ВЫСШЕЕ ПРОФЕССИОНАЛЬНОЕ ОБРАЗОВАНИЕ

Г. М. СЕРГИЕВСКИЙ, Н. Г. ВОЛЧЁНКОВ

ФУНКЦИОНАЛЬНОЕ И ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

Допущено
Учебно-методическим объединением вузов
по университетскому политехническому образованию
в качестве учебного пособия для студентов
высших учебных заведений, обучающихся
по направлению «Информатика и вычислительная техника»

ACADEMIA

Москва
Издательский центр «Академия»
2010

УДК 681.3.06(075.8)
ББК 32.973-018.2я73
С323

Рецензенты:
зам. директора Департамента промышленной автоматизации ЗАО «Стинс Коман» *Р. В. Душкин*;
начальник сектора Радиотехнического института, канд. физ-мат. наук, старший научный сотрудник *С. В. Попов*

Сергиевский Г. М.
С323 Функциональное и логическое программирование : учеб. пособие для студ. высш. учеб. заведений / Г. М. Сергиевский, Н. Г. Волчёнков. — М. : Издательский центр «Академия», 2010. — 320 с.
ISBN 978-5-7695-6433-8

Рассмотрены основные результаты как в теоретической части, так и в части практического применения, накопленные к настоящему времени в области функционального и логического программирования. Показано, что оба эти подхода, относящиеся к парадигме декларативного программирования, позволяют получить новые возможности в части трансформации и автоматического синтеза программ, доказательства свойств программ, частичных вычислений и др. Описаны области, в которых применение данных подходов имеет преимущества по сравнению с операторным программированием. Практические аспекты функционального программирования изучаются на примере языков Haskell — лучшей современной реализации функциональной парадигмы. В теоретическом обосновании приведены наиболее важные (для данных целей) результаты лямбда-исчисления и комбинаторной логики.

Представлена наиболее «продвинутая» практическая реализация идеи логического программирования: язык Пролог. Даны его детальное описание и приемы программирования. Основное внимание уделено таким областям применения Пролога, как программирование баз данных, синтаксический анализ, реализация переборного и эвристического поиска, задачи искусственного интеллекта, в том числе обработки нечетких данных, программирование в ограничениях (Constraint Lodic Programming). Подробно описаны теоретические основы логического программирования (метод резолюций, теорема Робинсона и др.).

Для студентов учреждений высшего профессионального образования.

УДК 681.3.06(075.8)
ББК 32.973-018.2я73

ISBN 978-5-7695-6433-8

СОДЕРЖАНИЕ

ПРЕДИСЛОВИЕ

Функциональное программирование (ФП), а во многих случаях и логическое программирование (ЛП) являются обязательными дисциплинами в большинстве западных университетов для студентов, изучающих информатику. Это связано с тем, что именно в них в наибольшей степени удается использовать теоретические методы исследования программ, которые в целом представляют собой неудобный объект для применения математических методов. Благодаря этому ФП и ЛП являются источником идей, приемов, технологий, которые в большей или меньшей степени используются во всех парадигмах программирования: частичные вычисления (суперкомпиляция), синтез программ, «ленивые вычисления» и т.п. На сегодня ФП и ЛП посвящены десятки монографий, сотни сайтов и трудно оцениваемое число публикаций. Поэтому необходимость изучения российскими студентами современного состояния теории и практики в этой области представляется весьма актуальной. Учебное пособие составлен по материалам курсов лекций, в течение многих лет читаемых авторами на кафедре кибернетики Московского инженерно-физического института (МИФИ). В нем достаточно полно (насколько позволяет объем учебного пособия) представлено современное состояние идей, методов и программных средств в области функционального и логического программирования. Кроме теоретических вопросов в учебное пособие входят краткий исторический экскурс, описание интернет-ресурсов, а также примеры и упражнения, позволяющие овладеть практикой базового программирования на языках Лисп, Haskell и Пролог.

Учебное пособие состоит из трех разделов: раздел I посвящен общим вопросам декларативного программирования, раздел II — ФП и раздел III — ЛП. Программно независимые вопросы ФП изложены на абстрактном языке, что позволило представить их в очень компактной форме и затронуть многие из тем, которые слабо отражены в русскоязычной литературе (синтаксически-ориентированные методы программирования, методы доказательства свойств программ, трансформации программ, частичные вычисления и, конечно, программирование на языке Haskell).

Описание логического программирования начинается с введения в «чистое» логическое программирование. Подробно рассматривается реализация идей логического программирования в Прологе —

алгоритмически полном языке, предназначенном для решения задач символьной обработки, эвристического поиска, некоторых задач искусственного интеллекта. Представлен краткий обзор систем и диалектов Пролога, их возможностей, особенностей и реализованных механизмов. В конце учебного пособия помещен список использованной и рекомендуемой литературы. Ссылки в тексте на книги, приведенные в этом списке, даны в круглых скобках, внутри которых через запятую указаны фамилия автора (или первого из авторов) и год выпуска книги, например (Бердж, 1983).

Предисловие, гл. 1, подразд. 2.1, гл. 3 — 14 написаны Г. М. Сергиевским, подразд. 2.2, 2.3, гл. 15 — 23 — Н. Г. Волчёнковым. Разделы II (ФП) и (ЛП) изложены таким образом, чтобы их можно было изучать независимо друг от друга, избежать дублирования изложения и обеспечить достаточную маневренность при изучении. Однако по мнению авторов предпочтительней эти темы изучать последовательно.

Авторы выражают благодарность всем преподавателям и студентам кафедры, помогавшим в написании данного учебного пособия, а также всем авторам, материалы которых были использованы при его написании, принося заранее извинения за невозможность сослаться в тексте на все случаи такого использования. Особая благодарность принявшему эстафету чтения курса ФП на кафедре Р. В. Душкину, усилиями которого было внедрено изучение Haskell (что в большой степени определило содержание данного раздела в учебном пособии), а также Антону Лебедеву, проделавшему большую работу по отладке примеров на языках Лисп и Haskell.

РАЗДЕЛ I
ОСНОВАНИЯ ФУНКЦИОНАЛЬНОГО И ЛОГИЧЕСКОГО ПРОГРАММИРОВАНИЯ. ИСТОРИЧЕСКАЯ СПРАВКА

Глава 1

Парадигмы программирования

Если проанализировать способы представления знаний, применяемые в практике человеческой деятельности, то нетрудно убедиться, что в подавляющем большинстве случаев это тексты, состоящие из повествовательных предложений на каком-либо (русском, английском и т. п.) языке. Для представления более формальных знаний применяются математические конструкции (часто дополняемые графическими элементами), которые также преимущественно имеют форму некоторых утверждений. И совсем редко, только в случаях, когда требуется особо точно передать информацию о реализации некоторых действий, применяется форма в виде последовательности инструкций из повелительных предложений.

Очевидно, что при общении с компьютером ситуация прямо противоположная: знания, как правило, представляются в форме программы, которая есть не что иное, как последовательность инструкций (команд). И хотя в программах на языках высокого уровня и присутствует (в виде объявлений) «повествовательная компонента», основную часть программы составляет именно последовательность команд. Такую гегемонию представления знаний в виде последовательности команд и пытается нарушить функциональное и логическое программирование. Рассмотрим более подробно, в каких случаях и каким образом может использоваться повествовательный способ представления знаний в ЭВМ.

1.1. Декларативные и процедурные способы представления знаний

Для получения решения задачи, например уравнения с использованием программных средств, можно действовать двумя путями:

написать непосредственно программу, реализующую алгоритм решения данного конкретного уравнения, или попытаться вначале реализовать программу решения целого класса уравнений, частным случаем которого является заданное уравнение. Тогда для решения задачи достаточно написать только условия задачи (уравнение) в соответствии с правилами реализованного «решателя».

Несомненно, при наличии «решателя» второй способ предпочтительней — вместо описания «как» (как решается задача) достаточно описать «что» (что должно быть решено). Поскольку по традиции программа понимается как последовательность команд, выполняемых компьютером, первый способ носит название *императивного*, или *процедурного* (так как команда с точки зрения классификации наклонений относится к императивному наклонению), а второй — *декларативного* (так как описание условий задачи без описания алгоритма решения представляется повествовательным или декларативным наклонением).

Первый способ обладает максимальной универсальностью, так как может быть применен для любой задачи, если существует и известен алгоритм ее решения. При этом процесс решения может быть сделан максимально эффективным, поскольку при построении алгоритма можно учесть все особенности данной конкретной задачи, позволяющие оптимизировать ее решение. Однако за это нужно заплатить и высокую цену — программно реализовать алгоритм решения на каком-либо императивном (процедурном) языке.

Второй способ в общем случае менее универсален, так как для каждого класса задач нужен свой язык описания задачи и свой «решатель» для задач, написанных на этом языке. Например, при решении задачи линейного программирования для описания условий задачи (системы линейных неравенств и оптимизируемой линейной формы) нужен язык, на котором можно будет описать только эти условия и ничего больше. Решатель, или интерпретатор языка описания задачи, должен реализовывать какой-либо из известных способов решения данного класса задач (например, *симплнекс-метод*). Очевидно, что если задача может быть формализована в терминах языка описания задач какого-либо «решателя», то ее решение может быть получено с минимальными затратами. Однако справедлив тезис: чем шире класс задач, которые умеет решать «решатель», тем более универсальные и, следовательно, менее эффективные методы решения он должен использовать, и наоборот.

Закономерен вопрос: «Можно ли «придумать» декларативные языки, т.е. языки, описывающие «что», а не «как», которые были бы универсальными, т.е. пригодными для описания любой задачи, и какие математические понятия можно положить в их основу?». Одними из наиболее универсальных понятий в математике являются понятия ***функции*** и ***предиката***, именно они легли в основу современных языков функционального и логического программирования.

При ФП задача специфицируется в виде набора определений функций, который можно назвать *библиотекой*, или *базой данных*, и некоторого выражения с фактическими параметрами, которое можно назвать *целевым выражением*, или *запросом*. Целью решения задачи является вычисление значения этого выражения, содержащего обычно обращения к встроенным и (или) определенным в базе данных функциям. В целях большей универсальности вид выражения заранее не фиксируется и пользователь может формулировать его произвольно.

При ЛП база данных описывается набором *логических правил* и *фактов*, которые можно рассматривать как аксиомы предметной области. Запрос представляет собой некоторую логическую формулу, которая, в отличие от предыдущего случая может содержать переменные. Целью решения является попытка доказать, что эта формула есть логическое следствие аксиом с определением всех вариантов означивания переменных в запросе, либо получить сообщение, что запрос не является логическим следствием.

Для практического применения данных подходов необходимо конкретизировать способы, с помощью которых строятся определения функций (в ФП) или логических формул (в ЛП). Понятно, что чем меньше ограничений накладывается на допустимые способы определения функций (или логических формул), тем проще пользователю формализовать свою задачу. Однако тем менее эффективным будет алгоритм интерпретации. На сегодня в парадигме и функционального, и логического программирования выработаны общепринятые стандарты таких определений, которые являются разумным компромиссом между выразительностью и эффективностью. Наиболее важные стандарты ФП и ЛП приведены в следующих главах.

Рассматривая декларативные и императивные парадигмы, нельзя не затронуть вопрос о месте в этом ряду объектно-ориентированного программирования (ООП). Данные языки содержат значительную декларативную компоненту — описание классов (механизм классов, правда, в несколько ином понимании, присутствует в соответствующих версиях ФП и ЛП). Однако сама вычислительная модель, лежащая в основе этих языков (в виде множества статически или динамически определенных объектов, обменивающихся сообщениями), и методы, изменяющие значения атрибутов, имеют явный процедурный характер. Поэтому данные языки с точки зрения рассматриваемой классификации ближе к императивным языкам. Вообще деление языков на императивные, ФП, ЛП и ООП в определенной мере условно — можно разрабатывать вполне объектно-ориентированные программы на ассемблере, хотя это и очень сложно. Можно следовать функциональной парадигме, используя язык С. Однако наиболее просто и естественно выглядят программы, написанные в соответствии с той парадигмой, для поддержки которой предназначен данный язык.

1.2. Две семантики декларативных языков

Характерной особенностью декларативных языков является возможность различения у них двух семантик: декларативной и процедурной.

Декларативная семантика программного текста на декларативном языке определяется использованием тех математических конструкций, которые положены в основу данного языка. Так, с декларативной точки зрения, программный текст, например на функциональном языке, понимается как набор математических определений функций, оформленных в соответствии со стандартами языка. Однако данный текст можно воспринять по-другому, если в центр внимания положить представление о том, как будет работать интерпретатор при его интерпретации. Если каждый программный текст рассматривается с точки зрения вычислительных процедур, которые он порождает в соответствии с реализованными правилами интерпретации, то говорят о ***процедурной***, или ***операционной***, ***семантике*** языка. Нетрудно видеть, что при одной и той же декларативной семантике может быть много разных процедурных семантик в соответствии с разными алгоритмами решения задач, которые могут быть реализованы в интерпретаторах.

Выбор процедурной семантики, при которой результат вычислений программы с помощью интерпретатора совпадает с результатом понимания данной программы в соответствии с ее декларативной семантикой, является чрезвычайно важным. В противном случае необходимо признать, что алгоритм интерпретации некорректен. Единственным исключением, которое в этом случае допускается, является сужение области определения при интерпретации по сравнению с декларативным пониманием программы (например, в ФП это означает, что программа может зациклиться на тех исходных данных, на которых функция, определяемая в соответствии с декларативным пониманием программы определена). Этот вопрос мог бы стать серьезным препятствием для развития ФП и ЛП, однако из-за наличия математических оснований (лямбда-исчисления для ФП и исчисления предикатов первого порядка для ФП) удалось математически обосновать принятые правила интерпретации.

И с теоретической и тем более с практической точек зрения чрезвычайно важен вопрос: «Можно ли, используя ФП (ЛП), писать программы, не принимая во внимание процедурную семантику?». Ответ ожидаем: «Нет, нельзя». Природу обмануть нельзя: именно непонимание этого факта привело ко многим заблуждениям представителей искусственного интеллекта в 1980-х гг. Полное игнорирование процедурной семантики может превратить программу в «исполняемую спецификацию», т.е. формально правильное определение задачи, но совершенно непригодное для практического использования из-за крайней вычислительной неэффективности. Для превращения спе-

цификации в реальную программу ее необходимо оптимизировать, используя какую-либо эвристику (догадку), что с успехом делает опытный программист.

1.3. Область применения языков функционального и логического программирования

Закономерны вопросы, чем объясняется интерес к ФП и ЛП, дает ли их применение какие-либо преимущества по сравнению с применением императивных или ООП языков, в чем состоят эти преимущества и какова сфера применения ФП и ЛП (если такие преимущества есть). Для ответа на эти вопросы нужно рассмотреть теоретические и практические аспекты.

В практическом плане данные языки как языки программирования особенно удобны для реализации обработки типов данных, которые сами имеют рекурсивную природу: списков, деревьев, графов и сводящихся к ним структур. Такого рода задачи характерны для обработки символьной информации, т. е. для создания трансляторов и решения задач искусственного интеллекта: обработки естественного языка, трансформации и автоматического синтеза программ, аналитического преобразования формальных текстов и др. В этих случаях программы, написанные на ФП (ЛП), обычно в 5 — 10 раз короче программ, написанных в императивном стиле. При этом, что не менее важно, эти программы гораздо понятнее, их легче писать и отлаживать. Более того, существуют специальные технологии (см. гл. 8), с помощью которых написание программ из полуинтуитивного процесса превращается в науку (при условии, что известно описание обрабатываемых типов данных).

Бытует мнение, что функциональное или логическое программирование ведет к потере производительности. Это утверждение справедливо только в той степени, в какой можно сказать, что за возможность писать на языке высокого уровня надо платить. Действительно, за счет сокращения времени написания программы (из-за значительного сокращения размеров текста) определение многих вычислений, которые пришлось бы описывать на языках более низкого уровня, можно осуществить, используя механизмы генерации кода транслятором. Будучи универсальными, эти механизмы генерируют программы, которые потенциально могут быть оптимизированы в каждом конкретном случае путем написания кода вручную. В то же время, если при написании программы тщательно следить за эффективностью кода (например, максимально использовать в ФП так называемую «хвостовую рекурсию»), то такая программа после компиляции не уступит в эффективности программам, написанным на императивных языках. Более того, в некоторых

случаях этот процесс оптимизации может быть автоматизирован за счет использования механизмов программных трансформаций (см. гл. 10). Кроме того, увеличение мощностей вычислительной техники сводит на нет возможную потерю производительности в общем случае. По данным (Душкин, 2007) для некоторых задач, реализуемых на языке Haskell (см. гл. 10), наблюдалось увеличение производительности до 2 раз по сравнению с языком C.

Подтверждением возможности использования языков функционального программирования для создания коммерческих продуктов может служить реализация известного пакета Autocad с использованием языка Лисп. Появились также сообщения о реализации Канадской национальной системы бронирования авиабилетов на языке Python, поддерживающем функциональную парадигму.

Многие идеи ФП нашли свое отражение в развиваемой компанией Microsoft технологии .NET. Разработчики .NET все-таки решили добавить в C# и другие языки платформы .NET проблемно-ориентированное расширение LINQ, поддерживающее интегрированный язык запросов. В LINQ можно использовать λ-функции, которые представляются в виде деревьев выражений. В C# в определенной степени поддерживаются также идеи «ленивых вычислений».

Последнее время интерес к языкам ФП и ЛП заметно растет в связи со свойственной им возможностью естественного распараллеливания вычислений, что делает их гораздо более перспективными в условиях появления многоядерных вычислительных комплексов.

Наконец, данные языки обладают еще одним аспектом, делающим их обязательным объектом теоретического программирования (Computer Science). Дело в том, что благодаря присущей им декларативной семантике программы на этих языках представляют собой уже готовые математические объекты, которые реально могут исследоваться и обрабатываться с использованием математических методов. Это особенно важно в связи с тем, что программы, написанные на императивных или ОО-языках, практически не поддаются исследованию такими методами. Хотя эта область исследований находится еще в начальной стадии, именно она обещает прорыв в будущем.

ВЫВОДЫ

1. Процедурный способ программирования соответствует вопросу «**как**» (необходимо описать, как решается задача), декларативный способ — вопросу «**что**» (достаточно описать, что должно быть решено).

2. Программа на декларативном языке состоит из двух компонент: условия задачи (которую иногда называют «базой данных») и целевого запроса.

3. Для декларативного программирования необходимо наличие «решателя» (называемого обычно интерпретатором), который «знает» как выполнить целевой запрос, исходя из условий, представленных в «базе данных».

4. В каждом декларативном языке программирования можно выделить две семантики: декларативную и процедурную. Декларативная семантика определяется в соответствии с семантикой тех математических конструкций, которые положены в основу данного языка, а процедурная — с помощью принятого механизма интерпретации этих конструкций.

5. Необходимым условием объявления некоторого декларативного языка является корректность предложенной для него процедурной семантики (правил интерпретации), т.е. совпадение результатов вычислений любой программы с результатами, имеющими место при восприятии этой программы как математического текста.

6. Корректность механизмов вычислений, используемых в современных интерпретаторах ФП и ЛП, обоснована теоретическим аппаратом лямбда-исчисления для ФП и исчисления предикатов первого порядка для ФП.

7. Применение парадигм ФП и ЛП дает наибольший выигрыш при работе с данными, которые сами имеют рекурсивную природу: списками, деревьями, графами и сводящимися к ним структурами.

8. Задачи такого рода характерны для обработки символьной информации: создания трансляторов, обработки естественного языка, трансформации и автоматического синтеза программ, аналитических преобразований формальных текстов и др.

9. Существуют специальные технологии (см. гл. 8), с помощью которых при известном описании обрабатываемых данных написание программ из полуинтуитивного процесса превращается в достаточно строгую процедуру.

10. Программы ФП и ЛП являются гораздо более приспособленными для реализации параллельных вычислений, чем традиционные программы.

Контрольные вопросы и задания

1. Почему среди декларативных языков наибольшее применение получили языки ФП и ЛП?

2. Перечислите математические понятия, которые лежат в основе функционального и логического программирования.

3. Какой способ программирования потенциально обеспечивает более высокую эффективность вычислений и почему?

4. Определите назначение «решателя» (интерпретатора) при декларативном подходе.

5. Как представляется «база данных» в ФП и ЛП?

6. Чем может отличаться запрос в ФП от запроса в ЛП?

7. Может ли декларативный язык иметь несколько процедурных семантик?

8. Какая некорректность процедурной семантики допускается для функциональных языков?

9. Можно ли при использовании декларативных языков полностью отказаться от учета процедурной семантики?

10. Какими свойствами обладают программы на ФП- и ЛП-языках?

11. Что можно сказать о вычислительной эффективности программ ФП и ЛП?

12. Приведите примеры применения ФП и ЛП.

13. За счет чего возможно применение математических методов для анализа, трансформации и синтеза программ ФП и ЛП?

Глава 2

Историческая справка и интернет-ресурсы

Опытный лингвист знает, что для профессионального владения каким-либо языком, понимания внутреннего механизма его развития необходимо рассматривать язык не только в синхроническом (по состоянию на текущий момент), но и диахроническом (с точки зрения развития во времени) аспектах.

2.1. Краткая история развития языков функционального программирования

Еще в 1924 г. М. Шейнфинкель (Moses Schonfinkel) разработал простую теорию функций, которая фактически явилась исчислением объектов-функций и предвосхитила появление ***лямбда-исчисления***. Собственно лямбда-исчисление, или исчисление лямбда-конверсий, было предложено в 1934 г. А. Чёрчем (Alonso Church), который применил его для исследования теории множеств. Вклад ученого был фундаментальным, его теория до сих пор называется лямбда-исчислением и часто именуется в литературе лямбда-исчислением Чёрча.

С первого взгляда кажется невозможным, что такой весьма простой язык для формализации функций, как язык лямбда-исчисления, является вполне достаточным для вычисления всего, что в принципе может быть вычислено. Однако доказательство выразимости в лямбда-исчислении рекурсивных функций (которые, как известно, являются одним из вариантов формализации понятия алгоритма) явилось причиной того, что лямбда-исчисление стало таким же известным и достаточно популярным математическим механизмом для описания вычислительных процессов, как машина Тьюринга или алгоритмы Маркова.

Вместе с тем другие знаменитые математики и логики также разрабатывали различные инструменты и формализмы, похожие на лямбда-исчисление. Так, в 1940 г. Х. Карри (Haskell Curry) создал теорию функций без переменных (иначе называемых комбинаторами), известную в настоящее время как ***комбинаторная логика***. Эта теория является развитием лямбда-исчисления и представляет собой формальный язык, который благодаря трудам Д. Тернера (David Turner) в 1960-х гг. положил начало целому направлению в ФП. Тернер предложил применять комбинаторы в качестве низкоуровневого кода для

трансляторов языков функционального программирования, т. е. предвосхитил появление абстрактных машин, использующих в качестве инструкций комбинаторы. Тот факт, что в истоках ФП лежат чисто математические основания, свидетельствует, что само по себе функциональное программирование является не только прикладной областью технологии, но и отдельным направлением знания в рамках дискретной математики.

Лямбда- и комбинаторные исчисления так бы и оставались достаточно экзотической ветвью дискретной математики, если бы в начале 1960-х гг. Джон Маккарти (John McCarthy) не увидел возможности создания на этой основе языка программирования, получившего название ***Лисп*** (Lisp — сокращение от List Processing). Лисп стал первым функциональным языком и на протяжении многих лет оставался единственным, породив своим появлением новую парадигму программирования — парадигму ФП. Этот язык содержал много новаторских идей, ставших потом классикой ФП: использование функций как основного механизма организации вычислений, введение понятия атома и использование списков как основного типа для представления как данных, так и программ и др. Язык Лисп развивался преимущественно в университетской среде, что при отсутствии на протяжении значительного времени стандарта на него породило многочисленные диалекты, наиболее известными из которых являются Common Lisp и Scheme.

Так как любой объект и любое определение на языке Лисп описываются с помощью списка, а все списки записываются с использованием круглых (обычных) скобок, то программы на нем выглядят весьма специфично, что создат сложности начинающиему программисту.

Язык Лисп как первый язык подобного рода был по существу квазифункциональным — в нем разрешалось использовать императивные средства, а также не поддерживались или поддерживались слабо многие формализмы и свойства, определенные функциональной парадигмой. Кроме того, определенная эклектичность Лиспа, в частности, за счет неограниченного применения императивных средств в противоположность более поздним разработкам, слишком строго выдерживающим чистоту теоретических концепций, делает его более приспособленным для создания реальных практических приложений (хотя и создает свои проблемы). Поэтому, несмотря на создание новых языков ФП, Лисп до сих пор остается реально применяемым языком функционального программирования.

Став основой функциональной парадигмы, Лисп послужил также мощным стимулом для дальнейших исследований и разработок. Следующим функциональным языком программирования (одним из самых первых после Лиспа), был ***APL***. Этот язык программирования позволял легко описывать математические идеи и имел мощнейшие механизмы для манипуляции массивами.

В конце 1970-х гг. Дж. Бэкус (John Bekus) — руководитель команды, разработавшей первый высокоуровневый язык программирования Фортран, а также изобретатель формы Бэкуса — Наура, в лекции на тему «Можно ли освободить программирование от стиля фон Неймана?» выдвинул идею о необходимости разработки аппликативных систем, позволяющих работать с функциями высших порядков и синтаксисом, приближенным к нотации комбинаторной логики. Развивая идеи APL, он определил ***FP***-системы, которые оказали большое влияние на все последующие функциональные языки программирования. Среди главных мотиваций данного подхода было желание преодолеть вычислительную неэффективность языков, основанных на фон-неймановской архитектуре с последовательным выполнением программ, а также стремление определить язык таким образом, чтобы программы на нем обладали «алгебраическими свойствами», позволяющими преобразовывать их в соответствии с законами такой алгебры.

Еще одно направление развития было связано с интенсивно разрабатывавшимися в конце 1970 — начале 1980-х гг. ***моделями типизации***. Исследователи из Эдинбургского университета во главе с Робертом Милнером (Robin Milner), одним из создателей системы типизации, занимавшиеся автоматическим доказательством теорем, поняли, что им необходим язык описания стратегий для поиска доказательств. Для этих целей они определили семантику языка ***ML*** (от MetaLanguage — метаязык). Однако через некоторое время они обнаружили, что новый язык обладает достаточными выразительными свойствам для того, чтобы быть языком программирования общего назначения. Кроме того что он мог работать с функциями высших порядков (таких, как FP), у него была такая немаловажная особенность, как автоматический вывод типов, т.е. способность получать тип выражения без явного описания самого типа. Этот язык также до сих пор широко используется. Более того, в некоторых учебных заведениях он преподается как базовый функциональный язык программирования. Как и в случае с языком Лисп, у ML имеется множество диалектов. Два самых известных диалекта — Standard ML и Caml.

Другим языком, похожим на ML, стал язык ***Miranda***, разработанный в 1985 — 1986 гг. Дэвидом Тернером (David Turner). Это был один из первых *ленивых функциональных языков программирования*. Такие языки пытаются отложить вычисления выражения до тех пор, пока не возникнет потребность в этом. Такой подход позволил пересмотреть семантику подобных языков и разработать в них механизмы, которые обрабатывают потенциально бесконечные структуры данных.

В середине 1980-х гг. в лабораториях телекоммуникационной компании Ericsson для управления параллельными вычислительными процессами был разработан язык, поддерживающий многие возмож-

ности функциональной парадигмы и широко используемый в промышленности, — ***Erlang***. Хотя этот язык программирования планировался для использования исключительно в телекоммуникации, он является языком общего назначения.

В результате стихийного развития вышло так, что практически каждая группа разработчиков и исследователей, занимавшаяся функциональным программированием, использовала собственный функциональный язык. Это препятствовало дальнейшему распространению данных языков и порождало многочисленные проблемы из-за необходимости постоянно отвлекаться на доработку трансляторов этих языков. Чтобы исправить ситуацию, объединенная группа ведущих исследователей в области функционального программирования решила воссоздать достоинства различных языков в новом универсальном функциональном языке. Первая реализация такого языка, названного ***Haskell*** в честь Хаскелла Карри (Haskell Curry), была создана в начале 1990-х гг. В настоящее время действителен стандарт ***Haskell-98***.

Нельзя не упомянуть и о достаточно известной в кругу любителей функционального программирования оригинальной парадигме ФП, связанной с языком ***Рефал***. Первая версия языка Рефал (РЕкурсивных Функций АЛгоритмический язык) была создана в начале 1960-х гг. Валентином Федоровичем Турчиным в качестве метаязыка для описания семантики других языков. Впоследствии в результате появления достаточно эффективных реализаций на ЭВМ он стал находить практическое использование в качестве языка программирования.

Рефал — язык бестиповой. В его основе лежит понятие объектного выражения в виде двунаправленного списка как универсального типа данных, который принципиально отличается от списков, используемых в рассмотренных ранее языках. Его минимальная версия получила название Базисный рефал. Диалект Базисного рефала под названием ***Рефал-2*** был реализован на многих типах отечественных ЭВМ и долгое время играл роль де-факто стандарта языка Рефал.

В середине 1980-х гг. В. Ф. Турчиным был предложен язык ***Рефал-5***, который содержит Базисный рефал в качестве подмножества. Расширения языка Рефал-5 качественно меняют стиль программирования, поэтому можно говорить о нем как о новом поколении языка. В настоящее время существуют две реализации языка Рефал: ***Рефал-6*** и ***Рефал Плюс***. Следует отметить, что Рефал явился первым языком функционального программирования, для которого был создан суперкомпилятор. В настоящее время разработчиков Рефала интересует не только суперкомпиляция (частичные вычисления), но и возможность использовать Рефал в качестве языка обработки xml-документов, поскольку, как оказалось, объектные выражения (или *R*-выражения) — основной и единственный тип данных Рефала — идеально подходят для представления xml-документов.

2.2. Краткая история развития языков логического программирования

Идеи логического программирования высказывались еще в начале 1930-х гг., когда французский математик Жак Эрбран (1908 — 1931) предложил теорему, которую можно трактовать как алгоритм *автоматического доказательства теорем* (АДТ) в исчислении предикатов первого порядка[1]. Через 30 лет использование теоремы Эрбрана в целях АДТ впервые было реализовано американским ученым Ван Хао.

В 1965 г. Дж. Робинсон модифицировал алгоритм Эрбрана так, что он стал пригоден для использования в ЭВМ, и разработал эффективный алгоритм унификации, составляющий базис его метода («принцип резолюции»).

Попытки реализовать некоторые идеи логического программирования делались в 1960-е гг. при разработке языков ***QA3*** и ***QA4***, а также Хьюиттом и Зусманом из MTI (Массачусетского технологического института) при создании языков ***Плэнер*** **(*Planner*)** и ***Микро-Плэнер*** **(*Micro-Planner*)**.

На рубеже 1960 — 1970-х гг. теория логического программирования совершенствовалась во многом благодаря работам Роберта Ковальского (Лондонский и Эдинбургский университеты), в частности, его работе «Логика предикатов как язык программирования» (1974 г.), в которой он показал, что для достижения эффективности логического вывода нужно ограничиться использованием лишь *хорновских дизъюнктов* — формул вида $P_1 \,\&\, ... \,\&\, P_n \rightarrow Q$ — вместо логических выражений общего вида $P_1 \,\&\, ... \,\&\, P_n \rightarrow Q_1 \vee ... \vee Q_m$.

В 1973 г. «группа искусственного интеллекта» Марсельского университета во главе с Аленом Колмероэ, опираясь на принцип резолюции, создала программу для доказательства теорем. В эту программу входил интерпретатор Ковальского. Она использовалась при построении систем обработки текстов на естественном языке. Программа была написана на Фортране и работала довольно медленно. Впоследствии этот продукт получил название ***Пролог*** **(*Prolog*)** — PROgrammation en LOGique. История «рождения» Пролога была обнародована впоследствии его создателями Аленом Колмероэ и Филиппом Русселем в статье «The birth of Prolog», размещенной в Интернете на сайте http://portal.acm.org/citation.cfm?id=1057820.

В 1976 г. Р. Ковальский и Маартен ван Эмден (Эдинбургский университет) предложили два подхода к созданию логических программ: процедурный и декларативный.

---

[1] Ж. Эрбран предложил идею сведения проверки доказуемости формул языка исчисления предикатов первого порядка к проверке истинности пропозициональных формул. Говорить об алгоритме АДТ в данном случае можно лишь с некоторой долей условности.