения этому разделу в педагогическом вузе). Методическая система фундаментальной подготовки в области информатики будущих учителей информатики, базируется, прежде всего, на достаточно серьезном внимании не только к алгоритмическим методам решения задач, но и изучению обоснования этих методов — теории рассматриваемого вопроса. Успешная реализация методической системы фундаментальной подготовки будущих учителей информатики невозможна

без серьезной математической подготовки, так как большинство теоретических разделов курса информатики должны излагаться с применением формального математического аппарата.

Третья — наличие профессиональной литературы по функциональному программированию, и почти полное отсутствие таковой для педагогического вуза.

Все это приводит к основной проблеме, корни которой лежат в противоречии между необходимостью повышения уровня профессиональной подготовки учителей информатики (в аспекте формирования знаний о функциональном программировании) и состоянием развития теории и практики обучения этому разделу в дисциплинах, ориентированных на программирование, в педагогических вузах. Таким образом, можно говорить о необходимости специального исследования, направленного на ее устранение.

Поэтому и стоит изучать и заниматься функциональным программированием (ФП). К тому же его изучение даст возможность посмотреть на эту деятельность с другой стороны и покажет другой стиль написания программ. Представляется *актуальным* реализация дидактического (методического) обеспечения для раздела дисциплины «Языки программирования».

Выделенная проблема определила *объект* данного исследования: обучение программированию в вузах.

Предмет исследования: обучение функциональному программированию в рамках дисциплины «Языки программирования».

Цель исследования: дидактическое обеспечение раздела «Функциональное программирование» дисциплины «Языки программирования» для студентов университетов.

Для достижения цели исследования должны быть решены следующие *задачи*:

1. проанализировать литературу по проблеме исследования;

2. проанализировать принципы функционального программирования;

3. сравнить языки программирования, включающие данный подход;

4. провести анализ учебных курсов по функциональному программированию;

5. сформировать комплект задач для лабораторных работ по функциональному программированию;

6. разработать методику обучения элементам функционального программирования в рамках дисциплины «Языки программирования».

Решение поставленных задач потребовало привлечение следующих *методов исследования*: изучение и теоретический анализ учебной и специальной литературы; систематизация и обобщение изученной литературы, и разработка учебно-методических материалов для изучения функционального программирования.

Научная новизна и теоретическая значимость исследования заключается:

1. в теоретическом обосновании целесообразности изучения функционального программирования;

2. в обосновании состава комплекта задач и разработке учебно-методического обеспечения раздела курса.

Практическая значимость исследования заключается в определении содержания теоретического материала и формировании комплекта задач, направленных на формирование специальных знаний и умений при изучении раздела дисциплины.

Структура и содержание работы соответствует логике научного исследования. Работа состоит из введения, двух глав, заключения и приложений.

В первой главе: изучена история развития функционального программирования, произведен анализ известных языков программирования

функциональной парадигмы, проанализирован опыт преподавания дисциплины в вузах и сети интернет, представлены результаты апробации.

Вторая глава содержит описание дидактического обеспечения раздела «Функциональное программирование».

В Приложении 1 представлен: фрагмент РПД «Основы функционального программирования». В приложении 2 представлено дидактическое обеспечение для раздела дисциплины «Языки программирования». Приложение 3 — скриншоты опубликованной статьи. Приложение 4 — диск с дидактическими материалами.

Глава 1. Функциональное программирование: история и развитие, подходы к преподаванию

1.1. История развития языков функционального программирования

Первым языком функционального программирования является Лисп. Автором Лиспа является Джон Маккарти, на период создания языка работавший в Массачусетском технологическом институте (MIT) в должности профессора по связи. Вместе с Марвином Мински он занимался работами по искусственному интеллекту, в связи с чем и возникла потребность в создании языка программирования, адекватного задачам, решаемым в этой области. Работа по созданию языка была проделана Д. Маккарти в MIT в период с 1958 по 1963 год, после чего он перешёл в Стэндфордский университет в Калифорнии, где получил должность «профессор по искусственному интеллекту».

Основой для Лиспа послужил ранний язык IPL, разработанный Ньэллом, Шоу и Саймоном. IPL был языком обработки списков и предназначался для реализации проекта «Логик-теоретик» — системы искусственного интеллекта, предназначенной для автоматического вывода теорем математической логики. IPL был довольно низкоуровневым языком, но в нём уже были реализованы такие базовые идеи, как единый механизм хранения программ и данных в виде списков — иерархических структур элементов, связанных ссылками (сама идея списочного представления знаний была позаимствована из исследований по психологии и ассоциативной памяти), а также идея динамического распределения памяти. После ознакомления в 1956 году с IPL у Джона Маккарти появилась идея реализовать обработку IPL-списков в Фортране, который как раз в это время проектировался в IBM (причём под ту же систему IBM 704, с которой Маккарти работал в MIT), но эта идея так и не была реализована. Позже Маккарти принял участие в работе «комитета по языку высокого уровня»,

разрабатывавшему Алгол, но и там его предложения были встречены холодно. В результате Д. Маккарти пришёл к мысли о необходимости создания нового языка программирования.

Первоначально Д. Маккарти сформулировал списочный формализм для описания данных S-выражения (символические выражения создаются из символов и списков) и основанный на нём же механизм описания лямбда-выражений, что позволило записывать программы в виде наборов функций, представленных в списочной форме. Как писал позже Д. Маккарти, изначально он планировал применять для записи программ отдельный формализм, отличающийся от S-выражений, но это оказалось излишним. Когда с помощью своей списочной записи Д. Маккарти описал алгоритм функционирования интерпретатора нового языка (формализм, который впоследствии стал известен как «Лисп на Лиспе»), заметили, что теперь для создания реального работающего интерпретатора достаточно просто перевести эту запись в машинный код. Д. Маккарти отнёсся к этой идее скептически, но проделал данную работу и получил первый интерпретатор Лиспа для компьютера IBM 704. В дальнейшем идея написания транслятора языка на нём самом многократно использовалась, и не только в функциональных и логических языках, но и в императивных.

Исторически первой реализацией Лиспа, включающей все современные базовые элементы языка, был интерпретатор, работавший на IBM 704, появившийся в октябре 1958 года. Это позволяет говорить о Лиспе как об одном из двух старейших языков высокого уровня, которые находятся в употреблении с момента создания до настоящего времени (первый — Фортран). Помимо этого, Лисп сохранил первенство ещё в одном отношении. Дело в том, что активная работа с динамическими списками сделала невозможным ручное управление памятью, которое в императивных языках отчасти сохраняется по сей день. Создание новых списочных ячеек и списков и выход из использования имеющихся при работе Лисп-программы происходят настолько активно, что практически невозможно обойтись без

системы автоматического управления памятью, которая контролировала бы использование ранее созданных в памяти объектов и периодически удаляла те из них, использование которых прекратилось, то есть системы сборки мусора. Д. Маккарти пришлось реализовать эту систему, благодаря чему Лисп, помимо прочего, является ещё и самым старым из применяемых сегодня языков программирования с автоматическим управлением памятью и сборкой мусора.

Позднее были созданы реализации для IBM 7090, в дальнейшем — для серий IBM 360 и 370. Компьютеры IBM оказались неудобны для работы в интерактивном режиме, вследствие чего в конце 1950-х годов небольшая группа разработчиков, в том числе работавших ранее в IBM, выделилась в самостоятельную компанию Digital Equipment Corporation (DEC). Первым её изделием стал компьютер PDP-1, изначально ориентированный на интерактивный режим работы. На этой машине в 1960 году была реализована интерактивная система «Lisp 1», включающая в себя интегрированные интерпретатор, редактор исходного кода и отладчик, позволявшая выполнять весь цикл работ над программой непосредственно в системе. По сути, это была первая «среда программирования» в том смысле, который вкладывается в это понятие сейчас. Тогда же в журнале «Communications of ACM» вышла статья Д. Маккарти «Recursive Functions of Symbolic Expressions and their Computation by Machine.», в которой Лисп был описан в виде алгебраического формализма на самом Лиспе. Статья стала классической, а формализм типа «Лисп на Лиспе» стал одним из наиболее употребимых в литературе по теории программирования. Ещё одним технологическим новшеством, появившимся в связи с реализацией системы «Lisp 1» был изобретённый Маккарти механизм, позволявший запускать интерпретатор Лиспа одновременно с выполнением обычных вычислительных работ в пакетном режиме (то, что сейчас известно, как «система разделения времени»).

К 1962 году была готова следующая версия оригинальной ЛИСП-системы «Lisp 1.5», в которой были устранены обнаруженные за время эксплуатации недостатки первой версии. Её описание было выпущено в издательстве «MIT Press» в виде отдельной книги. Поскольку руководство включало описание реализации системы, оно стало основой для создания ЛИСП-систем для множества других компьютеров, как в США, так и за её пределами. [9]

В нашей стране программирование соприкоснулось с Лиспом из первых рук. Джон Маккарти в конце 1968 года познакомил ученых из Москвы и Новосибирска с Лиспом, что побудило к реализации отечественных версий языка. Две реализации на БЭСМ-6 (ВЦ АН под рук. С.С. Лаврова и ВЦ СО АН под руководством А.П. Ершова) и одна на ЕС ЭВМ (ВЦ АН под рук. С.С. Лаврова) нашли применение в отечественных проектах по системному и теоретическому программированию, в исследованиях по математической лингвистике, искусственному интеллекту и обработке химических формул [6].

В 1980-е годы интерес к Лиспу в СССР сохранялся, тем не менее, литературы по языку издавалось очень мало (за десятилетие вышло две книги, обе переводные: «Функциональное программирование. Применение и реализация» Хендерсона, переведённая в 1983 году, и двухтомник «Мир Лиспа» Э. Хювёнена и Й. Сеппянена, перевод которой был издан в 1990). После распада СССР российское IT-сообщество достаточно быстро перешло на использование практически исключительно западной вычислительной техники и системного ПО. На сегодняшний день невозможно назвать ни одной ЛИСП-системы российского происхождения, которая находилась бы в эксплуатации.

Лисп нельзя назвать популярным или распространённым в России; его использование в основном ограничивается академическими исследованиями и работами отдельных энтузиастов. Кроме того, Лисп продолжает использоваться в учебных целях в некоторых российских университетах, но

и здесь в последние годы он оказался заметно потеснён: как язык общего назначения он не преподаётся и не используется, а в качестве учебных языков для преподавания функционального программирования часто предпочитают использовать более молодые функциональные языки, появившиеся в последние два десятилетия. Тем не менее, интерес к языку сохраняется, свидетельством чего является появление переводных и оригинальных печатных работ по Лиспу, возобновившееся в последние годы.

Идеи этого языка вызвали, не утихающие по сей день, дискуссии о приоритетах в программировании и сущности программирования. Лисп послужил эффективным инструментом экспериментальной поддержки теории программирования и развития сферы его применения. Рост интереса к Лиспу коррелирует с улучшением элементной базы, повышением эксплуатационных характеристик оборудования и появлением новых сфер применения ИТ.

Существует и активно применяется более трехсот диалектов Лиспа и родственных ему языков: Interlisp, muLisp, Clisp, Scheme, Cmucl, Logo, Hope, Sisal, Haskell, Miranda, HomeLisp и др.

Наиболее популярный сегодня диалект Common Lisp является универсальным языком программирования. Он широко используется в самых разных проектах: Интернет-серверы и службы, серверы приложений и клиенты, взаимодействующие с реляционными и объектными базами данных, научные расчёты и игровые программы.

Одно из направлений использования языка Лисп — его использование в качестве скриптового языка, автоматизирующего работу в ряде прикладных программ:

- язык Лисп используется как язык сценариев в САПР AutoCAD (диалект AutoLISP);
- его диалект — SKILL — используется для написания скриптов в САПР Virtuoso Platform компании Cadence Design Systems;

- язык Лисп является одним из базовых средств текстового редактора Emacs (диалект EmacsLISP);
- язык Лисп используется как язык сценариев в издательском программном обеспечении Interleaf/Quicksilver (диалект Interleaf Lisp);
- в оконном менеджере Sawfish применяется специальный диалект Лиспа Rep, который в значительной степени повторяет диалект Лиспа от Emacs;
- диалект Scheme используется в качестве одного из скриптовых языков в графическом процессоре Gimp;
- диалект GOAL используется для высоко динамичных трёхмерных игр;
- язык Лисп может использоваться для написания скриптов в аудио редакторе Audacity.

Сферы применения языка Лисп многообразны: наука и промышленность, образование и медицина, от декодирования генома человека до системы проектирования авиалайнеров.

Haskell. На конференции по функциональным языкам программирования и компьютерной архитектуре (FPCA, 1987) в Портленде (Орегон) участники пришли к соглашению, что должен быть создан комитет для определения открытого стандарта для подобных языков. Целью комитета являлось объединение существующих функциональных языков в один общий, который бы предоставлял базис для будущих исследований в разработке функциональных языков программирования.

Так появился Haskell. Он был назван в честь одного из основателей комбинаторной логики Хаскела Карри (Haskell Curry). Новый язык должен был стать свободным языком, пригодным для исследовательской деятельности и решения практических задач. Свободные языки основаны на стандарте, который формулируется комитетом разработчиков. Дальше любой желающий может заняться реализацией стандарта, написать компилятор

языка. Первая версия стандарта Haskell была опубликована 1 апреля 1990 года [4].

Ocaml. OCaml (**Objective Caml**) — современный объектно-ориентированный язык функционального программирования общего назначения, который был разработан с учётом безопасности исполнения и надёжности программ. Этот язык имеет высокую степень выразительности, что позволяет его легко выучить и использовать. Язык OCaml поддерживает функциональную, императивную и объектно-ориентированную парадигмы программирования. Был разработан в 1985 году во французском институте INRIA, который занимается исследованиями в области информатики. Самый распространённый в практической работе диалект языка ML.

ML (Meta Language) — семейство строгих языков функционального программирования с развитой параметрически полиморфной системой типов и параметризируемыми модулями.

Инструментарий OCaml включает в себя интерпретатор, компилятор в байткод и оптимизирующий компилятор в машинный код, сравнимый по эффективности с Java и лишь немного уступающий по быстродействию C и C++.

На языке OCaml, в частности, написан рендеринг формул Википедии, использующих тег <math>, файлообменный клиент MLDonkey.

F#. История F# началась в 2002 году, когда команда разработчиков из Microsoft Research под руководством Дона Сайма решила, что языки семейства ML вполне подходят для реализации функциональной парадигмы на платформе .NET. Идея разработки нового языка появилась во время работы над реализацией обобщённого программирования для Common Language Runtime. Известно, что одно время в качестве прототипа нового языка рассматривался Haskell, но из-за функциональной чистоты и более сложной системы типов потенциальный Haskell.NET не мог бы предоставить разработчикам простого механизма работы с библиотекой классов .NET Framework, а значит, не давал бы каких-то дополнительных преимуществ.

Как бы то ни было, за основу был взят OCaml, язык из семейства ML, который не является чисто функциональным и предоставляет возможности для императивного и объектно-ориентированного программирования. Однако, хоть Haskell и не стал непосредственно родителем нового языка, тем не менее, оказал на него некоторое влияние [14].

На сегодняшний день функциональное программирование занимает свою нишу среди парадигм программирования. Его используют в научных исследованиях и сложных математических вычислениях, веб-программирование, анализе данных, телекоммуникационном оборудовании, робототехнике, соцсетях (твиттер) и высоко-нагруженных системах.

1.2. Математические основы функционального программирования

Программы на традиционных языках программирования, таких как Си, Паскаль, Java и т.п. состоят их последовательности модификаций значений некоторого набора переменных, который называется *состоянием*. Если не рассматривать операции ввода-вывода, а также не учитывать того факта, что программа может работать непрерывно (т.е. без остановок, как в случае серверных программ), можно сделать следующую абстракцию. До начала выполнения программы состояние имеет некоторое начальное значение σ_0, в котором представлены входные значения программы. После завершения программы состояние имеет новое значение σ_0, включающее в себя то, что можно рассматривать как «результат» работы программы. Во время исполнения каждая команда изменяет состояние; следовательно, состояние проходит через некоторую конечную последовательность значений [1]:

$$\sigma = \sigma_0 \longrightarrow \sigma_1 \longrightarrow \sigma_2 \longrightarrow \cdots \longrightarrow \sigma_n = \sigma_0$$

Состояние модифицируется с помощью команд *присваивания*, записываемых в виде $v=E$ или v:=E, где v — переменная, а E — некоторое выражение. Эти команды следуют одна за другой; операторы, такие как if и while, позволяют изменить порядок выполнения этих команд в зависимости

от текущего значения состояния. Такой стиль программирования называют *императивным* или *процедурным*.

Функциональное программирование представляет парадигму, в корне отличную от представленной выше модели. Функциональная программа представляет собой некоторое выражение (в математическом смысле); выполнение программы означает вычисление значения этого выражения. С учетом приведенных выше обозначений, считая, что результат работы императивной программы полностью и однозначно определен ее входом, можно сказать, что финальное состояние (или любое промежуточное) представляет собой некоторую функцию (в математическом смысле) от начального состояния, т.е. $\sigma_0 = f(\sigma)$. В функциональном программировании используется именно эта точка зрения: программа представляет собой выражение, соответствующее функции *f*. Функциональные языки программирования поддерживают построение таких выражений, предоставляют широкий выбор соответствующих языковых конструкций.

Основы лямбда-исчисления

Подобно тому, как теория машин Тьюринга является основой императивных языков программирования, лямбда-исчисление служит базисом и математическим «фундаментом», на котором основаны все функциональные языки программирования.

Лямбда-исчисление было предложено в начале 30-х гг. логиком А. Черчем, который надеялся использовать его в качестве формализма для обоснования математики. Вскоре были обнаружены проблемы, делающие невозможным его использование в этом качестве (сейчас есть основания полагать, что это не совсем верно), и лямбда-исчисление осталось как один из способов формализации понятия алгоритма.

В настоящее время лямбда-исчисление является основной из таких формализаций, применяемой в исследованиях, связанных с языками программирования. Связано это, вероятно, со следующими факторами.

- Это единственная формализация, которая, хотя и с некоторыми неудобствами, действительно может быть непосредственно использована для написания программ.
- Лямбда-исчисление дает простую и естественную модель для таких важных понятий, как рекурсия и вложенные среды.
- Большинство конструкций традиционных языков программирования может быть более или менее непосредственно отображено в конструкции лямбда-исчисления.
- Функциональные языки являются в основном удобной формой синтаксической записи для конструкций различных вариантов лямбда исчисления. Некоторые современные языки (Haskell, Clean) имеют 100% соответствие своей семантики с семантикой подразумеваемых конструкций лямбда-исчисления.

Многие языки программирования также допускают определение функций только с присваиванием им некоторых имен. Например, в языке Си функция всегда должна иметь имя. Это кажется естественным, однако поскольку в функциональном программировании функции используются повсеместно, такой подход может привести к серьезным затруднениям.

Пусть $x = 2$ и $y = 4$. Тогда $x \cdot x = y$.

Лямбда-нотация позволяет определять функции с той же легкостью, что и другие математические объекты. *Лямбда-выражением* будем называть конструкцию вида

$$\lambda x.E$$

где E — некоторое выражение, возможно, использующее переменную x.

Пример. *λx.x2* представляет собой функцию, возводящую свой аргумент в квадрат.

Использование лямбда-нотации позволяет четко разделить случаи, когда под выражением вида *f(x)* мы понимаем саму функцию *f* и ее значение

в точке x. Кроме того, лямбда-нотация позволяет формализовать практически все виды математической нотации. Если начать с констант и переменных и строить выражения только с помощью лямбда-выражений и применений функции к аргументам, то можно представить очень сложные математические выражения.

Применение функции f к аргументу x обозначается как $f\ x$, т.е., в отличие от того, как это принято в математике, не используются скобки. По причинам, которые станут ясны позднее, будем считать, что применение функции к аргументу ассоциативно влево, т.е. $f\ x\ y$.

В качестве сокращения для выражений вида $\lambda x.\lambda y.E$ используется запись $\lambda x\ y.E$ (аналогично для большего числа аргументов). «Область действия» лямбда-выражения простирается вправо насколько возможно, т.е., например, $\lambda x.x\ y$ означает $\lambda x.(x\ y)$, а не $(\lambda x.x)y$.

Существует операция *каррирования*, позволяющая записать такие функции в обычной лямбда-нотации. Идея заключается в том, чтобы использовать выражения вида $\lambda x\ y.x + y$. Такое выражение можно рассматривать как функцию $R \rightarrow (R \rightarrow R)$, т.е. если его применить к одному аргументу, результатом будет функция, которая затем принимает другой аргумент. Таким образом:

$$(\lambda x\ y.x + y)\ 1\ 2 = (\lambda y.1 + y)\ 2 = 1 + 2.$$

Переменные в лямбда-выражениях могут быть *свободными связанными*. В выражении вида x^2+2x+2 переменная x является свободной; его значение зависит от значения переменной x и в общем случае ее нельзя переименовать. Однако в выражении переменная x является связанной: если вместо x везде использовать обозначение z, значение выражения не изменится.

Следует понимать, что в каком-либо подвыражении переменная может быть свободной (как в выражении под интегралом), однако во всем выражении она связана какой-либо операцией связывания переменной. Та

часть выражения, которая находится «внутри» операции связывания, называется *областью видимости* переменной.

В лямбда исчислении выражения $\lambda x.E[x]$ и $\lambda y.E[y]$ считаются эквивалентными (это называется α-эквивалентностью, и процесс преобразования между такими парами называютα-преобразованием). Разумеется, необходимо наложить условие, что y не является свободной переменной в $E[x]$.

Лямбда-исчисление основано на формальной нотации лямбда-терма, составляемого из переменных и некоторого фиксированного набора констант с использованием операции применения функции и лямбда-абстрагирования. Сказанное означает, что все лямбда-выражения можно разделить на четыре категории:

1. **Переменные:** обозначаются произвольными строками, составленными из букв и цифр.

2. **Константы:** также обозначаются строками; отличие от переменных будем определять из контекста.

3. **Комбинации:** т.е. применения функции S к аргументу T; и S и T могут быть произвольными лямбда-термами.

4. **Абстракции** произвольного лямбда-терма S по переменной x, обозначаемые как $\lambda x.S$.

Таким образом, лямбда-терм определяется рекурсивно и его грамматику можно определить в виде следующей формы Бэкуса-Наура:

$$Exp = Var | Const | Exp\ Exp | \lambda\ Var\ .\ Exp$$

В соответствие с этой грамматикой лямбда-термы представляются в виде синтаксических деревьев, а не в виде последовательности символов. Отсюда следует, что соглашения об ассоциативности операции применения функции, эквивалентность выражений вида $\lambda x\ y.S$ и $\lambda x.\lambda y.S$, неоднозначность в именах констант и переменных проистекают только из необходимости представления лямбда-термов в удобном человеку виде, и не являются частью формальной системы.

Лямбда-исчисление описывает все операции, используемые в функциональных языках программирования, и действия с лямбда-термами.

Булевские значения и условия

Для кодирования значений true и false можно использовать любые неравные лямбда-термы, но лучше всего определить их следующим образом:

```
true , λx y.x
false , λx y.y
```

Используя эти определения, можно легко определить условное выражение (напоминающее конструкцию ?:) языка Си. Это — условное *выражение*, а не условная *команда*, следовательно, else-часть обязательна:

```
if E then E1 else E2 , E E1 E2.
```

Действительно:

```
if true then E1 else E2 , true E1 E2
,(λx y.x) E1 E2
,E1
```

и

```
if false then E1 else E2 , true E1 E2
,(λx y.y) E1 E2
,E2
```

Теперь легко определить все обычные логические операторы:

```
not p , if p then false else true
p and q , if p then q else false
p or q , if p then true else q
```

Пары и кортежи

Упорядоченную пару можно представить следующим образом:

```
(E1, E2) , λf.f E1 E2
```

Скобки здесь необязательны, но можно использовать их для того, чтобы вызывать аналогии с соответствующей записью для обозначения векторов или комплексных чисел в математике. В действительности запятую можно рассматривать просто как некоторый инфиксный оператор, как + или −. С учетом данного определения, функции для извлечения компонентов пары можно записать так:

```
fst p , p true
snd p , p false
```

Определения удовлетворяют требованиям:

```
fst(p, q) = (p, q) true =
=(λf.f p q) true =
=true p q =
=(λx y.x) p q =
=p
```

и

```
snd(p, q) = (p, q) false =
=(λf.f p q) false =
=false p q =
=(λx y.y) p q =
=q
```

Тройки, четверки и произвольные n-кортежи можно построить с помощью пар:

```
(E1, E2, . . . , En) = (E1, (E2, . . . , En)).
```

Следует только ввести соглашения, что оператор «запятая» ассоциативен вправо.

Функции fst и snd можно расширить на случай *n*-кортежей. Функция селектора, которая получает *i-й* компонент кортежа *p*. Записывается как *(p)i*. Тогда *(p)1 = fst p* и *(p)i = fst(sndi−1 p).*

Пример нахождения факториала с помощью рекурсии.

```
f(n) = 1, if n = 0; else n × f(n - 1).
```

В лямбда-исчислении, функция не может непосредственно ссылаться на себя. Тем не менее, функции может быть передан параметр, связанный с ней. Как правило, этот аргумент стоит на первом месте. Связав его с функцией, мы получаем новую, уже рекурсивную функцию. Для этого аргумент, ссылающийся на себя (здесь обозначен как r), обязательно должен быть передан в тело функции.

```
g := λr. λn.(1, if n = 0; else n × (r r (n-1)))
                    f := g g
```

Функциональное программирование предлагает обходиться вычислением результатов функций от исходных данных и результатов других функций, и не предполагает явного хранения состояния программы. Соответственно, не предполагает оно и изменяемость этого состояния (в отличие от императивного, где одной из базовых концепций является **переменная**, *хранящая* своё значение и позволяющая *менять* его по мере выполнения алгоритма) [1].

Существуют различия в понимании функции в математике и функции в программировании, вследствие чего нельзя отнести императивные языки к функциональным, использующим менее строгое понятие. Функция в математике не может изменить вызывающее её окружение и запомнить результаты своей работы, а только предоставляет результат вычисления функции.

Программирование с использованием математического понятия функции вызывает некоторые трудности, поэтому функциональные языки в той или иной степени предоставляют и императивные возможности, что ухудшает дизайн программы (например, возможность безболезненных дальнейших изменений).

Дополнительное отличие от императивных языков программирования заключается в декларативности. Тексты программ на функциональных языках программирования описывают «как решить задачу», но не предписывают последовательность действий для решения.

При сравнении функционального и императивного подхода к программированию можно заметить следующие свойства функциональных программ.

- Функциональные программы не используют переменные в том смысле, в котором это слово употребляется в императивном программировании. В частности, в функциональных программах не используется оператор присваивания.

- Как следствие из предыдущего пункта, в чисто функциональных программах нет циклов.
- Выполнение последовательности команд в функциональной программе бессмысленно, поскольку одна команда не может повлиять на выполнение следующей.
- Функциональные программы используют функции гораздо более замысловатыми способами. Функции можно передавать в другие функции в качестве аргументов и возвращать в качестве результата, и даже в общем случае проводить вычисления, результатом которого будет функция.
- Вместо циклов функциональные программы широко используют рекурсивные функции.
- Программы на функциональных языках обычно намного короче и проще, чем те же самые программы на императивных языках.

Пример (быстрая сортировка Хоара на абстрактном функциональном языке):

```
quickSort ([]) = []
quickSort ([h : t]) = quickSort ([n | n ∈ t, n <= h]) +
[h] + quickSort ([n | n ∈ t, n > h])
```

- В функциональных языках большая часть ошибок может быть исправлена на стадии компиляции, поэтому стадия отладки и общее время разработки программ сокращаются. Вдобавок к этому строгая типизация позволяет компилятору генерировать более эффективный код и тем самым ускорять выполнение программ.
- Механизм модульности позволяет разделять программы на несколько сравнительно независимых частей (модулей) с чётко определёнными связями между ними. Тем самым облегчается процесс проектирования и последующей поддержки больших программных систем. Поддержка модульности не является свойством именно функциональных языков программирования, однако поддерживается большинством таких языков.

- В чистом функциональном программировании объекты нельзя изменять и уничтожать, можно только создавать новые путем декомпозиции и синтеза существующих. О ненужных объектах позаботится встроенный в язык сборщик мусора. Благодаря этому в чистых функциональных языках все функции свободны от побочных эффектов [6].

В императивных языках программирования (например, C++) вызов функции приводит к вычислению всех аргументов. Этот метод вызова функции называется вызов-по-значению. Если какой-либо аргумент не использовался в функции, то результат вычислений пропадает, следовательно, вычисления были произведены впустую. Противоположностью вызова-по-значению является вызов-по-необходимости (ленивые вычисления). В этом случае аргумент вычисляется, только если он нужен для вычисления результата.

Лямбда-исчисление служит базисом и математическим «фундаментом», на котором основаны все функциональные языки программирования.

В настоящее время лямбда-исчисление является основой в исследованиях, связанных с функциональными языками программирования. Связано это, вероятно, со следующими факторами.

- Это единственная формализация, которая, хотя и с некоторыми неудобствами, действительно может быть непосредственно использована для написания программ.
- Лямбда-исчисление дает простую и естественную модель для таких важных понятий, как рекурсия и вложенные среды.
- Большинство конструкций традиционных языков программирования может быть более или менее непосредственно отображено в конструкции лямбда-исчисления.
- Функциональные языки являются в основном удобной формой синтаксической записи для конструкций различных вариантов лямбда исчисления.

1.3. Сравнение языков программирования

F# и PYTHON

В настоящее время существует большое количество разнообразных: по возможностям и функциям языков из данной парадигмы. Самыми современными и часто обсуждаемыми и набирающими популярность среди программистов-функциональщиков являются F# и Python. Эти языки были выбраны для сравнения, чтобы выявить сильные и слабые стороны каждого. Сразу следует заметить, что хоть они и в чем-то похожи друг на друга, но все-таки у них разные задачи и области применения.

Их объединяет несколько черт.

Во-первых, они обладают мультипарадигменностью т.е. совмещают в себе императивную, объектно-ориентированную и функциональную парадигмы.

Во-вторых, на том и на другом можно создавать не только отдельные приложения, но и сайты, использовать в серверной и клиентской частях.

В-третьих, оба присутствуют в Visual Studio, что дает возможность пользоваться библиотеками .NET и функциями этой IDE.

В-четвертых, в том и в другом получается намного короче программный код по сравнению с объектно-ориентированными языками.

В-пятых, считается, что из-за своеобразного упрощения синтаксиса код на данных языках удобочитателен и лаконичен.

В-шестых, присутствует сборка мусора.

Отличия.

— Python старше (1989 г.), поэтому он более распространен. F# впервые реализован в 2005 г., но, скорее, следует выделить вторую дату — 2010 г., когда его интегрировали в Visual Studio 2010 [16].

— F# разрабатывает и продвигает одна фирма (MS), поэтому принят общий стандарт этого языка. У Python активно развивается — новые версии выходят примерно раз в два года, поэтому у него нет общего стандарта.

— F# входит в категорию Open Source т.е. можно просматривать и изменять исходный код библиотек. На нем пишутся приложения, создаются библиотеки, сайты (большой проект Web sharper), используется в серверной и клиентской части, применяется для программирования роботов, можно запустить mono.

Python используется прежде всего в сферах.

- связанных с интернет-технологиями: поисковой системе Google, YouTube, веб-фреймворке App Engine, BitTorrent.
- для тестирования программного обеспечения: Intel, Cisco, HP, IBM. И для написании логики приложений - часто в играх [17].

— Программы на Python запускаются сразу, потому что отсутствует стадия компиляции и связывания, необходимая для F#. Что вызывает некоторые удобства при работе.

— F# обладает высокой производительностью, строгой типизацией и поддержкой параллельных вычислений. Python обладает не высокой производительностью и не полной поддержкой параллельных вычислений, отсутствует строгая типизация.

По данным сайта TIOBE за 2015-1016: Python занимает 5-е место, а F# - 29-е, что говорит о большей популярности Python, но среди функциональных языков он идет сразу после Lisp и занимает соответственно 3-е место. (TIOBE занимается анализом запросов в сети интернет языков программирования и предоставляет статистику для всеобщего ознакомления) [7].

Как видно из таблицы 1 «Сравнение языков», F# интегрирован в небольшое количество IDE по сравнению со своим соперником и к тому же на Python можно писать в текстовых редакторах.

Таблица 1
Сравнение языков

	F#	Python
Дата создания	2005 г.	1989 г.
Продвигающая фирма	MS	нет определенной фирмы
IDE	Visual Studio, Sharp Develop, Xamarin Studio, mono	netBeans, Eclipse + PyDev, Visual Studio + Python Tools, PyCharm, Sublime Text, NotePad++
Производительность	высокая	отстает от F#
Типизация	строгая	не строгая
Поддержка html	Да	Нет
Автозаполнение	Да	Да
Поддержка параллельных вычислений	полная	не полная
Литература	на русском языке очень мало книг	намного больше
Библиотеки	к каждой есть своя документация	из-за различных, продвигающих этот язык фирм, происходит путаница. Есть библиотеки, но есть сложность с поиском документации.

Возможная область применения F# гораздо больше, чем у Python и при его изучении можно узнать много нового и полезного к тому же программировать на нем намного интереснее, но требуется больше времени для его изучения.

При выборе языка для изучения необходимо отталкиваться от двух вопросов «Для чего?» и «Что хотите?» и следует помнить, что:

- Python можно быстро выучить и начать им пользоваться;
- F# более серьезнее, позволяет познать все тонкости функциональных языков [16].

F# и Lisp

Lisp является самым первый функциональным языком программирования, а F# — современным.

Их объединяет несколько черт:

- оба можно отнести к функциональной парадигме;
- современные диалекты Lisp можно отнести к языкам, обладающих мультипарадигмальностью, как и F#;
- предназначены для обработки больших объемов данных;

Различия:

- для определения вложенности блоков в Lisp используются скобки, что затрудняет понимание кода. В F# вложенность определяется отступом;
- F# базируется на .Net, что дает возможность пользоваться всеми функциями платформы;
- у Lisp множество диалектов, которые служат разным целям;
- в Lisp отсутствуют правила приоритета;
- Lisp используется в областях искусственного интеллекта.

По данным сайта TIOBE Lisp занимает 33 место. F# на 29 месте.

В таблице 2 «Сравнение языков» сравниваются два языка по некоторым пунктам.

Таблица 2
Сравнение языков

	F#	Lisp
Дата создания	2005 г.	1958 г.
Продвигающая фирма	MS	нет определенной фирмы, разработал Д.Маккарти.
IDE	Visual Studio, Sharp Develop, Xamarin Studio, mono	LispWorks IDE, Allegro, Common Lisp, Emacs + Slime, Lispbox, CLISP, Clozure CL и др.
Производительность	высокая	высокая
Типизация	строгая	динамическая
Поддержка html	Да	Нет

Автозаполнение	Да	Нет
Поддержка параллельных вычислений	полная	В зависимости от диалекта
Литература	на русском языке очень мало книг	Большое количество книг и статей на различных языках
Библиотеки	к каждой есть своя документация	к каждому диалекту есть свои расширения, обладающие документацией

Хотя Lisp создан давно, его диалекты продолжают развиваться и используются по сей день в различных ИТ сферах. В некоторые диалекты была добавлена поддержка ООП и параллельного программирования. Как следствие, появились диалекты, обладающие мультипарадигменностью [19].

1.4. Анализ учебных курсов

Функциональное программирование преподается в крупных вузах страны: МГУ, СПбГУ ИТМО, НГУ, ПГНИУ, МИФИ и т.д. В некоторых вузах оно преподается уже более 10 лет.

Изучение происходит на различных языках, реализующих данную парадигму. В большинстве случаев это Lisp и F#.

Функциональное программирование в Государственном университете - Высшей школе экономики (Автор Д.В. Сошников)

Курс предназначен для направления «Бизнес-информатика»

Дисциплина читается студентам бакалавриата отделения программной инженерии факультета бизнес-информатики ГУ-ВШЭ. Она входит в блок специальных дисциплин по выбору, и читается в первом и втором модулях четвертого учебного года. Продолжительность курса составляет 40 аудиторных учебных часов, в том числе: 20 часов лекционных занятий, 20 часов практических занятий, и 98 часов самостоятельной работы. Всего 138 часов. Рубежный контроль – отчёты по лабораторным работам и письменный зачёт по окончанию 2-го модуля.

Курс знакомит слушателей с парадигмой функционального программирования, в которой решение задач сводится к описанию функций,

перерабатывающих некоторые входные данные в выходные и строящихся из более простых функций на основе принципов функциональной абстракции и аппликации (см. табл. 3). Рассматриваются теоретические основы функционального программирования (лямбда-исчисление, комбинаторная логика), на примере функционального подхода дается представление о некоторых теоретических разделах компьютерных наук (семантика языков программирования, доказательство программ). С другой стороны, курс содержит значительную практическую составляющую, основанную на промышленном языке программирования F# (входит в состав Microsoft Visual Studio 2010), рассматриваются вопросы использования функциональных языков для построения компиляторов, грамматического разбора, для анализа финансовых показателей и т.д. [18]

Таблица 3
Тематическое планирование

Название темы	Всего часов по дисциплине	Аудиторные часы		Самостоятельная работа
		Лекции	Практические занятия	
Первый модуль (20 часов)				
Тема 1. Введение в функциональное программирование	16	4	2	10
Тема 2. Рекурсивные структуры данных – списки и деревья	30	4	6	20
Тема 3. Лямбда-исчисление как алгоритмическая модель	14	2	2	10
Второй модуль (20 часов)				
Тема 4. Лямбда-исчисление как язык программирования	12	2	0	10
Тема 5. Важные приёмы функционального программирования	18	2	4	12
Тема 6. Метапрограммирование, асинхронное и параллельное программирование	16	2	2	12
Тема 7. Объектно-ориентированные и императивные элементы языка F#.	16	2	2	12
Тема 8. Функциональные аспекты современных языков программирования.	16	2	2	12
Итого	**138**	**20**	**20**	**98**

«Функциональное и логическое программирование» в Армавирском Институте прикладной информатики, математики и физики

Курс предназначен для направления «Информатика и вычислительная техника».

В рамках дисциплины изучается функциональная и логическая парадигмы программирования. Курс “Функциональное и логическое программирование” должен расширить представления будущего специалиста о возможностях вычислительной техники, сферах ее применения, показать наиболее перспективные направления развития информатизации общества.

Основной целью дисциплины является формирование и закрепление системного подхода при разработке программ с применением языков логического и функционального программирования, в дисциплине рассматриваются средства и методы создания таких программ.

Ядро дисциплины составляют средства и приемы создания программ с использованием языков логического и функционального программирования.

Обучение функциональной парадигме происходит на языке Lisp. Его изучению предоставляется 24 часа, из 12 аудиторных часов: 6 часов лекций и 6 часов лабораторных работ (см. табл. 4). Самостоятельной работе 12 часов. Остальное время отводится для изучения логическому программированию [10].

Таблица 4
Тематическое планирование

Раздел, тема	Всего часов	В т.ч. аудиторных, час			Самост. работа, час
		Всего аудит.	Из них		
			Лекции	Лаб. работы	
Тема 1. Введение. Понятие декларативного программирования	4	2	2		2
Тема 2. Язык логического программирования Пролог. Общие сведения о Прологе	4	2	2		2

Тема 3. Представление знаний о предметной области в виде фактов и правил базы знаний Пролога.	8	6	2	4	2
Тема 4. Поиск с возвратом. Управление поиском	8	6	2	4	2
Тема 5. Арифметические вычисления.	8	6	2	4	2
Тема 6. Рекурсия в Прологе	8	6	2	4	2
Тема 7. Списки. Операции со списками.	8	6	2	4	2
Тема 8. Программирование баз данных Динамические базы данных.	10	8	2	6	2
Тема 9. Обработка строк.	8	6	2	4	2
Тема 10. Применение логического программирования в задачах искусственного интеллекта	10	6	2	4	4
Тема 11. Технология визуального программирования в среде Visual Prolog.	8	6	2	4	2
Тема 12. Основы функционального программирования. Общие сведения о языке LISP	4	2	2	0	2
Тема 13. Приемы программирования; представление и интерпретация функциональных программ.	8	4	2	2	4
Тема 14. Рекурсия в LISP.	12	6	2	4	6
Итого:	144	72	28	44	36

Функциональное программирование и интеллектуальные системы в Дагестанском государственном университете

Курс предназначен для направления «Бизнес-информатика»

Дисциплина «Функциональное программирование и интеллектуальные системы» входит в базовую часть профессионального цикла образовательной программы бакалавриата по направлению «Бизнес-информатика». Дисциплина реализуется кафедрой «Математическое моделирование,

эконометрика и статистика» на факультете управления. Дисциплина «Функциональное программирование и интеллектуальные системы» направлена на изучение теоретических основ и получении практических навыков разработки программных систем с использованием функционального подхода к программированию.

Преподавание дисциплины предусматривает проведение следующих видов учебных занятий: лекции, практические занятия, самостоятельная работа. Рабочая программа дисциплины предусматривает проведение следующих видов контроля: текущий контроль успеваемости в форме опросов, рефератов, дискуссий, тестов, решения задач и промежуточный контроль в форме зачета. Объем дисциплины 3 зачетных единиц, в том числе в академических часах 108 ч. по видам учебных занятий зачет. На изучение функционального программирование отводится: лекции 22 часа, практические занятия 28 часов и 58 часов самостоятельная работа (см. табл. 5) [13].

Таблица 5
Тематическое планирование

Раздел, тема	Всего часов по дисциплине	Аудиторные часы		Самостоятельная работа
		Лекции	Практические занятия	
Тема 1.1. Предмет и задачи курса. Эволюция информационных систем	10	2	2	6
Тема 1.2. Понятие ИИТ, основные свойства. Области применения ИИТ и классификация интеллектуальных информационных систем.	14	2	4	8
Тема 1.3. Традиционные способы представления и обработки знаний в интеллектуальных системах.	12	2	2	8
Тема 2.1. Архитектура ИИС. Проектирование ИИС. Методы приобретения знаний.	12	2	4	6
Тема 2.2. Определение и краткая история	10	2	2	6

функционального программирования.				
Тема 2.3. Основы функционального программирования на F#.	14	4	4	6
Тема 3.1. Рекурсивные структуры данных. Списки. Примеры работы со списками.	12	2	4	6
Тема 3.2. Типовые приемы функционального программирования.	10	2	2	6
Тема 3.3. Императивные и объектно- ориентированные возможности F#. Параллельное и асинхронное программирование.	14	4	4	6
Итого	108	22	28	58

Задачи дисциплины: выработка у студентов системного подхода к решению задач инженерии знаний, навыков использования методов функционального программирования, способности ориентироваться в многообразии методов построения интеллектуальных информационных систем, их классификации. Рассмотреть особенности различных парадигм программирования, сравнить возможности процедурного и функционального программирования для решения различных классов задач. Освоить приёмы функционального программирования и дать навыки разработки приложений на языке F#.

Функциональное программирование в Кембридже

В Кембридже существует несколько курсов изучающие функциональную парадигму. Они обучают основам функционального программирования, используя языки Haskell, Scala и F#. Главная цель - познакомить студентов с фундаментальными понятиями программирования, как рекурсия, абстракции, функции высшего порядка и типы данных, подчеркивая практическое использование этих конструкций и применение в графической среде.

Курс состоит из лекции:

1. Введение в функциональное программирование.
2. Списки, рекурсия.
3. Рекурсии.
4. Class test: Алгебраические типы данных.
5. Деревья выражений, лямбда.
6. Абстрактные типы.
7. Классы типов.
8. IO и монады.
9. Пробный экзамен: Логика и программы.

Общее количество часов, отводящихся под курс, 100. Лекций 20 часов, практических занятий 37 часов, и 43 часов отводится на самостоятельную работу. Самостоятельной работе уделяется много часов. В практическую часть входят не только лабораторные работы, но и семинары. На семинарах выступают докладчики, и ведется обсуждение качества кода, некоторой задачи. Перед лекциями обучающиеся должны прочитать необходимый материал и ответить на вопросы в интернет-системе. Лекции записываются на видео и выкладываются в доступную сеть [8].

Обучающиеся могут принять участие в обучении других и/или младших курсов, за что будут получать дополнительные баллы. В конце курса обучающиеся представляют свой проект в рамках курсовой работы. Чаще всего выполненный в парах на соответствующем языке.

Весовые коэффициенты для компонентов курса:

Компонент	**Проценты**
Лаборатории, обзоры, обзор кода, обучение других	10%
Наборы задач	50%
Последний проект	15%
Промежуточный экзамен	10%
Итоговый экзамен	15%
Всего	***100%***

Интернет-курсы.

В сети интернет существует большое количество курсов, обучающих данной парадигме. Они знакомят с различными функциональными языками. Самые крупные из них представлены в таблице 6 «Интернет курсы».

Таблица 6
Интернет курсы

	Сайт курса	Название курса	Язык
1	Открытое образование	Функциональное программирование: базовый курс	Lisp
2	Лекториум	Функциональное программирование	Scheme
3	Интуит	Академия Microsoft: Функциональное программирование	F#
4	Coursera	Функциональное программирование на языке Scala	Scala
5	Twirpx	Функциональное программирование	Haskell

Один из самых известных курсов изучения функционального программирования на языке F#, является курс от академии Microsoft, который преподает Дмитрий Сошников. Курс ведется в МГУ и СПбГУ. Есть интернет версия курса, который можно просмотреть на сайте Интуит. Курс состоит из 30 лекций, различной продолжительности, нескольких тестов и заданий. Его могут посмотреть и пройти все желающие, получив за это сертификат. Он объемный по содержанию, включает в себя историю, основные возможности функциональных языков и монады и асинхронные и параллельные вычисления.

Как в вузах, так и в интернет-курсах рассматриваются основные темы функциональной парадигмы: функции высших порядков, рекурсия, замыкания и лямбда-выражения, ленивые вычисления, абстрактные типы данных, свертки.

За рубежом функциональное программирование популяризовано в технических учебных заведениях США и Европы, поэтому большое количество литературы по функциональному программированию пишется на английском языке. Только на сайте Гарварда расположено 3690 статей и документов по функциональному программированию.

Глава 2. Разработка дидактического обеспечения раздела «Функциональное программирование»

2.1. Раздел «Функциональное программирование» в рабочей программе дисциплины «Языки программирования»

Целью изучения раздела «Функциональное программирование» является овладение обучающимися основными элементами языка программирования в рамках функциональной парадигмы. Для достижения цели решаются *задачи:*

- знакомство обучающихся с языком программирования F#;
- овладение основами функциональных возможностей данного языка;
- умение применять полученные знания в решении типовых задач.

По завершении изучения раздела обучающийся должен.

Знать:

- основные возможности функциональной парадигмы;
- основные конструкции языка F#;
- правила записи программ.

Уметь:

- использовать конструкции данного языка;
- решать типовые задачи;
- выделять подзадачи и строить функции для их решения;
- осуществлять отладку и тестирование программ на F#.

Владеть:

- навыками составления функции;
- навыками решения типовых задач.

Трудоемкость раздела составляет 40 часов, из которых 6 часов лекции, 10 часов практических и 24 часа отводиться под самостоятельную работу.

В ходе курса изучаются следующие темы:

1. Основы языка F#.
2. Списки в F#.
3. Последовательности в F#.
4. Деревья в F#.

Зачетом являются все выполненные практические задания.

Распределение часов по темам и видам учебной нагрузки представлено в таблице 7.

Таблица 7
Тематическое планирование

Тема	Всего часов	Лекции	Лабораторные работы	Самостоятельная Работа
1. Основы языка F# 1.1. Базовые операции и операторы 1.2. Функциональные возможности	4	2	2	6
2. Списки в F# 2.1. Генерация списков 2.2. Обработка списков	3	1	2	6
3. Последовательности в F# 3.1. Работа с последовательностями	3	1	2	6
4. Деревья в F# 4.1. Формирование, виды обходов 4.2. Работа с деревьями	6	2	4	6
Итого 40 часов				

Рассмотрим краткое содержание каждой темы.

Тема 1. Основы языка F#. Краткая история функционального программирования. Современное состояние функционального программирования. Язык программирования F#. Основные конструкции и синтаксис языка. Выполнение практических заданий «Основы языка F#».

Тема 2. Списки в F#. Понятие списка. Основные возможности и функции работы над списками. Генерация и обработка списков. Выполнение практических заданий «Списки в F#».

Тема 3. Последовательности в F#. Понятие последовательности. Отличие от списков. Основные возможности и функции работы над последовательностями. Выполнение практических заданий «Последовательности в F#».

Тема 4. Деревья в F#. Понятие дерева. Назначение и применение. Формирование и виды обхода. Выполнение практических заданий «Деревья в F#».

Таблица 8
Лабораторные работы

Название тем	Цель и содержание лабораторной работы
Лабораторная работа №1: Основы языка F#.	
Знакомство с языком.	Обучение синтаксису и семантике языка на практике. Обучающиеся должны предоставить решение практических заданий.
Лабораторная работа №2: Списки в F#.	
Работа со списками. Генерация и обработка списков.	Научить работе со списками на практике. Обучающиеся решают два вида задач: на генерацию и обработку.
Лабораторная работа №3: Последовательности в F#.	
Работа с последовательностями.	Научить работе с последовательностями на практике. Работа с файловой системой с помощью последовательностей.
Лабораторная работа №4: Деревья в F#.	

Деревья.	Обучить работе с деревьями. Познакомить с видами обхода и кодом формирования дерева. Обучающиеся предоставляют решения практических заданий.

Для раздела «Функциональное программирование» было разработано методическое пособие, содержащее весь необходимый теоретический материал для изучения и решения типовых задач.

2.2. Методика преподавания раздела «Функциональное программирование»

Анализ курсов по изучению языков, реализующих функциональную парадигму, в России и за рубежом, позволяет сделать следующий вывод: основное внимание уделяется практике. Теоретическая часть лаконична и обычно ограничивается основными возможностями языка и среды программирования. Все остальные знания обучаемые получают посредством практики. Как известно, знания, полученные на практике, усваиваются лучше и глубже, чем при теоретическом обучении.

На занятиях рекомендуется проводить совместные рассуждения о возможностях решения некоторых задач. Обсуждать программный код выполненных практических заданий. Это позволяет включить всех обучаемых в работу и позволяет делиться между собой знаниями.

Чаще всего у обучающихся изначально хватает мотивированности заниматься программированием. Тем не менее, можно предложить продемонстрировать видео/презентацию с интересными проектами, сделанными с помощью данного языка и/или языков данной парадигмы. К началу изучения функциональной парадигмы все обучающиеся уже сталкивались с императивным программированием. А если этого не произошло по каким-то причинам и у обучающихся возникает страх перед программированием — необходимо объяснить, что им потребуется не очень

много знать функций и преподаватель всегда поможет. К тому же совместное обсуждение довольно быстро подтянет их до необходимого уровня.

Тема 1. Основы языка F#. В начале изучения есть необходимость рассказать обучающимся краткую историю функционального программирования. Привести реальные примеры использования языков, реализующих данную парадигму в жизни. Например: Erlang — используется в сложных распределительных системах, в частности, в телекоммуникационном оборудовании, Scala — на нем реализован twitter, F# — на нем пишутся приложения, создаются библиотеки, сайты (большой проект Web sharper), используется в серверной и клиентской части, применяется для программирования роботов, Python используется в веб-программировании, в некоторых интернет сервисах (youtube), Lisp — используется в сферах искусственного интеллекта и т.д. Далее происходит знакомство со средой программирования, рекомендуется выбрать Visual Studio, но можно использовать другие среды разработки. После начинается изучение основных конструкции и функции в F#. Если учащиеся уже сталкивались с языками программирования — можно провести аналогию между операторами из императивных языков и выражениями в F#. Изучаются основные конструкции и синтаксис языка.

В первой части предлагается начать изучение с простых операций и схожих структур, используемых в императивном программировании: базовые арифметические операции, условный оператор, оператор выбора, функции, рекурсия.

Например условное выражение(F#) и условный оператор(Pascal).

F#

```
let result =
    if x % 2 = 0 then
        "Четное"
    else
        "Нечетное"
result
```

Pascal

```
if x % 2 = then
    Result:= "Четное"
else
    Result:= "Нечетное"
```

Особое внимание необходимо уделять изучению и применению рекурсии, так как она является основным используемым инструментом. Как показывает практика, на первых порах обучаемые забывают о правилах вложенности, из-за отсутствия явного выделения блоков скобками, как это происходит в большинстве других языков.

Например, в задаче: Написать рекурсивную функцию, которая будет получать положительное число, а выводить строку, содержащую цифры от 0 до введенного числа.

```
open System
[<EntryPoint>]
let main (args : string[]) =
 let rec sum a =
   if a>=0 then
   sum (a-1) + string (a)
   else
    ""
 printfn "%A" (sum 9)
 Console.ReadKey() |> ignore
 0
```

В функцию main входят все нижние строки, потому что они имеют отступ. В функцию sum входит только условное-выражение.

Далее идет обучение таким функциональным возможностям, как композиция функций, каррирование и функции высших порядков. Изложение данной темы целесообразно иллюстрировать значительным количеством примеров, что позволит продемонстрировать нюансы этих возможностей. Закрепление темы требует качественных практических заданий, в т.ч. комбинированного характера.

Например, для того чтобы показать удобства использования композиции функции, можно показать это на следующем примере.

Пример: Реализовать несколько функции, каждая будет решать свою задачу: 1-возводить число в квадрат, 2-переводить число в строку, 3-возвращать длину строки. И реализовать композицию этих функции.

```
let square x = x * x
let toString (x : int) = x.ToString()
let strLen (x : string) = x.Length
```

```
let lenOfSquare = square >> toString >> strLen;;
lenOfSquare 5
```

Далее обучающиеся решают типовые задачи по теме начав на лабораторных работах, и если не успеют, решают дома, а потом предоставляют результат. Лабораторные и самостоятельные работы помогают закрепить навыки, полученные на аудиторных занятиях и изучить дополнительный материал. На лабораторных работах следует показать на практике все то, о чем говорилось на лекции: привести пример, который разберете и обсудите с обучающимися. Далее дать им индивидуальные практические задания. Проблемы, возникающие у обучающихся — стараться обсуждать совместно. Самым активным, рекомендуется давать дополнительный материал, как для изучения, так и для решения. Рекомендуется проводить совместное обсуждение способов решения сложных задач.

Дальнейшее освоение этого материала происходит при решении задач из этой и последующих тем.

Тема 2. Списки в F#. Познакомить с понятием списка и основными возможностями и функциями работы над списками. Самое главное необходимо объяснить для чего придумали списки и какие задачи они помогают решить. Здесь можно рассказать о динамическом изменении размера и видах списков, удобстве использовании. Это можно сделать на сравнении с массивом. Чаще всего в языках программирования используется фиксированный массив, а список может динамически изменять свой размер до необходимого. Возможно, это будет сохранение/получение не определенного количества элементов и дальнейшая работа с ними. Иначе может возникнуть впечатление, что они не нужны.

Например: Создать список из делителей заданного числа.

```
open System
[<EntryPoint>]
let main (args : string[]) =
 let delch ch = [
    for i= 1 to ch do
     if ch%i=0 then
```

```
        yield i
    ]
printfn "Полученный список %A" (delch 10)
printfn "(press any key to continue)"
Console.ReadKey() |> ignore
0
```

Достаточно простая программа, показывающая работу со списком

При изучении списков необходимо наглядно показать их структуру: рекомендуется привести примеры из реальной жизни. Например, это можно с помощью коробков — положив внутрь записки с числами или еще какой-нибудь информацией или на стопке книг — в данном случае их можно рассматривать как список списков. Будет большим плюсом, если обучающиеся уже изучали списки на каком-либо языке и знакомы с основными функциями работы над ними. Если же нет — то следует уделить больше времени на эту тему, так как она является одной из ключевых. Данная тема разделена на две части: генерация и обработка списков. Рекомендуется сначала изучить генерацию списков — как создавать их с использованием различных способов: с помощью циклов, с помощью рекурсии, с помощью клавиатурного ввода. А уже потом изучать обработку списков, используя изученные способы генерации. Рассмотреть основные функции работы над списками: нахождение длины списка, обход списка, свертка и т.д. Познакомить с библиотекой функции в Visual Studio. Рекомендуется установить программное обеспечение на русском языке, чтобы было понятнее для обучающихся. Если при изучении возникнут какие-либо проблемы с пониманием у учащихся, рекомендуется обсудить проблему всей группой. Также после изучения обучающиеся выполняют практические задания из соответствующего раздела к каждой части темы.

Практику следует проводить также как и в предыдущей теме. Самостоятельная работа позволяет обучающимся глубже разобраться в этой теме и познакомиться с библиотекой функции работы над списками. Рекомендуется преподавателю менять задачи обучающимся по необходимости, смотря на решения предыдущих заданий, чтобы охватить

различные особенности и ранее не используемые функции. Будет плюсом дать задание на самостоятельную работу: о выяснении быстроты выполнения и определения используемого количества памяти между списком и массивом, которые используются в решении одной задачи.

Тема 3. Последовательности в F#. В следующем разделе изучаются последовательности. При изучении этой темы часть сложностей снимается, так как последовательности схожи со списками. Но следует разграничить эти типы данных и показать, в чем различие между типами seq и list, а оно заключается в том, что в каждый конкретный момент времени в памяти существует только один элемент последовательности. Это дает возможность создавать последовательность очень больших размеров, в отличие от списка и массива, где возможно переполнение. Например: Создание последовательности всех целых чисел, входящих в тип int32.

```
let allIntSeq = seq { for i = 0 to System.Int32.MaxValue -> i }
```

Попытка создания списка с таким же количеством элементов приведет к переполнению и ошибке.

Пример 2: Вывести все пути к текстовым файлам в указанном каталоге и подкаталогах.

```
open System
open System.IO
[<EntryPoint>]
let main argv =
  let str = "C:\\Temp"
  let rec allFilesUnder basePath =
    seq {
     yield! Directory.GetFiles(basePath)
     for subdir in Directory.GetDirectories(basePath) do
      yield! allFilesUnder subdir
    }
  let sp = Seq.toList (allFilesUnder str)
  for c in sp do
      if not(c.Contains("txt")) then
        printfn "%A" c
  Console.ReadKey() |> ignore
  0
```

В Visual Studio также присутствует библиотека с функциями работы над последовательностями, как и со списками. Необходимо это показать и

дать обучающимся это на самостоятельное изучение. Проблем с освоением этой темы быть не должно, так как она схожа со списками.

Выполнение практических заданий «Последовательности в F#» происходит в самостоятельном режиме с последующим совместным обсуждением некоторых задач и способов их решения. Практика закрепит знания полученные на лекциях и поможет сформировать ясное понимание последовательности.

Тема 4. Деревья в F#. Это самая сложная тема курса. Для успешного освоения обучающимися — необходимо познакомить их с понятием дерева и дать наглядное представление дерева, показать все методы обхода, а уже потом переходить к объяснению отдельных участков кода формирования дерева. Также необходимо рассказать, для чего используются деревья. Можно использовать пример нахождения максимального числа и сравнить решение в двоичном дереве поиска, в массиве и списке. И посмотреть, где будет сложность больше. Это позволит объяснить некоторые плюсы использования деревьев.

Существует вероятность того, что у учащихся возникнут сложности с пониманием написания кода — придется дать им готовый шаблон формирования дерева, который они будут дорабатывать, и решать свои задачи.

Практику следует начинать с легких заданий: таких как поменять порядок обхода или способ вывода элементов дерева, а уже потом переходить на решение самостоятельной работы. Выполняя самостоятельную работу «Деревья в F#», обучающиеся закрепят на практике понятие дерева, его назначение и применение. Поработают со способами формирования и видами обхода дерева. Рекомендуется дать задание на самостоятельное исследование определения быстроты работы программы, используемой дерево.

2.3. Экспериментальное преподавание и его результаты

В первом семестре 2016-2017 учебного года проводилась апробация раздела «Функциональное программирование» курса «Языки программирования». Апробация проходила в Пермском гуманитарно-педагогическом университете со студентами третьего курса, обучающимися по направлению подготовки 44.03.04 "Профессиональное обучение" (по отраслям), профиль «Информатика и вычислительная техника». К моменту проведения курса студенты уже были знакомы с императивной и объектно-ориентированной парадигмами программирования (языки Pascal, C/C++, C#) и имели навыки программирования в рамках названных парадигм.

Апробация курса включила в себя проведение лекций и лабораторных занятий.

Были рассмотрены все темы, обозначенные в рабочей программе, а именно.

1. Основы языка F#.
    1.1. Базовые операции и операторы.
    1.2. Функциональные возможности.
2. Списки в F#.
    2.1. Генерация списков.
    2.1. Обработка списков.
3. Последовательности в F#.
    3.1. Работа с последовательностями.
4. Деревья в F#.
    4.1. Формирование, виды обходов.
    4.2. Работа с деревьями.

В процессе изучения дисциплины студенты выполняли лабораторные работы в соответствии с планом и, в итоге, реализовали зачетные задания. В ходе работы обучаемые проявили интерес к изложенному материалу. В итоге

заметно улучшилось их понимание различий парадигм программирования. Все обучаемые справились с поставленными задачами, некоторые студенты выполнили дополнительные задания. Фактическая продолжительность изучения каждой темы совпала с планируемой, рабочая программа в корректировках не нуждалась.

Вторая апробация проходила в 2016-2017 году в Пермском государственном педагогическом университете со студентами второго курса магистратуры, обучающимися по направлению «Педагогическое образование» профиль «Информатика и ИКТ». На момент апробации студенты обладали опытом программирования на различных языках, реализующих императивную парадигму и были знакомы с логическим программированием. Студенты в интенсивном режиме освоили необходимые знания и выполнили все практические задания.

В результате у студентов сформировалось понимание принципов и возможностей функциональной парадигмы. Все группы успешно справились с данной программой.

Материалы представлялись и обсуждались на научно-методической конференции «Математика и междисциплинарные исследования-2017» 16 мая 2017 года в Пермском государственном национальном исследовательском университете. Были опубликованы материалы выступления в сборнике статей конференции (см. приложение 3) [3].

Заключение

Сегодня в вузах, связанных с информационными технологиями, изучается преимущественно императивная парадигма. Но быстрое развитие ИТ в последнее время заставляет пересматривать цели курса информатики и ИКТ и вносить коррективы в его содержание. В этой работе сделана попытка взглянуть на раздел «Основы алгоритмизации и программирования» с точки зрения учебного курса «Функциональное программирование».

Цель настоящей работы заключалась в создании дидактических материалов обучения, раздела «Функциональное программирование» дисциплины «Языки программирования».

Приведём полученные результаты.

1. В ходе работы был проведён обзор литературы по теме исследования, преимущественно практического содержания.
2. Проведён анализ существующих курсов по данной тематике.
3. Описана методика преподавания раздела «Функциональное программирование».
4. Составлен дидактический комплект для изучения программирования на языке F#.
5. Дидактические материалы представлены в Приложении 4 в виде веб-сайте, который при необходимости можно разместить в сети интернет.
6. Проведена успешная апробация в двух различных группах ПГГПУ.
7. По теме исследования написана и опубликована статья, состоялось выступление на конференции.

Задачи решены в полном объеме, цель достигнута – созданы дидактические материалы для обучения студентов разделу «Функциональное программирование» дисциплины «Языки программирования».

Перспективы данной темы заключаются в дальнейшем изучении функционального программирования, и добавление раздела «Функциональное программирование» в дисциплину «Языки программирования», а также для использования материала в преподавании студентам ПГНИУ.

Библиографический список

1. *Гордон М.* Введение в функциональное программирование / М. Гордон. – М. : Мир, 1996. – 104 с.

2. *Городняя Л.В.* Основы функционального программирования: Учебное пособие / Л.В. Городняя. – 2-е изд. – Новосибирск.: НГУ, 2004. – 165 с.

3. *Горяева И.А., Мухутдинова Д.Н.* Сборник статей конференции: «Математика и междисциплинарные исследования» – 2017 Горяева И.А., Мухутдинова Д.Н. – Пермь. : ПГНИУ, 2017. – 235 с.

4. *Гринштейн Г.* История языков программирования: как Haskell стал стандартом функционального программирования [Электронный ресурс] / *Г. Гринштейн.* – Режим доступа: https://habrahabr.ru/post/307618/. (Дата обращения: 27 мая 2017 г.).

5. *Душкин Р.В.* Лекции по функциональному программированию / Р.В. Душкин. – М. : МИФИ, 2001. – 69 с.

6. *Зыков С.В.* Введение в теорию программирования. Функциональный подход. – М.: Учебный Центр безопасности информационных технологий Microsoft МИФИ, 2003. – 356 с.

7. *Интуит.* Национальный открытый университет [Электронный ресурс] URL: http://www.intuit.ru/studies/courses/471/327/info (дата обращения: 04.04.2017).

8. *Кембридж.* Программа дисциплины «Функциональное программирование»(Introduction To Functional Programming) / [Электронный ресурс] URL: http://www.cl.cam.ac.uk/ (дата обращения: 04.04.2017).

9. *Лазин Е.А.* Введение в F# / Е.А. Лазин, М.К. Моисеев, Д.Н. Сорокин // Практика функционального программирования / под ред. Б.Д. Астапов. – М. : .ФПРОГ. – 2010. – Вып. 5. – С.26-30.

10. *Лапшин Н.*А. Программа дисциплины «Функциональное и логическое программирование» / Н.А. Лапшин – Армавир. : Армавирская государственная педагогическая академия, 2013.

11. *Ньюард Т.В.* Первое знакомство с F# [Электронный ресурс] / *Т.В. Ньюард*. – Режим доступа: https://msdn.microsoft.com/ru-ru/magazine/cc164244.aspx. (Дата обращения: 27 мая 2017 г.).

12. Пратт Т.У. Языки программирования: разработка и реализация / Т.У. Пратт – Москва. : Изд. «Мир» 1979.

13. *Рабаданова Р.*М. Программа дисциплины «Функциональное программирование и интеллектуальные системы» / Р.М. Рабаданова – Дагестан. : ДГУ, 2015.

14. *Роганова Н.А.* Функциональное программирование: Учебное пособие для студентов высших учебных заведений – М. : ГИНФО, 2002. 260 с.

15. *Семакин И.Г, Хеннер Е.К.* Методика преподавания информатики / И.Г. Семакин, Е.К. Хеннер – Под общей ред. М.П. Лапчика. – М. : Издательский центр «Академия», 2008. – 624 с.

16. *Смит К.* Программирование на F#. – Пер. с англ. – СПб. : Символ-Плюс, 2011. – 448 с.

17. *Сошников Д. В.* Программирование на F# / Д.В. Сошников. – М. : ДМК Пресс, 2011. – 192 с.

18. *Сошников Д.В.* Программа дисциплины «Функциональное программирование» / Д.В. Сошников – Москва. : ГУ ВШЭ, 2010.

19. *Харрисон П.* Функциональное программирование / П. Харрисон, А. Филд. – М. : Мир, 1993. – 640 с.

Приложение 1. Фрагмент РПД «Основы функционального программирования» в рамках дисциплины «Языки программирования»

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение

высшего профессионального образования

«Пермский государственный гуманитарно-педагогический университет»

Кафедра информатики и ВТ

Рабочая программа дисциплины Направление подготовки

_____

Профиль подготовки: Информатика и ИКТ

Пермь 2017

1. **Цель изучения:**

   Целью изучения раздела «Функциональное программирование» является овладение обучающимися основами программирования в рамках функциональной парадигмы. Научить принципам написания программ и методам решения задач, с используемыми структурами данных в функциональном программировании.

2. **Место дисциплины в структуре ОП:**

   Раздел относится к базовой части дисциплины «Языки программирования», шифр дисциплины Б1.В.ОД.19.

   Перечень дисциплин, предшествующих изучению данной дисциплины:

   1. Математика
   2. Алгебра логики

3. **Объем дисциплины:**

3.1. Объем дисциплины и виды учебной работы

Вид учебной работы	Формы обучения	
	Очная	Заочная
Общая трудоемкость: часы/зачетные единицы	40	
Номера семестров	1	
Аудиторные занятия(всего): В том числе:	16	
Лекции(Л)	6	
Практические занятия(ПЗ)		
Семинарские занятия(СЗ)		
Лабораторные работы(ЛР)	10	
Самостоятельная работа(СРС) (всего)	24	
Форма промежуточной аттестации (экзамен, зачет) - № семестров	зачет	

4.1. Распределение часов по темам и видам учебной работы

4.1.1. Очная форма обучения

№ п/п	Раздел/тема	Всего час.	Виды учебной работы в часах			
			Аудиторная работа			Самостоятельная работа
			Лек.	Сем./пр	Лаб.	
1.	Основы языка F#	10	2		2	6
2.	Списки в F#	9	1		2	6
3.	Последовательности в F#	9	1		2	6
4.	Деревья в F#	12	2		4	6

5. Содержание дисциплины

5.1. Программа дисциплины (общее содержание всех тем)

Тема 1. Основы языка F#.

Основные понятия: Функция, рекурсия, композиция, функции высших порядков

Содержание темы: Рассматривается краткая история функционального программирования. Отмечается современное состояние. Язык программирования F#. Изучаются основные конструкции и синтаксис языка. Выполнение практических заданий «Основы языка F#».

Тема 2. Списки в F#.

Основные понятия: Список, обход, map, filter, fold.

Содержание темы: Понятие списка. Основные возможности и функции работы над списками. Генерация и обработка списков. Выполнение практических заданий «Списки в F#».

Тема 3. Последовательности в F#.

Основные понятия: Последовательность, yield

Содержание темы: Понятие последовательности. Отличие от списков. Основные возможности и функции работы над последовательностями.

Генерация и обработка последовательностей. Выполнение практических заданий «Последовательности в F#».

Тема 4. Деревья в F#.

Основные понятия: Дерево, бинарное дерево, узлы, листья.

Содержание темы: Понятие дерева. Назначение и применение. Формирование и виды обхода. Выполнение практических заданий «Деревья в F#».

5.2. Содержание лабораторных работ (лабораторный практикум)

Наименование раздела дисциплины/тема	Тема лабораторной работы	Содержание лабораторной работы	Учебно-методическое обеспечение (Базовый учебник, ссылка на ЭБС и др.)
Раздел 1: Основы языка F#.	Основы языка F#	Типовые задачи по теме	Учебно-методическое пособие «Функциональное программирование». Основы языка F#
Раздел 2: Списки в F#.	Генерация списков	Типовые задачи по теме	Учебно-методическое пособие «Функциональное программирование». Списки в F#
	Обработка списков	Типовые задачи по теме	
Раздел 3: Последовательности в F#	Последовательности в F#	Типовые задачи по теме	Учебно-методическое пособие «Функциональное программирование». Последовательности в F#
Раздел 4: Деревья в F#	Деревья в F#	Типовые задачи по теме	Учебно-методическое пособие «Функциональное программирование». Деревья в F#

6. Фонд оценочных средств:

Разделы, темы дисциплины	Код формируемой компетенции	Оценочное средство
Основы языка F#.		баллы
Списки в F#.		баллы
Последовательности в F#		баллы
Деревья в F#		баллы

7. Учебно-методическое, информационное и материально-техническое обеспечение дисциплины

7.1. Основная литература

№ п/п	Автор и название литературного источника	Выходные данные	ссылки на электронные библиотечные системы и базы данных
1.	Смит К. Программирование на F#. — Пер. с англ. — СПб.: Символ-Плюс, 2011. — 448 с.		
2.	Сошников Д. В. Программирование на F#. – М.: ДМК Пресс, 2011. – 192 с.		
3.	Интуит. Национальный открытый университет [Электронный ресурс] URL: http://www.intuit.ru/studies/courses/471/327/info (дата обращения: 04.04.2017		

7.2. Рекомендуемая литература

№ п/п	Автор и название литературного источника	Выходные данные	ссылки на электронные библиотечные системы и базы данных
1.	Смит К. Программирование на F#. — Пер. с англ. — СПб.: Символ-Плюс, 2011. — 448 с.		
2.	Сошников Д. В. Программирование на F#. – М.: ДМК Пресс, 2011. – 192 с.		
3.	Интуит. Национальный открытый университет [Электронный ресурс] URL: http://www.intuit.ru/studies/courses/471/327/info (дата обращения: 04.04.2017		

7.3. Перечень ресурсов информационно-телекоммуникационный сети "Интернет" (базы данных, информационно-справочные и поисковые системы и др.):

1. Интуит. Национальный открытый университет [Электронный ресурс] URL: http://www.intuit.ru/studies/courses/471/327/info.

7.4. Материально-техническое обеспечение:

- компьютерное и мультимедийное оборудование
- пакет прикладных обучающих программ

Приложение 2. Учебно-методическое пособие по дисциплине «Функциональное программирование»

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Государственное образовательное учреждение высшего профессионального образования

«Пермский государственный гуманитарно-педагогический университет»

Функциональное программирование на F#

Оглавление

Основные сведения

Основные сведения о работе с F#

F# входит в систему разработки Visual Studio начиная с 2010г., а пакет расширения можно установить на версию 2008г.

Файлы, содержащие код на F#, обычно имеют следующие расширения:

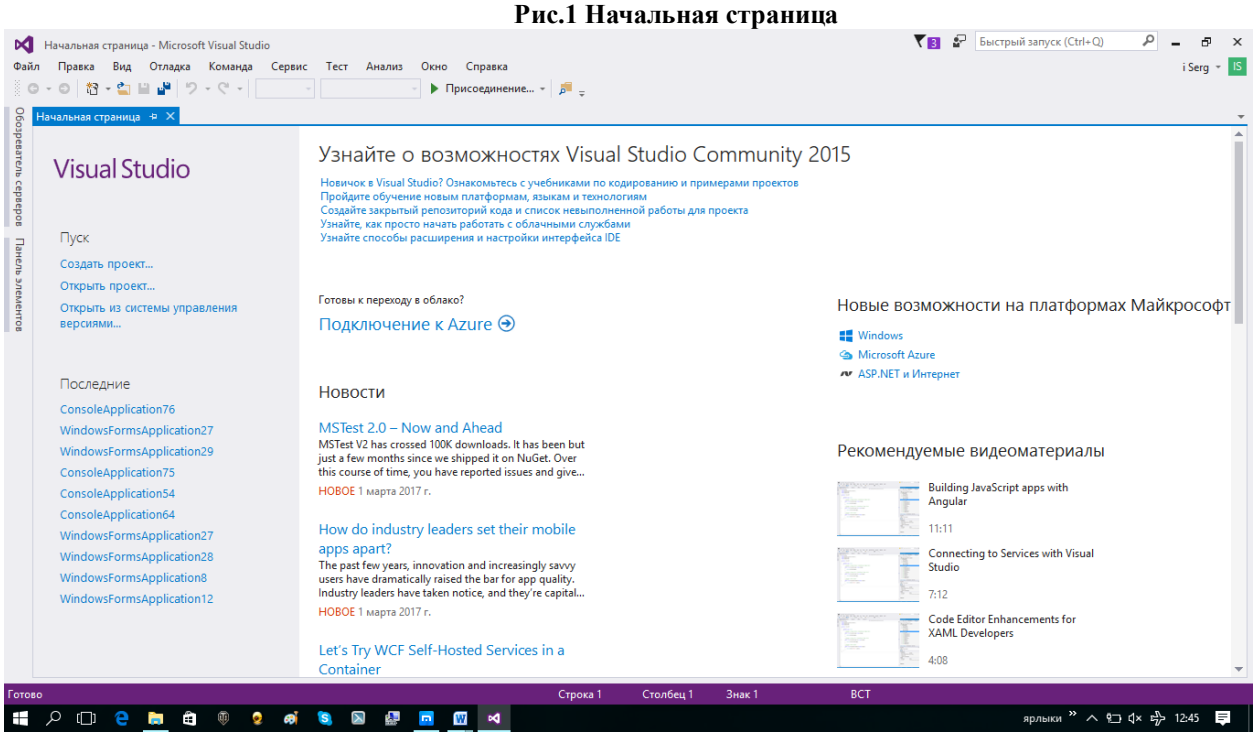
- *.fs — обычный файл с кодом, который может быть скомпилирован;
- *.fsi — файл описания публичного интерфейса модуля. Обычно генерируется компилятором на основе кода, а затем редактируется вручную;
- *.fsx — исполняемый скрипт. Может быть запущен прямо из проводника Windows при помощи соответствующего пункта всплывающего меню или передан на исполнение в интерактивную консоль Fsi.exe.

Для F# не поддерживается режим конструктора страниц ASP.NET. Это не означает, что F# нельзя использовать вместе с ASP.NET — отнюдь. Просто в Visual Studio, работая с F#, нельзя без дополнительных средств перетаскивать элементы управления, как при работе с C# или Visual Basic, но можно создавать элементы программно.

Создание приложения

После запуска Visual Studio 2015, откроется «Начальная страница» (рис. 1):

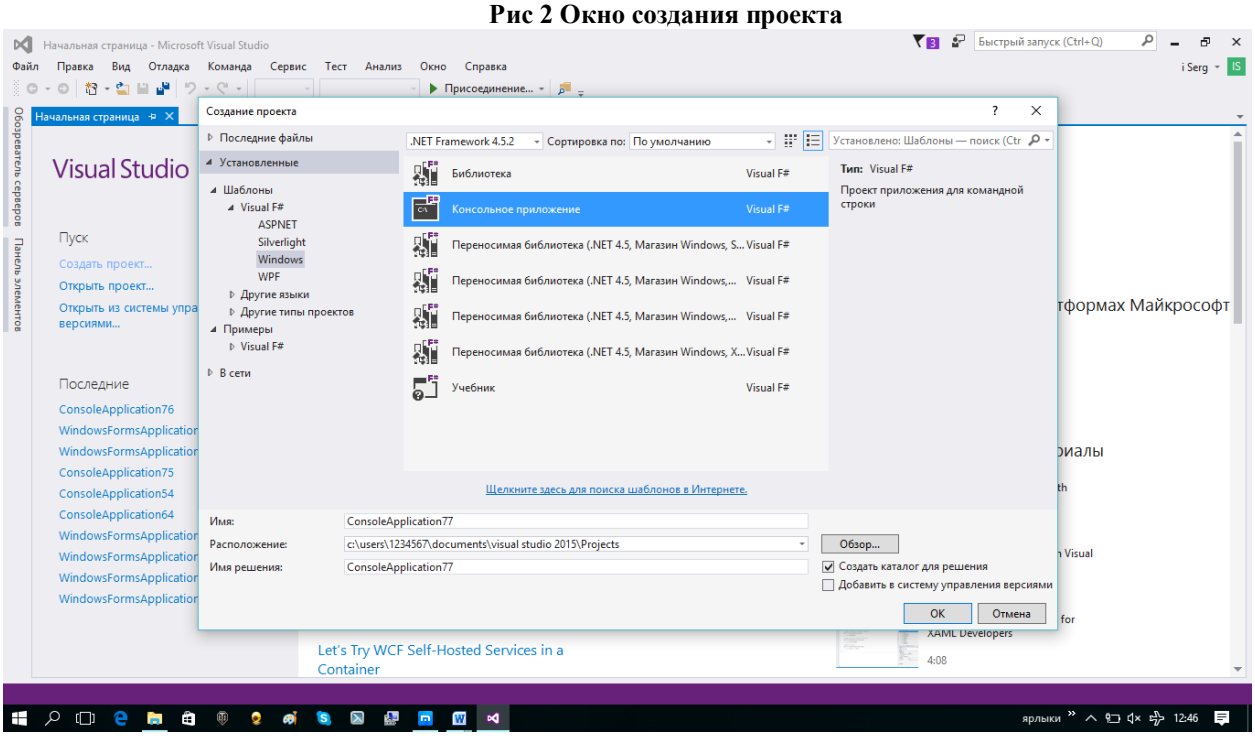
Рис.1 Начальная страница



Для начала, надо создать пустой проект, для этого выполним последовательно: *Файл -> Создать -> Проект…* (также можно просто нажать сочетание клавиш Ctrl+Shift+N или пункт «Создать проект…» на «Начальной странице»): Откроется окно создания проекта и выбора необходимых нам параметров. Выберем слева в пункте *Установленные шаблоны -> Visual F#*, далее найдём в списке *Учебник по F#*. Также здесь можно выбрать, какой использовать «фреймворк» (набора компонентов для написания программ). В нашем случае выберем *.NET Framework 4*.

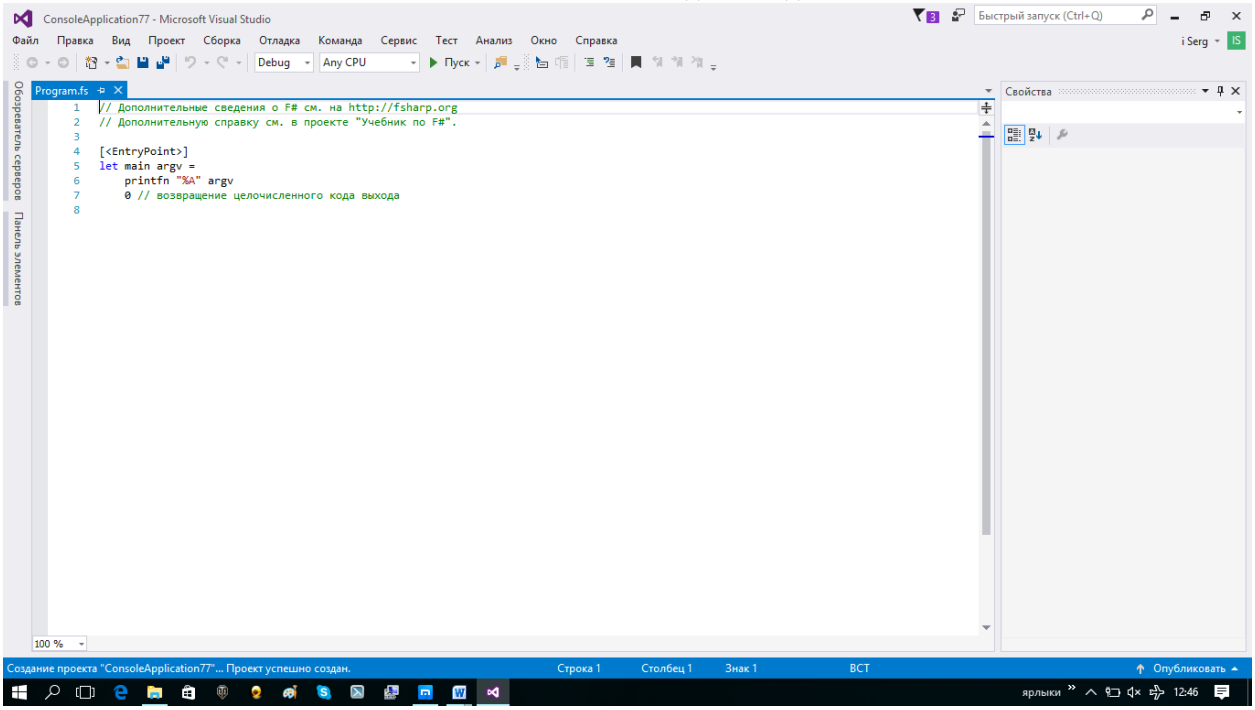
В поле *Имя* вводим название программы. В поле *Расположение* указана конечная директория, где будет находиться весь проект (значение «по умолчанию» можно поменять, выполнив действия: *Сервис -> Параметры… -> Проекты и решения ->* меняем путь в поле *Размещение проектов*). Выберем расположение удобное для быстрого поиска. В поле *Имя решения* вводится либо название программы «по умолчанию» из поля Имя автоматически, либо можно ввести своё собственное. Под этим именем будет создана конечная папка проекта (если Имя и Имя решения разные) (см. рис.2).

Рис 2 Окно создания проекта



После нажатия клавиши ОК мы увидим сформированный проект и исходный код консольного приложения (не пустого изначально). Функция *main* является главной и в ней уже пишется рабочий код (см. рис.3).

Рис. 3 Исходный код



Часть 1. Основы языка F#

- F# поддерживает функциональное программирование, то есть такой стиль программирования, когда описывается, что должна делать программа, а не как она должна работать.
- F# поддерживает объектно-ориентированное программирование. Позволяет заключать программный код в классы и объекты, что дает возможность упростить его.
- F# поддерживает императивное программирование. Можно реализовать изменение содержимого областей памяти, читать и записывать файлы, обмениваться данными по сети и так далее.
- F# является статически типизированным языком. Вследствие этого информация о типах доступна уже на этапе компиляции, что обеспечивает большую надежность программного кода.
- F# – это язык для платформы .NET. Он работает на основе общей языковой инфраструктуры (Common Language Infrastructure, CLI), и поэтому при использовании этого языка доступны: механизм автоматической сборки мусора (управления памятью) и мощные библиотеки классов. Кроме того, F# поддерживает все концепции, характерные для .NET, такие как делегаты, перечисления, структуры, и так далее.

F# и Visual Studio Создание и отладка проектов на языке F# выполняется точно так же, как и проектов на языке C# или VB.NET, однако проекты на F#, состоящие из нескольких файлов, имеют некоторые уникальные особенности. Кроме того, при работе с этим языком поддерживается такая возможность, как F# Interactive, которая позволит вам резко поднять свою производительность.

Инструментальные средства наполняют жизненной силой любой язык программирования, и F# не является исключением. F# имеет полноценную

поддержку в Visual Studio. Для языка F# в среде Visual Studio поддерживаются все известные возможности, такие как отладчик, IntelliSense (механизм автодополнения), шаблоны проектов и другие.

Создание значений

Описание всех новых значений начинается со служебное слова *let*. Вот несколько примеров:

```
let x = 3
let y = 3.0
let s = "string"
let список = [1; 4 ; 3 ; 0]
let кортеж = (s, x)
let последовательность = [1..5]
```

Это неизменяемые аргументы. На их основе только можно создать новые данные.

```
let z = x + 2
let c = s + " вот так"
```

Арифметика

Оператор	Описание	Пример	Результат
+	Сложение	1 + 2	3
-	Вычитание	1 – 2	-1
*	Умножение	2 * 3	6
/	Деление	8 / 3	2
**	Возведение в степень	2.0 ** 8.0	256.0
%	Деление по модулю (остаток от деления)	7 % 3	1

Математические функции

Функция	Описание	Пример	Результат
abs	Абсолютное значение числа	abs-1.0	1.0
ceil	Округления вверх до ближайшего целого	ceil 9.1	10
exp	Возведение e в степень	exp 1.0	2.718
floor	Округление вниз до ближайшего целого	floor 9.9	9.0
sign	Знак числа	sign -5	-1
log	Натуральный логарифм	log 2.71828	1.0

log10	Логарифм по основанию 10	log10 1000.0	3
sqrt	Корень квадратный	sqrt 4.0	2.0
cos	Косинус	cos 0.0	1.0
sin	Синус	sin 0.0	0.0
tan	Тангенс	tan 1.0	1.557
pown	Возведение целого числа в степень	pown 2 10	1024

Функции определяются точно так же, как значения, с той лишь разницей, что все, что следует за именем функции, является ее параметрами. Несколько примеров: в первом возвращается квадрат числа, во втором примере число возводиться в степень. *Обратите внимание, что тип аргумента можно прописывать явно.*

```
let square x = x*x
square 2
```

```
let f (x:float) = x**3.0
f 2
```

Пробельные символы имеют значение

В других языках программирования, таких как C#, для обозначения окончания инструкций и выделения блоков программного кода используются точка с запятой и фигурные скобки. При этом программисты для повышения удобочитаемости обычно оформляют программный код, используя отступы, поэтому эти дополнительные символы часто лишь загромождают программный код. В языке F# пробельные символы – пробелы и символы перевода строки – имеют важное значение. Компилятор F# позволяет использовать пробельные символы для разграничения блоков кода. Например, *любой код, имеющий отступ больше, чем ключевое слово, считается телом инструкции*. Поскольку символы табуляции могут соответствовать различному числу пробелов, они запрещены к использованию в коде F#. *Обратите внимание, что вложенность блоков принято определять отступом в четыре пробела.*

```
let main (args : string[]) =
    let rnd = System.Random()
    let delch =
        for i = to 10 do
           printfn rnd.Next(10)
```

Комментарии

Подобно любым языкам программирования, F# позволяет добавлять комментарии в программный код. Однострочные комментарии начинаются с двух символов слеша // – все, что следует за ними до конца строки, будет игнорироваться компилятором. Для добавления более объемных описаний, занимающих несколько строк, можно использовать многострочные комментарии, которые заключаются в пары символов **(* и *)**

F# Interactive

Visual Studio имеется очень удобный инструмент, который называется F# Interactive, или FSI. F# Interactive (Интерактивный сеанс F#) относится к классу инструментов, известных под названием REPL (Read-Evaluate-Print Loop – цикл чтения-вычисления-вывода). Этот инструмент принимает код F#, компилирует его, выполняет и выводит результаты. Это позволяет легко и быстро экспериментировать с программным кодом F#, не создавая новые проекты или полноценные приложения только ради того, чтобы увидеть результаты работы фрагмента из пяти строк. В Visual Studio с типичными настройками окно F# Interactive можно открыть, нажав комбинацию клавиш Control+Alt+F. Как только окно FSI будет открыто, в него можно вводить код F#, пока вы не введете последовательность ;; и символ перевода строки. Введенный код будет скомпилирован и выполнен.

Попробуйте выполнить следующие фрагменты в окне FSI. Обратите внимание, что каждый фрагмент завершается символами ;;:

```
2 + 2;;
val it : int = 4

let x = 1
```

```
let y = 2.3;;
val x : int = 1
val y : float = 2.3

float x + y;;
val it : float = 3.3

let cube x = x * x * x;;
val cube : int -> int

cube 4;;
val it : int = 64
```

FSI существенно упрощает тестирование и отладку приложений благодаря тому, что в Visual Studio вы можете переслать F# код из своего текущего проекта в окно FSI, выделив его и нажав комбинацию клавиш Alt+Enter.

Управление потоком выполнения(if)

В языке F# пробельные символы – пробелы и символы перевода строки – имеют важное значение. Компилятор F# позволяет использовать пробельные символы для разграничения блоков кода. Например, любой код, имеющий отступ больше, чем ключевое слово if, считается телом инструкции if. Поскольку символы табуляции могут соответствовать различному числу пробелов, они запрещены к использованию в коде F#. Внутри функций можно изменять поток выполнения с помощью ключевого слова if. Условное выражение должно иметь тип bool, и если результатом его вычисления является значение true, выполняется вложенный блок кода.

Более сложные ветвления могут быть реализованы с помощью условных выражений (if-expression). Условные выражения действуют именно так, как от них и ожидается: если условное выражение возвращает true, то выполняется первый блок кода, в противном случае выполняется второй блок. Однако есть одно обстоятельство, которое существенно отличает F# от других языков программирования: условные выражения в языке F# возвращают значение. В следующем примере результат, возвращаемый условным выражением, связывается с именем result. То есть если условие

x % 2 = 0 выполняется, то значением result будет ”Четное ”, в противном случае ”Нечетное”:

```
open System
[<EntryPoint>]
let main argv =
    let isEven x =
      let result =
        if x % 2 = 0 then
          "Четное"
        else
          "Нечетное"
      result

    printfn "%A" (isEven 3)
    Console.ReadKey() |> ignore //ждет ввод символа и не дает
закрыться консоли
    0 // возвращение целочисленного кода выхода
```

printfn – является функцией вывода и может работать только со строчными типами. Для вывода значений других типов требуется переводить в строковый формат либо использовать `"%A"`, данный параметр сообщает, что необходимо игнорировать тип данных и выводить то, что есть.

В языке F# имеется синтаксический помощник, помогающий избегать использования глубоко вложенных условных выражений, – ключевое слово elif. С его помощью можно связывать цепочки условных выражений, избегая вложенности:

```
open System
[<EntryPoint>]
let main (args : string[]) =
  let isWeekday day =
    if day = "Понедельник" then true
    elif day = "Вторник" then true
    elif day = "Среда" then true
    elif day = "Четверг" then true
    elif day = "Пятница" then true
    else false
  printfn "%A" (isWeekday "Понедельник")
  Console.ReadKey() |> ignore //ждет ввод символа и не дает закрыться
консоли
  0 // возвращение целочисленного кода выхода
```

Поскольку результатом условного выражения является значение, все операторы (clause) в таком выражении должны возвращать значения одного и того же типа.

Сопоставления с образцом(match)

В любых программах возникает необходимость фильтровать и сортировать данные. Для этой цели в функциональном программировании используется сопоставление с образцом (pattern matching). Сопоставление с образцом напоминает инструкцию switch в языках C# и C++, но обладает более широкими возможностями. По сути это набор правил, которые выполняются при совпадении входного значения с образцом. Затем выражение сопоставления с образцом возвращает результат правила, для которого было найдено соответствие. Вследствие этого все правила должны возвращать значения одного и того же типа. Для использования сопоставления с образцом применяются ключевые слова **match** и **with** с набором правил, за каждым из которых следует стрелка ->. Следующий фрагмент демонстрирует использование сопоставления с результатом выражения isOdd x для имитации поведения условного выражения. Первое правило соответствует значению true, и, если будет найдено совпадение с этим правилом, в консоль будет выведено сообщение “x нечетное”:

```
open System
[<EntryPoint>]
let main (args : string[]) =
  let isOdd x = (x % 2 = 1)
  let describeNumber x =
    match isOdd x with
     | true -> printfn "x нечетное"
     | false -> printfn "x четное"

  printfn "%A" (describeNumber 8)
  Console.ReadKey() |> ignore //ждет ввод символа и не дает закрыться
консоли
  0 // возвращение целочисленного кода выхода
```

Сопоставление с константами – это простейшая форма сопоставления с образцом. В следующем примере создается таблица истинности для операции конъюнкция, сопоставляя оба значения кортежа одновременно.

```
open System
[<EntryPoint>]
let main (args : string[]) =
  let testAnd x y =
    match x, y with
    | 1, 1 -> true
    | 1, 0 -> false
    | 0, 1 -> false
    | 0, 0 -> false
    | _, _ -> false

  printfn "%A" (testAnd 1 0)
  Console.ReadKey() |> ignore //ждет ввод символа и не дает закрыться
консоли
  0 // возвращение целочисленного кода выхода
```

Символ подчеркивания _ в выражениях сопоставления играет роль группового символа (wildcard) и совпадает с любыми значениями. Благодаря этому мы можем упростить предыдущий пример, добавив правило с групповым символом, которое будет выполняться для любых комбинаций входных значений кроме true true:

```
let testAnd x y =
    match x, y with
    | 1, 1 -> true
    | _, _ -> false
```

Правила сопоставления с образцом проверяются в порядке их объявления. Поэтому если вставить правило с групповыми символами первым, все последующие правила никогда не будут проверяться.

Циклы while

Выражения while продолжают выполняться до тех пор, пока булево выражение не вернет значение false. (По этой причине циклы while не могут применяться в чисто функциональном стиле, так как цикл никогда не завершится, потому что вы никогда не сможете изменить возвращаемое

значение предиката.) Обратите внимание, что условное выражение в цикле while должно возвращать значение типа bool, а тело цикла – значение unit. Следующий пример показывает, как можно использовать цикл while для перебора нескольких значений:

```
open System
[<EntryPoint>]
let main (args : string[]) =
 let mutable i = 0
 while i < 5 do
   i <- i + 1
   printfn "i = %d" i
 Console.ReadKey() |> ignore //ждет ввод символа и не дает закрыться
консоли
 0 // возвращение целочисленного кода выхода
```

Условие цикла while проверяется еще до выполнения тела цикла, поэтому если при начальных условиях выражение вернет false, тело цикла не будет выполнено ни разу.

С помощью служебного слова mutable создается изменяемая переменная. А ее изменение происходит, таким образом: `i <- i + 1`. Еще один способ создать изменяемую переменную – использовать ref. *Обратите внимание, на синтаксис обращения к переменной.*

```
let i = ref 0
 while !i < 5 do
   i := !i + 1
   printfn "i = %d" !i
```

Циклы for

Когда необходимо выполнить фиксированное число итераций, можно использовать выражение for, которое в языке F# имеет две разновидности: простой цикл и перечисление.

Простые циклы for

Простые циклы for создают новое целочисленное значение и последовательно увеличивают его до определенного значения. Цикл не будет

выполняться, если значение счетчика окажется больше максимального значения:

```
open System
[<EntryPoint>]
let main (args : string[]) =
 for i = 1 to 5 do
   printfn "%d" i

 Console.ReadKey() |> ignore //ждет ввод символа и не дает закрыться
консоли
 0 // возвращение целочисленного кода выхода
```

Чтобы организовать счет в обратном порядке, следует использовать ключевое слово downto. В этом случае цикл не будет выполняться, если значение счетчика окажется меньше минимального значения:

В обратном порядке:

```
for i = 5 downto 1 do
   printfn "%d" i
```

Простые счетные циклы for поддерживают только целочисленные счетчики, поэтому если требуемое количество итераций больше System. Int32.MaxValue, используйте циклы-перечисления for.

Циклы-перечисления for

Более обобщенная разновидность циклов for выполняет итерации по элементам последовательности. Циклы-перечисления for могут работать с любыми типами seq, такими как array или list:

```
for i in [1 .. 5] do
 printfn "%A" i
```

Циклы-перечисления for обладают более широкими возможностями, чем циклы foreach в языке C#, благодаря тому, что они могут использовать сопоставление с образцом. Элемент, который следует за ключевым словом for, в действительности является правилом сопоставления с образцом, поэтому если это новый идентификатор, такой как i, он просто захватывает

значение сопоставления. Однако имеется возможность использовать более сложные правила.

Функции высших порядков

В F# имеется возможность посылать функции в другие функции в виде аргументов. Это позволяет разнообразить и сократить код. В примере имеется главная функция func, в которую мы посылаем функции fun1 и fun2 и число 7. В первой скобке(func1 x) применяется к числу первая функция(fun1). Во второй скобке(fun2) применяется к числу вторая функция(fun2).

```
let func func1 func2 x = (func1 x) * (func2 x)
let fun1 a = a+3
let fun2 a = a*2
func fun1 fun2 7
```

Рекурсивные функции

Функция, которая вызывает саму себя, называется рекурсивной. Рекурсивные функции могут быть чрезвычайно полезны при использовании функционального стиля программирования, в чем вы вскоре убедитесь. Чтобы определить рекурсивную функцию, достаточно просто добавить ключевое слово ***rec***. В следующем фрагменте определяется функция, выводящая строку “0123456789”.

Определение рекурсивной функции

```
open System
[<EntryPoint>]
let main (args : string[]) =
 let rec sum a =
   if a>=0 then
   sum (a-1) + string (a)
   else
    ""
 printfn "%A" (sum 9)
 Console.ReadKey() |> ignore //ждет ввод символа и не дает закрыться
консоли
 0 // возвращение целочисленного кода выхода
```

Ключевое слово ***rec*** может показаться ненужным излишеством, так как другие языки программирования не требуют явного обозначения рекурсивных функций. Настоящая цель введения ключевого слова rec состоит в том, чтобы сообщить механизму выведения типов, что функция может использоваться в процессе определения типов. Ключевое слово rec позволяет вызывать функцию еще до того, как механизм выведения типов определит тип функции. Используя рекурсию в комбинации с функциями высшего порядка, вы легко сможете смоделировать конструкции циклов, которые можно найти в императивных языках программирования, не используя изменяемые значения.

В следующем примере демонстрируется рекурсия с аккумулятором(хвостовая рекурсия). Он накапливает результат, а по завершению рекурсии возвращает свое значение. Это позволяет ускорить работу рекурсии тем, что тут не нужен разворот с возвратом значений.

```
open System
[<EntryPoint>]
let main (args : string[]) =
 let sum a=
  let rec sumtail a acc =
    if a>=0 then
     sumtail (a-1)  (string (a)+acc)
    else
     acc
  sumtail a ""
 printfn "%A" (sum 9)
 Console.ReadKey() |> ignore //ждет ввод символа и не дает закрыться
консоли
 0 // возвращение целочисленного кода выхода
```

Взаимная рекурсия

Две функции, вызывающие друг друга, называются взаимно рекурсивными. Взаимно рекурсивные функции представляют определенную сложность для механизма вывода типов в языке F#. Чтобы определить тип первой функции, необходимо знать тип второй функции, и наоборот. Чтобы определить взаимно рекурсивные функции, необходимо

объединить их с помощью ключевого слова and, которое говорит компилятору F# выводить типы обеих функций одновременно:

Определение взаимно рекурсивных функций(четное – false, нечет true)

```
let rec isOdd n = if n = 0 then false elif n = 1 then true else isEven
(n-1)
and isEven n = if n = 0 then true elif n = 1 then false else isOdd (n-
1);;
```

```
open System
[<EntryPoint>]
let main (args : string[]) =
 let rec isOdd n =
  if n = 0 then
   false
  elif n = 1 then
   true
  else isEven (n - 1)
 and isEven n =
  if n = 0 then
   true
  elif n = 1 then
   false
  else isOdd (n - 1)

 printfn "%A" (isOdd 8)
 Console.ReadKey() |> ignore //ждет ввод символа и не дает закрыться
консоли
 0 // возвращение целочисленного кода выхода
```

Конвейер

Язык F# предоставляет лаконичное решение проблемы передачи промежуточных результатов от функции к функции с помощью прямого конвейерного оператора (pipe-forward opeator) |>.

Конвейер бывает прямой 3 |> f() и обратный f() <| 3.

Прямой конвейерный оператор позволяет переупорядочить параметры функции так, что при вызове функции последний ее параметр указывается первым. Преимущество прямого конвейерного оператора состоит в том, что с его помощью можно составлять целые цепочки вызовов функций. То есть передавать результат предыдущей функции на вход следующей функции. Прямой конвейерный оператор помогает компилятору «увидеть» последний

параметр функции ранее, благодаря чему механизм выведения типов может определить правильные типы функций без дополнительных аннотаций.

Пример у нас есть 3 функции. Сначала идет выполнение вызов функции f от числа, потом результат сразу передается функции f2. Результат от f2 в f3 и выдается результат.

```
let f x = x * x
let f2 x = x + x
let f3 x = x * 10
f 3 |> f2 |> f3
```

Композиция

Прямой оператор композиции (forward composition operator) >> объединяет две функции, при этом функция слева вызывается первой. Также есть обратный оператор.

При использовании прямого конвейерного оператора необходимо указать переменную, чтобы «запустить» конвейерную обработку.

```
let square x = x * x
let toString (x : int) = x.ToString()
let strLen (x : string) = x.Length
let lenOfSquare = square >> toString >> strLen;;

lenOfSquare 5
```

Обратные операторы играют важную роль: они позволяют изменить порядок вычислений. Аргументы функций вычисляются в направлении слева направо, то есть если вы хотите вызвать функцию и передать ей результат другой функции, вы можете заключить выражение в круглые скобки или воспользоваться обратными операторами.

Лабораторная № 1 (основы языка F#)

1. Найти сумму кубов трех чисел.
2. Найти разность квадратов двух чисел.
3. Даны два числа. Найти среднее арифметическое их квадратов и среднее арифметическое их модулей.
4. Найти корни квадратного уравнения $ax^2+bx+c=0$, заданного своими коэффициентами a, b, c (коэффициент a не равен 0), если известно, что дискриминант уравнения неотрицателен.
5. Подсчитать сумму цифр трехзначного числа x.
6. Найти минимум двух максимумов: $\min(\max(a,b), \max(c,d))$.
7. Дано четырехзначное натуральное число n. Найти сумму первой и последней цифры этого числа.
8. Длины сторон первого прямоугольника a и b, его площадь в 6 раз меньше площади второго прямоугольника. Найти длину стороны второго прямоугольника, если длина одной из его сторон равна c.
9. Посчитать факториал числа(без рекурсии).
10. Найти факториал числа(с помощью рекурсии).
11. С помощью рекурсии вывести все числа до x, кратные трем.
12. Реализовать функцию, определяющую количество делителей числа.
13. Реализовать функцию, определяющую простое число или нет.
14. Выведите простые числа из промежутка от 1 до 100.
15. Написать функцию нахождения площади, периметра и длину диагонали прямоугольника.
16. Дано вещественное число A и натуральное число N. Вывести все целые степени числа A от 1 до N.
17. Написать функцию, выводящую элементы арифметической прогрессии до n-ного элемента.
18. Написать функцию, выводящую элементы геометрической прогрессии до n-ного.

19. Зная сумму членов арифметической прогрессии, их количество и d: вывести на экран эту прогрессию.(например s=100, n=20, d=2. Ответ: -14, -12...22,24).
20. Зная сумму членов геометрической прогрессии, их количество и q: вывести на экран эту прогрессию.
21. Вывести последовательность чисел Фибоначчи до n включительно.(1.1.2.3.5……..) $F(0)=1$, $F(1)=1$, $F(n)=F(n-1)+F(n-2)$
22. Определить при помощи конструкции if. В какой плоскости расположена точка. $A(x, y)$
23. Определить при помощи конструкции match. В какой плоскости расположена точка. $A(x, y)$
24. По номеру месяца, вывести его название.
25. Дано три числа. Найти наибольшее.

Часть 2. Списки

Списки объединяют данные в виде цепочки. Такой подход позволяет обрабатывать сразу все элементы списка с помощью агрегатных операторов. Самый простой способ объявить список состоит в том, чтобы указать список значений, разделенных точками с запятой, заключенный в квадратные скобки. Позднее вы узнаете, как объявлять списки с использованием более мощного синтаксиса генераторов списков (list comprehension syntax). Пустой список, не имеющий элементов, объявляется с помощью пустых квадратных скобок []:

```
let spisok = ["s"; "p"; "I"; "s"; "o"; "k"]
let spis2 = [1; 5; 3; 7]
let pustoi = []
```

В отличие от списков в других языках программирования, списки в языке F# имеют весьма ограниченные возможности доступа к элементам и управления ими. Фактически списки поддерживают всего две операции.

- Первая элементарная операция над списками – операция добавления, которая выполняется с помощью оператора ::. Этот оператор добавляет элемент в начало списка. В следующем примере цифра ‘1’ добавляется в начало списка ‘a’:

  ```
  let a = [2; 3; 4]
  let b = 1 :: a
  ```

- Вторая элементарная операция над списками – операция объединения, которая выполняется с помощью оператора @. Этот оператор объединяет два списка. В следующем примере выполняется объединение двух списков, ‘a’ и ‘b’, в результате чего создается новый список:

  ```
  let a = [2; 4; 6]
  let b = [1; 3; 5]
  let c = a @ b
  ```

Диапазоны списков

Объявление элементов списков в виде перечней значений, разделенных точками с запятой, быстро превращается в утомительное занятие, особенно при работе с длинными списками. Объявить список упорядоченных числовых значений можно с помощью синтаксиса диапазонов. Первое выражение в диапазоне определяет минимальное значение, а второе – максимальное. В результате получается список последовательных значений от минимального до максимального с шагом, равным 1:

```
let s = [1..10]
```

Если указано необязательное значение шага, то в результате будет создан список значений в диапазоне между двумя числами с заданным шагом. Обратите внимание, что шаг может иметь отрицательное значение:

```
let s = [1..2..10]
let s = [10..-3..1]
```

Генераторы списков

Наиболее выразительный способ создания списков заключается в использовании генераторов списков (list comprehensions), которые позволяют создавать списки непосредственно в коде F#. В самом простом случае генератор списков – это некоторый программный код, заключенный в квадратные скобки [ ]. Тело генератора списков будет выполняться, пока не завершится, а сам список будет создан из элементов, возвращаемых с помощью ключевого слова yield.

```
let spisok x =
   [
    yield x-1
    yield x
    yield x+1
   ]
```

Внутри генераторов списков допускается использовать практически любые возможности языка F#, включая объявления функций и циклы for. В следующем фрагменте демонстрируется генератор списков, который

объявляет функцию negate и возвращает числа от 1 до 10, инвертируя знак четных значений:

```
open System
[<EntryPoint>]
let main argv =
  let x =
    [
    let negate x = -x
    for i in 1 .. 10 do
     if i % 2 = 0 then
       yield negate i
     else
       yield i
    ]
  printfn "%A" x
  Console.ReadKey() |> ignore
  0
```

При использовании циклов for внутри генераторов списков программный код можно упростить, используя -> вместо do yield. Ниже приводятся два идентичных фрагмента кода:

```
let mult x = [ for i in 1 .. 10 -> x * i ]
```

Функции модуля List

Модуль List из стандартной библиотеки F# содержит множество методов, упрощающих обработку списков. Эти встроенные методы станут для вас основными инструментами обработки списков. Их можно увидеть и изучить, написав служебное слово «List» и поставив точку. Самые часто используемые рассмотрим ниже.

Функция ***List.map*** – это операция проекции, которая создает новый список на основе заданной функции. Каждый элемент нового списка является результатом выполнения функции.

В следующем примере посылается с помощью конвейера список [1..5] в List.map. Дальше, вместо каждого элемента возвращается случайное значение из промежутка и соединяется в список.

```
let rnd = System.Random()
let myList = [1..5] |> List.map (fun x -> rnd.Next(-50, 50))
```

Еще один пример, где возвращаются квадраты чисел:

```
let squares x = x * x
List.map squares [1 .. 10]
```

List.filter — Возвращает список, содержащий только те элементы, для которых указанная функция вернула значение true. В следующем примере возвращаются элементы, которые больше 10.

```
let rnd = System.Random()
let myList = [1..10] |> List.map (fun x -> rnd.Next(0, 50))
let pr x =
  x > 10
let delch lst =
  List.filter pr lst
let o = delch s
```

List.iter — Последний агрегатный оператор, List.iter, проходит по всем элементам списка и вызывает функцию, переданную вами в виде параметра. В примере функция List.iter используется для обхода всех чисел в списке и вывода их на консоль.

```
let print x = printfn "Вывод %A" x
   List.iter print [1 .. 5]
```

Лабораторная № 2 (генерация списков)

1. Дано натуральное число. Получить список из цифр этого числа.
2. Создать список по правилу: на нечетных местах стоят элементы, соответствующие номеру этого места, на чётных — нули.
3. Элементы в 10 раз больше номеров позиций, в которых они находятся.
4. Элемент совпадает с остатком от деления номера позиции, в которой он находится, на 3.
5. Создать список из делителей заданного числа.
6. Создать список из первых N кратных заданного числа.
7. Дано число N. Составить список из чисел, не превышающих N и не являющихся его делителями.
8. Составить список, в котором чередуются два заданных числа.
9. Составить список, в котором чередуются нули и единицы.
10. Составить список из букв латинского алфавита.

11. Составить список из чередующихся отрицательных и положительных значений из заданного диапазона.
12. Составить список из квадратов чисел из заданного диапазона.
13. Составить список из значений заданной функции на заданном отрезке.
14. Составить список, содержащий все четные числа от 1 до 100.
15. Создать список из простых чисел.
16. Создать список из пар случайных чисел.
17. Создайте список, который будет состоять из факториалов чисел от 1 до *n*.
18. Создайте список, состоящий из первых пяти степеней введенного числа.
19. Создайте список, состоящий из элементов арифметической прогрессии.
20. Создайте список, состоящий из элементов геометрической прогрессии.
21. Создайте список, состоящий из 10 элементов, начиная с числа «*x*».
22. Создайте список, состоящий из убывающей последовательности, начиная с числа «*x*».
23. Сформировать список по принципу [1; 2; 6; 24; 120; 720].
24. Сформировать список по принципу [1; 2; 4; 7; 11; 16].
25. Сформировать список по принципу [a, A; b, B; c, C…].

Лабораторная № 3 (обработка списков)

1. Найти сумму элементов списка.
2. Поменять порядок следования элементов списка на противоположный. Пример: [1,2,3,4] -> [4,3,2,1].
3. Исключить элементы на нечетных номерах.
4. Написать функцию, которая выведет все элементы списка по одному.
5. Определить, сколько раз встречается минимальный элемент в списке.
6. Определить количество нулей в списке.
7. Найти разность максимального и минимального элементов списка.
8. Найти сумму тех элементов списка, которые оканчиваются на 1.
9. Выбрать максимальный из модулей элементов списка.

10. В списке натуральных чисел подсчитать количество чисел, оканчивающихся заданной цифрой.
11. Найти сумму чётных элементов списка целых чисел.
12. Определить, сколько раз встречается максимальный элемент в списке.
13. Определить количество нечётных отрицательных элементов в списке целых чисел.
14. Указать минимальный элемент среди нечётных чисел в списке.
15. Определить количество ненулевых элементов списка.
16. Выяснить, упорядочены ли элементы списка по возрастанию.
17. Реализовать функцию, которая выдает первую позицию вхождения заданного числа в список.
18. Реализовать функцию, проверяющую, является ли строка палиндромом.
19. Подсчитать количество четных чисел в списке (с использованием стандартной функций map). Использование рекурсии не допускается.
20. Подсчитать количество четных чисел в списке (с использованием стандартной функций filter). Использование рекурсии не допускается.
21. Реализовать три варианта функции, подсчитывающей количество четных чисел в списке (с использованием стандартных функций fold). Использование рекурсии не допускается.
22. Дано два списка: первый содержит названия команд, второй заработанные очки. Соедините эти списки в список пар и выведите пару в которой число максимально.
23. Дан список. Создать второй, состоящий из true и false, при условии, что на соответствующих местах четных чисел из первого будет стоять – true, на местах нечетных – false.
24. Указать максимальный элемент среди четных.
25. Дан список символов. Определить количество символа, введенного с клавиатуры.

Часть 3. Последовательности

Наиболее часто отложенные вычисления используются при работе с последовательностями или типом seq, который представляет упорядоченную последовательность элементов, очень похожую на тип list. Для определения последовательностей можно использовать синтаксис генераторов списков (такие конструкции называются выражениями последовательности). Определение последовательности начинается с ключевого слова seq, за которым следуют фигурные скобки. В следующем фрагменте определяется последовательность из пяти элементов и выполняется итерация по ним с помощью функции Seq.iter. (Существует целый модуль, аналогичный модулю List, содержащий множество различных функций для работы с последовательностями.)

```
let UnderN n = seq { for i in 1 .. 2 .. n -> i }
Seq.iter (printfn "%A") (UnderN 10)
```

Различия между типами seq и list заключается в том, что в каждый конкретный момент времени в памяти существует только один элемент последовательности. Тип seq – это всего лишь псевдоним для интерфейса System.Collections.Generic.IEnumer able <'a> в платформе .NET.

Так как содержимое списка целиком хранится в памяти, это означает, что вы ограничены объемом имеющихся ресурсов. Вы легко можете определить бесконечную последовательность, но бесконечный список просто не поместится в память. Кроме того, при использовании списков значение каждого его элемента должно быть известно заранее, тогда как последовательности могут развертываться динамически (это так называемая «pull»-модель).

```
// Последовательность всех целых чисел
let allIntsSeq = seq { for i = 0 to System.Int32.MaxValue -> i }
```

Последовательности вычисляются отложено, поэтому каждый раз при получении очередного элемента выполняется код выражения последовательности.

Функции модуля Seq

Модуль Seq содержит множество полезных функций для работы с последовательностями. Большинство аналогичные с функциями из модуля List.

Функция	Описание
Seq.length	Возвращает длину последовательности.
Seq.exists	Возвращает признак наличия в последовательности элемента, который удовлетворяет функции поиска.
Seq.filter	Отфильтровывает все элементы последовательности, для которых указанная функция возвращает false.
Seq.concat	Объединяет серию последовательностей в единую последовательность seq.

В следующем примере с помощью *iter* происходит обход последовательности и вывод каждого элемента. Filter возвращает все элементы удовлетворяющие предикату *usl*. ToList преобразует последовательность в список.

```
open System
[<EntryPoint>]
let main argv =
  let posl = seq [1; 2; 6; 0; -7; 7]
  let usl pos =
       pos % 2 = 0
  Seq.iter (printfn "%A") posl
  let qwe = Seq.toList(Seq.filter usl posl)
  printfn "%A" qwe
  Console.ReadKey() |> ignore
  0
```

Интересно отметить, что выражения последовательности могут быть рекурсивными. С помощью ключевого слова *yield!* (произносится как йилд банг) можно вернуть подпоследовательность, которая объединяется с основной последовательностью.

Пример демонстрирует, как использовать ключевое слово *yield!* внутри выражения последовательности. Функция *allFilesUnder* возвращает значение типа *seq<string>* и получает имена всех файлов в указанной папке и

рекурсивно во всех подпапках. Функция *Directory.GetFiles* возвращает массив строк, содержащий имена всех файлов в папке *basePath*. Поскольку тип *array* совместим с типом *seq*, инструкция *yield!* возвращает все эти файлы.

```
let rec allFilesUnder basePath =
    seq {
     yield! Directory.GetFiles(basePath)
     for subdir in Directory.GetDirectories(basePath) do
       yield! allFilesUnder subdir
    }
```

Лабораторная работа № 4 (последовательности)

1. Для последовательности целых чисел найти сумму и количество отрицательных элементов.
2. Подсчитайте количество положительных, отрицательных и нулевых чисел последовательности.
3. Определите количество чисел, равных максимальному из всех чисел последовательности.
4. Определите номер первого и последнего максимального элемента последовательности.
5. Введите с клавиатуры число и определите, сколько раз оно встречается в последовательности.
6. Определите длину участка с максимальным количеством подряд идущих нулей.
7. Для последовательности целых чисел найти сумму и количество элементов, кратных 5 и не кратных 7;
8. В последовательности целых чисел найти номер последнего отрицательного элемента.
9. Для последовательности действительных чисел найти количество перемен знаков между последовательными элементами.
10. Для последовательности натуральных чисел определить количество членов этой последовательности являющихся квадратами четных чисел.

11. Является ли последовательность арифметической прогрессией? (Если да, вывести a_1 и d).
12. Определите и выведите все отрезки возрастания последовательности. Каждый участок выводится с новой строки.
13. Ввести с клавиатуры число и определить порядковый номер элемента последовательности, наиболее удаленного от введенного.
14. С помощью последовательности вывести все пути к текстовым файлам (.txt) в указанном каталоге и подкаталогах.
15. С помощью последовательности вывести все пути к не текстовым файлам в указанном каталоге и подкаталогах.
16. С помощью последовательности вывести список файлов в указанном каталоге.
17. Вывести самое длинное название самого файла в каталоге.
18. Вывести самое короткое название файла в каталоге.
19. Вывести первый по алфавиту файл.
20. Вывести последний по алфавиту файл.
21. Вывести количество файлов в указанном каталоге.
22. Вывести количество файлов в указанном каталоге и подкаталогах.
23. Выяснить есть ли в каталоге и подкаталогах файл с введенным названием.
24. Выяснить есть ли в каталоге файл с введенным названием.
25. Выяснить есть ли файлы в указанном каталоге.

Часть 4. Деревья

Важным рекурсивным типом данных, который часто встречается в функциональных программах, являются деревья. Определение дерева очень похоже на определение списка, однако, поскольку деревья встречаются в разных задачах, и их структура несколько варьируется, реализации деревьев нет в стандартной библиотеке F#.

В дискретной математике деревом называется ациклический связанный граф. В информатике обычно дают другое рекуррентное определение дерева общего вида типа T – это элемент типа T с присоединенными к нему 0 и более поддеревьями типа T. Если к элементу присоединено 0 поддеревьев, он называется терминальным, или листом, в противном случае узлом. В соответствии с этим дерево может быть представлено следующим образом:

Другой важной разновидностью деревьев являются двоичные деревья – такие деревья, у каждого узла которых есть два (возможно, пустых) поддерева – левое и правое. Интересной особенностью двоичных деревьев также является тот факт, что любое дерево общего вида может быть представлено в виде двоичного.

В соответствии с определением двоичных деревьев для их описания удобно использовать следующий тип:

```
type 't btree =
    Node of 't * 't btree * 't btree
    | Nil
```

Основная процедура обработки дерева – это обход, когда каждый элемент дерева посещается (то есть обрабатывается) ровно один раз. Существует несколько вариантов обхода дерева:

Порядок обхода	Название		Пример(для дерева выражения)
Корень – левое поддерево – правое поддерево	Прямой	Префиксный	+ * 1 2 3
Левое поддерево – корень – правое поддерево	Обратный	Инфиксный	1* 2 + 3
Левое поддерево – правое поддерево - корень	Концевой	Постфиксный	1 2 * 3 +

Программно они будут выглядеть так:

```
let prefix root left right = (root(); left(); right())
let infix root left right = (left(); root(); right())
let postfix root left right = (left(); right(); root())
```

Описав, таким образом, три порядка обхода (и, возможно, три сопряженных порядка), останется в самой процедуре обхода лишь описать три соответствующие функции для обработки корня, левого и правого поддеревьев и передать их как аргументы в переданный в виде аргумента порядок обхода trav:

```
let iterh trav f t =  //обход
    let rec tr t h =
        match t with
        Node (x,L,R) -> trav
                            (fun () -> (f x h)) // обход корня
                            (fun () -> tr L (h+1)) // обход левого поддерева
                            (fun () -> tr R (h+1)); // обход прав. поддерева
        | Nil -> ()
    tr t 0
```

Пример инфиксного обхода дерева с использованием этой процедуры:

```
let print_tree T = iterh infix (fun x h -> printfn "%s%A" (spaces h)x) T
```

Добавление в дерево поиска реализуется аналогичным образом – когда доходим до листа и понимаем, что элемента в дереве нет, присоединяем его к листу в соответствующем месте (слева или справа) в зависимости от значения ключа:

```
let rec insert x t =
  match t with
   Nil -> Node(x,Nil,Nil)
   | Node(z,L,R) -> if x<z then Node(z,insert x L,R)
```

```
                    else Node(z,L,insert x R)
```

Для добавления целого списка элементов в дерево можем воспользоваться сверткой, где дерево выступает в роли аккумулятора:

```
let list_to_tree L = List.fold (fun t x ->  insert x t) Nil L
```

Например, можно использовать такой список:

```
let L =
   [
    let r = new Random()
    for i in 1..5 do
      yield r.Next(0, 5)
   ]
```

Полный код генерирующий двоичное дерево поиска будет выглядеть следующим образом:

```
open System
type 't btree =                 //описание
    Node of 't * 't btree * 't btree
    | Nil
[<EntryPoint>]
let main (args : string[]) =
 let infix root left right = (left(); root(); right())  //порядок обхода

 let iterh trav f t =  //обход
    let rec tr t h =
        match t with
        Node (x,L,R) -> trav
                            (fun () -> (f x h)) // обход корня
                            (fun () -> tr L (h+1)) // обход левого поддерева
                            (fun () -> tr R (h+1)); // обход прав. поддерева
        | Nil -> ()
    tr t 0
 let spaces n = List.fold (fun s _ -> s+" ") "" [0..n]
 let print_tree T = iterh infix (fun x h -> printfn "%s%A" (spaces h)x) T
 let rec insert x t =          //вставка элемента
  match t with
   Nil -> Node(x,Nil,Nil)
   | Node(z,L,R) -> if x<z then Node(z,insert x L,R)
                    else Node(z,L,insert x R)
 let L =  //генерация списка
   [
    let r = new Random()
    for i in 1..5 do
      yield r.Next(0, 5)
   ]
 printfn "Исходный список %A" L
 let list_to_tree L = List.fold (fun t x ->  insert x t) Nil L
 let BT = list_to_tree L
 let A = print_tree BT
 printfn "(press any key to continue)"
 Console.ReadKey() |> ignore
 0
```

Лабораторная работа № 5 (деревья)

1. Сформировать дерево из случайных чисел и найти в нем количество единиц.
2. Сформировать дерево из случайных чисел и вывести на экран число и уровень, на котором оно находиться.
3. Сформировать дерево и выяснить есть ли в нем число, введенное с клавиатуры.
4. Вывести сумму элементов дерева.
5. Вывести вершину дерева.
6. Вывести значение максимального элемента дерева.
7. Вывести значение минимального элемента дерева.
8. Сформировать список из дерева.
9. Сформировать новое дерево из положительных элементов старого дерева.
10. Написать программу, которая строит список из всех листьев начального дерева.
11. Написать программу, которая находит в заданном непустом бинарном дереве длину (количество ветвей) пути от корня до ближайшей вершины с заданным элементом E.
12. Написать программу, которая определяет количество вхождений вершины с заданным элементом E в бинарное дерево.
13. Сформировать новое дерево из отрицательных элементов старого дерева.
14. Написать программу, которая определяет количество вхождений вершины с заданным элементом E в бинарное дерево.
15. Вывести значения узлов дерева.
16. Найти сумму четных значений, находящихся в листьях.
17. Вывести узлы с двумя листьями.
18. Вывести узлы с одним листом.
19. Вывести сумму положительных элементов дерева.
20. Найти количество отрицательных элементов дерева.
21. Выяснить каких чисел больше: положительных или отрицательных в дереве.
22. Выяснить, что больше: сумма четных чисел или сумма нечетных чисел в дереве.
23. Сформировать список из четных элементов дерева.
24. Выяснить на сколько больше максимальный элемент минимального.
25. Вывести положительные листья.

Приложение 3. Опубликованная статья

УДК 378

О ПОДХОДАХ К ПРЕПОДАВАНИЮ ФУНКЦИОНАЛЬНОГО ПРОГРАММИРОВАНИЯ (НА ПРИМЕРЕ F#)

Иванов Сергей Владимирович, Шестаков Александр Петрович

Пермский государственный гуманитарно-педагогический университет,
614990, Россия, г. Пермь, ул. Пушкина, 42, ivanov_s@pspu.ru

Рассматривается функциональный подход к программированию, его отличительные особенности, преимущества перед другими парадигмами программирования. Изучение функциональных языков программирования имеет определенную специфику, что требует особого дидактического обеспечения при организации обучения. Представлено примерное тематическое планирование для введения в функциональное программирование на базе F#. Описываются требования к знаниям, умениям обучаемых перед началом изучения данного курса и требования к знаниям, умениям обучаемых по завершению курса. Раскрывается содержание раздела «Функциональное программирование на F#» в рамках дисциплины «Языки программирования». Даются краткие методические рекомендации для изучения функционального программирования. Приведены сведения о результатах экспериментального преподавания функционального программирования на факультете информатики и экономики в ПГГПУ в рамках дисциплины «Языки программирования».

Ключевые слова: функциональное программирование, F#, тематическое планирование, дидактическое обеспечение.

На сегодняшний день функциональное программирование практически не изучается в учебных заведениях. А оно имеет ряд особенностей, отличных от императивного программирования, которые можно рассматривать как преимущества [3].

Особенности функционального программирования:

- чистота функции, нет побочных эффектов (не используются глобальные переменные). Функции можно использовать сколько угодно раз, не опасаясь за побочные

действия. Весь код разбит на отдельные функции, которые легко тестировать. Проще выявлять ошибки;

- неизменяемые данные;
- использование функций высших порядков;
- композиция (применение функции к результату другой);
- ленивые (lazy) вычисления [1].

Функциональное программирование можно изучать для ознакомления и расширения знаний обучающихся, как высших учебных заведений в рамках раздела дисциплины «Языки программирования», так и в качестве факультатива для учащихся старших классов общеобразовательной школы, обучающихся по профилю «Информатика и ИКТ». Для освоения данной дисциплины не требуется знаний других языков, но к этому времени чаще всего обучающиеся уже сталкивались с императивным программированием. Для изучения функциональной парадигмы рекомендуется выбрать Лисп или современный мульпарадигменный язык F#.

Таблица. Примерное тематическое планирование

Тема	Всего часов	Лекции	Лабораторные работы
1. Основы F# 1.1. Базовые операции и операторы 1.2.Функциональные возможности	4	2	2
2. Списки в F# 2.1. Генерация списков 2.2. Обработка списков	3	1	2
3. Последовательности в F# 3.1.Генерация последовательностей	3	1	2

3.2.Обработка последовательностей			
4. Деревья в F# 4.1. Формирование, виды обходов 4.2. Работа с деревьями	6	2	4
Итого 16 часов			

Каждая часть курса сопровождается подборкой практических задач для лабораторных работ; чаще всего задания являются индивидуальными.

Перед началом изучения функционального программирования обучающиеся должны:

- знать:
  - необходимые математические операции и функции;
  - логические операции (функции).
- уметь:
  - выполнять декомпозицию задач (разбивать сложные задачи на подзадачи);
  - применять математические знания на практике.

В результате освоения данного курса обучающиеся должны:

- иметь представление: о функциональной парадигме программирования, о функциях высших порядков и их применении, о списках, последовательностях, деревьях, об интерпретации и о компиляции функциональных программ;
- знать:
  - основные конструкции языка F#;
  - основные возможности функциональной парадигмы;
  - правила записи программ;
- уметь:
  - пользоваться конструкциями данного языка;

•решать типовые задачи;

•выделять подзадачи и строить функции для их решения;

•осуществлять отладку и тестирование программ на F#.

В первой части предлагается начать изучение с простых операций и схожих структур, используемых в императивном программировании: базовые арифметические операции, условный оператор, оператор выбора, функции, рекурсия. Особое внимание необходимо уделять изучению и применению рекурсии, так как она является основным используемым инструментом. Как показывает практика, на первых порах обучаемые забывают о правилах вложенности, из-за отсутствия явного выделения блоков скобками, как это происходит в большинстве других языков.

Далее идет обучение таким функциональным возможностям, как композиция функций, каррирование и функции высших порядков. Изложение данной темы целесообразно иллюстрировать значительным количеством примеров, что позволит продемонстрировать нюансы этих возможностей. Закрепление темы требует качественных практических заданий, в т.ч. комбинированного характера.

Дальнейшее освоение этого материала происходит при решении задач из этой и последующих тем.

При изучении списков необходимо наглядно показать их структуру: рекомендуется привести примеры из реальной жизни. Например, это можно с помощью коробков — положив внутрь записки с числами или еще какой-нибудь информацией или на стопке книг — в данном случае их можно рассматривать как список списков. Рекомендуется сначала изучить генерацию списков — как создавать их с использованием различных способов: с помощью циклов, с помощью рекурсии, с помощью клавиатурного ввода. А уже потом изучать обработку списков, используя изученные способы генерации. Рассмотреть основные функции работы над списками: нахождение длины списка, обход списка, свертка и т.д. [2]

В следующем разделе изучаются последовательности. При изучении этой темы часть сложностей снимается, так как последовательности схожи со списками. Но следует разграничить эти типы данных и показать, в чем различие между типами seq и list, а оно заключается в том, что в каждый конкретный момент времени в памяти существует только один элемент последовательности. Это дает возможность создавать последовательность очень больших размеров, в отличие от списка, где возможно переполнение.

Заключительная тема для изучения — деревья. Для успешного освоения учащимися необходимо дать им наглядное представление дерева и показать все методы обхода, а уже потом переходить к объяснению отдельных участков кода формирования дерева. Если у учащихся возникнут сложности — придется дать им готовый шаблон формирования дерева, который они будут дорабатывать, и решать свои задачи.

В результате экспериментального преподавания функционального программирования на факультете информатики и экономики в ПГГПУ в рамках дисциплины «Языки программирования» группа 1236 познакомилась с данной парадигмой и освоила в полном объеме подготовленный материал. Все обучающиеся выполнили и сдали необходимые лабораторные работы и были впечатлены новыми возможностями.

Освоение данной парадигмы: покажет новые методы обработки данных, которые не изучаются в других курсах программирования, даст хорошее понимание рекурсии.

Библиографический список

1. *Смит К.* Программирование на F#. — Пер. с англ. — СПб.: Символ-Плюс, 2011. — 448 с.
2. *Сошников Д. В.* Программирование на F#. – М.: ДМК Пресс, 2011. – 192 с.
3. *Интуит.* Национальный открытый университет [Электронный ресурс] URL: http://www.intuit.ru/studies/courses/471/327/info (дата обращения: 04.04.2017).

ABOUT THE APPROACHES TO FUNCTIONAL PROGRAMMING TEACHING (ON THE BASIC OF F#)

Ivanov Sergej V., Shestakov Alexander P.

Perm State Humanitarian-Pedagogical University, 15, Pushkina st., Perm, 614990, Russia, ivanov_s@pspu.ru

There considered functional programming approach, its distinctive features and advantages over other programming paradigms. Studying of functional programming languages has some specificity, what requires special didactic support during the arrangement of teaching. There presented approximate thematic planning for the introduction in the functional programming on the basic of F#. There described requirements for students' knowledge and skills in the beginning of this course and in the ending of this course. There opened the contents of chapter "Functional Programming on F#" in the discipline "Programming Languages". There given quick guidelines for the studying of functional programming. There provides information about the results of experimental teaching of functional programming at the faculty of Information Technologies and Economics in PSHPU in the discipline "Programming Languages".

Keywords: functional programming, F#, thematic planning, didactic support.

Приложение 4. Диск с материалами работы