А. Л. Ездаков

ФУНКЦИОНАЛЬНОЕ И ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

ИЗДАТЕЛЬСТВО БИНОМ

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

РАЗВИТИЕ ПАРАДИГМ ПРОГРАММИРОВАНИЯ

Знакомое многим из курса философии слово «*парадигма*» имеет в информатике и программировании узкопрофессиональный смысл, сближающий информатику и программирование с лингвистикой. Парадигма программирования как исходная концептуальная схема постановки проблем и их решения является инструментом грамматического описания фактов, событий, явлений и процессов, возможно, не существующих одновременно, но интуитивно объединяемых в общее понятие.

Каждая парадигма программирования имеет свой круг приверженцев и свой класс успешно решаемых задач, предполагает разные приоритеты при оценке качества программирования, разные инструменты и методы работы и, соответственно, — стиль мышления и изобразительные стереотипы.

Ведущая парадигма прикладного программирования на основе императивного управления и процедурно-операторного стиля построения программ стала популярной более пятидесяти лет назад среди специалистов по организации вычислительных и информационных процессов. Последнее десятилетие резко расширило сферу применения средств информатики, распространив ее на массовое общение и досуг. Соответственно, изменились критерии оценки информационных систем и предпочтения в выборе средств и методов обработки информации.

Характерное для современных средств вычислительной техники доминирование одной архитектурной линии, стандартного интерфейса, типовой технологии программирования и т. д. может привести к снижению гибкости при обновлении информационных технологий. Учащиеся, привыкшие прочно усваивать все раз и навсегда, могут испытывать трудности при переходе на новые версии программного обес-

печения или новые модели аппаратных средств. При изучении языков программирования такие проблемы обычно обходят за счет одновременного изучения различных языков программирования или предварительного изложения некоей основы, задающей грамматическую структуру для обобщения понятий (изменяемость которых трудно улавливается на упрощенных учебных примерах). Именно такую основу дает нам изучение функционального программирования, поскольку оно нацелено на изложение и анализ парадигм, сложившихся в практике программирования в разных областях деятельности с различным уровнем квалификации специалистов. По этой причине оно может быть полезно и как концептуальная основа при изучении новых явлений в информатике.

Общие парадигмы программирования, сложившиеся в самом начале «эры компьютерного программирования», — и парадигмы прикладного, теоретического и функционального программирования в том числе, — имеют наиболее устойчивый характер.

Прикладное программирование имеет четкую проблемную направленность, отражающую процессы компьютеризации различных информационных и вычислительных методов, в том числе исследованных задолго до появления ЭВМ. В таких областях программирование мало чем отличается от кодирования; для него, как правило, достаточно операторного стиля представления действий. В практике прикладного программирования принято доверять хорошо проверенным шаблонам и библиотекам процедур, избегая рискованных экспериментов. В прикладном программировании особенно ценится точность и устойчивость научных расчетов. Язык Fortran стал признанным ветераном прикладного программирования, и лишь в последнее десятилетие он стал несколько уступать в этой области языкам Pascal и C, а на суперкомпьютерах — языкам параллельного программирования, таким как Sisal.

Теоретическое программирование нацелено на сопоставимость результатов научных экспериментов в области программирования и информатики. Здесь программирование

выступает как средство выражения формальных моделей, показа их значимости и фундаментальности. Эти модели унаследовали основные черты родственных математических понятий и позволили выработать алгоритмический подход в информатике. Стремление к доказательности построений и оценке их эффективности, правдоподобия, правильности, корректности и других формализуемых отношений на схемах и текстах программ послужило основой для создания структурированного программирования и других методик достижения надежности разработки программ. Стандартные версии Algol и Pascal, послужившие рабочим материалом для теории программирования, сменились здесь более удобными для экспериментирования аппликативными языками, такими как ML, Miranda, Scheme и другими диалектами Lisp, а в наши дни к ним присоединились версии языков C и Java.

Функциональное программирование сформировалось как дань математической направленности при исследовании и развитии искусственного интеллекта и освоении новых горизонтов в информатике. Абстрактный подход к представлению информации; лаконичный, универсальный стиль построения функций; ясность обстановки исполнения для разных категорий функций; свобода рекурсивных построений; доверие интуиции математика и исследователя; уклонение от преждевременного решения непринципиальных проблем распределения памяти; отказ от необоснованных ограничений на область действия определений — все эти качества соединены Джоном Маккарти в идее разработанного им языка Lisp. Продуманность и методическая обоснованность даже самых первых реализаций Lisp позволила быстро накопить опыт решения новых задач, подготовить их для прикладного и теоретического программирования. В настоящее время существуют уже сотни функциональных языков программирования, ориентированных на разные классы задач и разновидности применяемых технических средств.

Основные парадигмы функционального программирования складывались по мере возрастания сложности решаемых задач. Происходило расслоение средств и методов програм-

мирования в зависимости от глубины и общности проработки технических деталей организации компьютерной обработки информации. Выделялись разные стили программирования, наиболее зрелые из которых — *машинно-ориентированное*, *системное*, *логическое*, *трансформационное* и высокопроизводительное *параллельное программирование*.

Машинно-ориентированное программирование характеризуется аппаратным подходом к организации работы компьютера, нацеленным на максимальный учет всех возможностей оборудования. Здесь во главу угла ставится конфигурация оборудования, состояние памяти, команды передачи управления, очередность событий, исключения и прерывания, время реакции устройств и успешность реагирования на возникающие события. Долгое время предпочтительным средством программирования в этой области оставался Ассемблер, и лишь в последние годы он уступил позиции языкам Pascal и C, в том числе в области микропрограммирования. Впрочем, усовершенствование пользовательского интерфейса может вновь превратить Ассемблер в основной язык машинно-ориентированного программирования.

Системное программирование долгое время развивалось в рамках сервисных и заказных работ. Свойственный такой ситуации производственный подход опирался на предпочтение воспроизводимых процессов и стабильных программ, разрабатываемых для многократного использования. Для таких программ оправданы компиляционная схема обработки, статический анализ свойств, автоматизированная оптимизация и контроль. В этой области доминирует императивно-процедурный стиль программирования, являющийся непосредственным обобщением операторного стиля прикладного программирования. Он допускает некоторую стандартизацию и модульное программирование, но обрастает довольно сложными построениями, спецификациями, методами тестирования, средствами интеграции программ и т. п. Жесткость требований к эффективности и надежности здесь удовлетворяется разработкой профессионального инструментария, использующего сложные ассоциативно-семантические эвристики наряду с методами синтаксически-управляемого кон-

струирования и генерации программ. Бесспорный потенциал такого инструментария на практике ограничен трудоемкостью его освоения, что приводит к возникновению квалификационного ценза.

Высокопроизводительное программирование нацелено на достижение предельно возможных скоростных характеристик при решении особо важных задач. Естественный резерв роста производительности компьютеров — это *параллельные процессы*. Их организация требует детального учета временны́х отношений и неимперативного стиля управления действиями. Суперкомпьютеры, поддерживающие высокопроизводительные вычисления, потребовали особой техники системного программирования. В их числе — графово-сетевой подход к представлению систем и процессов для параллельных архитектур, который реализован в специализированных языках параллельного программирования и суперкомпиляторах, приспособленных для отображения абстрактной иерархии процессов уровня задач на конкретную пространственную структуру процессоров реального оборудования.

Логическое программирование возникло как упрощение функционального программирования для математиков и лингвистов, решающих задачи символьной обработки. Особенно привлекательна здесь возможность в качестве понятийной основы использовать недетерминизм, освобождающий от преждевременных упорядочений при программировании обработки формул. Продукционный стиль реализации процессов с возвратами достаточно естественен для лингвистического подхода к уточнению формализованных знаний экспертами, снижает стартовый барьер.

Трансформационное программирование методологически объединило технику оптимизации программ, макрогенерации и частичных вычислений. Центральное понятие в этой области — *эквивалентность информации*. Она проявляется в определении преобразований программ и процессов, в поиске критериев применимости преобразований, в выборе стратегии их использования. Смешанные вычисления, отложенные действия, «ленивое» программирование, задержанные процессы и т. п. используются как методы

повышения эффективности информационной обработки при некоторых дополнительно выявляемых условиях.

Экстенсивные подходы к программированию — это естественная реакция на радикальное улучшение эксплуатационных характеристик оборудования и компьютерных сетей, соответствующая превращению вычислительных средств из сложных технических инструментов в бытовые приборы (или их составную часть). При этом появилась возможность для обновления подходов к программированию, а также для реабилитации ряда старых идей, слабо развивавшихся из-за низкой технологичности и производительности ЭВМ. Здесь представляет интерес формирование *исследовательского*, *эволюционного*, *когнитивного* и *адаптационного* подходов к программированию, создающих перспективу рационального освоения реальных информационных ресурсов и компьютерного потенциала.

Исследовательский подход с учебно-игровым стилем профессионального, обучающего и любительского программирования может дать импульс росту изобретательности в совершенствовании технологий программирования, не справлявшихся с проблемами при использовании прежней аппаратной базы.

Эволюционный подход с мобильным стилем уточнения программ достаточно явно просматривается в концепции объектно-ориентированного программирования, постепенно перерастающего в *субъектно-ориентированное* и даже в *эго-ориентированное* программирование. Повторное использование определений и наследование свойств объектов могут удлинить жизненный цикл отлаживаемых информационных сред, повысить надежность их функционирования и простоту использования. Когнитивный подход с интероперабельным стилем визуально-интерфейсной разработки открытых систем и использование новых аудио-, видеосредств и нестандартных устройств открывают пути активизации восприятия сложной информации и упрощения ее адекватной обработки.

Адаптационный подход с эргономичным стилем индивидуализируемого конструирования персонифицированных информационных систем предоставляет информатикам воз-

можность грамотного программирования, организации и обеспечения технологических процессов в реальном времени, чувствительных к человеческому фактору.

Направление развития парадигмы программирования отражает изменение круга лиц, заинтересованных в развитии и применении информационных систем. Многие важные для практики программирования понятия, такие как события, исключения и ошибки, потенциал, иерархия и ортогональность построений, экстраполяция и точки роста программ, измерение качества и т. д., не достигли достаточного уровня абстрагирования и формализации. Все это позволяет прогнозировать развитие парадигм программирования и выбирать учебный материал в расчете на перспективу компонентного программирования (COM/DCOM, Corba, UML, .Net и др.). Если традиционные средства и методы выделения многократно используемых компонентов подчинялись критерию модульности, понимаемой как оптимальный выбор минимального сопряжения при максимальной функциональности, то современная элементная база допускает оперирование поликонтактными узлами, выполняющими простые операции.

В образовательном процессе парадигма программирования является инструментом формирования профессионального поведения. Программирование прошло долгий путь от профессиональной деятельности высококвалифицированной элиты технических специалистов и научных работников практически до одного из способов развлечения активной части цивилизованного общества. Освоение информационных систем через понимание необходимости компетентных решений и ответственного применения средств вычислительной техники во многом сменилось чисто интуитивными навыками хаотичных попыток решить поставленную задачу со скромной надеждой на везение и без претензий на реальные знания. Обслуживание центров коллективного пользования, профессиональная поддержка целостности информации и подготовки данных почти полностью отступили перед «самообслуживанием» при работе на персональных компьютерах, независимым функционированием сетей и разнород-

ных серверов с взаимодействием различных средств коммуникации. Противопоставление разрабатываемых программ, обрабатываемых данных и управления заданиями сегодня уступает представлению об интерфейсах, приспособленных для участия в информационных потоках подобно навигации в Web. Прежние критерии качества — скорость, экономия памяти и надежность обработки информации — все больше заслоняются внешней и игровой привлекательностью и широтой доступа к мировым информационным ресурсам. Замкнутые программные комплексы с известными гарантиями качества и надежности все больше вытесняются открытыми информационными комплектами с совершенно непредсказуемым развитием состава, способов хранения и обработки информации.

Все эти симптомы обновления парадигмы программирования определяют направление изменений, происходящих в системе базовых понятий, в концепции реализации информационных процессов и в информатике в целом. Тенденция использования интерпретаторов (а точнее — неполной компиляции) вместо компиляторов, анонсированная в концепции Java (в сравнении с C), и соблазн использования принципов объектно-ориентированного программирования на фоне общепринятого императивно-процедурного стиля можно рассматривать как неявное движение к функциональному стилю. Моделирующая сила функциональных формул достаточна для полноценного представления разных парадигм, что позволяет на их основе экстраполировать на будущее приобретение практических навыков организации информационных процессов.

ГЛАВА 1

ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

Реализация функционального языка программирования

Одним из языков программирования, реализующих парадигму функционального программирования, является Lisp (*LISt Processing* — «обработка списков»). Его основы разработал в 1956 г. профессор Массачусетского технологического института Джон Маккарти вместе со студенческой научной группой для более удобной реализации проекта «Принимающий советы» по искусственному интеллекту для анализа и разбора английских фраз. Сначала это были «рабочие» версии языка для первых компьютеров IBM и DEC, а более или менее завершенный вариант Lisp 1.5 появился в 1965 г.

Lisp, как и следует из его названия, предназначен для обработки *списков*, состоящих из *атомов* — абстрактных элементов, представляющих собой формально неограниченные по длине цепочки символов. В более привычном понимании они могут трактоваться как строки, как числа или могут представлять собой некие логические структуры с вложенными на неограниченную глубину подсписками в виде *иерархических деревьев*. Для обработки списков используется функциональная модель, базирующаяся на *теории Lambda-исчислений Чёрча*. Фактически программа на Lisp представляет собой набор *lambda-функций*; при этом работа со списками осуществляется с помощью базового набора примитивов типа CAR/CDR (взять первый элемент списка, который сам может быть списком / получить список без первого элемента). Таких примитивов в минимальном наборе — всего 13 штук. С их помощью (а главное — благодаря рекурсивной системе обработки информации) Lisp позволяет очень компактно описывать функции, для реализации которых на других языках программирования потребовались бы сотни и тысячи строк кода. Такие задачи как автоматическое дока-

зательство теорем, понимание естественного языка и окружающего мира, логические исчисления, написание компиляторов и другие, в которых требуется обработка абстрактной структурной информации, как оказалось, очень удачно описываются и программируются на Lisp.

После первых впечатляющих успехов актуальной стала задача эффективной реализации языка. Сначала все Lisp-системы были интерпретируемыми, что позволяло достичь определенной гибкости, но сильно замедляло процесс работы программы. Автор Lisp опубликовал в 1965 г. свою книгу «LISP 1.5 Programmer's Manual», в которой описал не только сам этот язык, но и структуру так называемой *виртуальной Lisp-машины* как некоей абстрактной схемы функционирования Lisp-системы, а также формальное определение структуры компилятора и интерпретатора. Этот труд стал образцом классического описания языка программирования и его окружения; на него ссылаются вплоть до нынешнего дня. Удивительная ясность и простота Lisp в сочетании с его мощностью и оригинальной идеологией сделала его не просто языком программирования, но своего рода способом формального описания алгоритмов. Кроме того, некоторые идеи, заложенные в Lisp, такие как «сборка мусора» или оптимизация памяти, освобождение ее от «висячих ссылок», актуальны и сегодня.

Где-то к началу 1970-х гг. относится первое неудачное решение в отношении Lisp, лишившее его основного преимущества перед другими языками — прозрачности структуры программы. Кто-то (может быть, это был сам Джон Маккарти) ввел в систему команд примитив `PROG`, позволяющий писать операторы последовательно один за другим, как в Fortran или Algol, а также добавил оператор `GO` (`goto`), без которого примитив `PROG`, очевидно, был лишен всякого смысла. С этого момента развитие Lisp «пошло под откос».

Lisp-машина была реализована в начале 1970-х гг. в виде, как сказали бы сейчас, «программы, зашитой в ПЗУ» (для повышения эффективности функционирования) в ряде компьютеров таких фирм как Xerox и Texas Instruments.

В конце 1970-х гг., с расцветом теории искусственного интеллекта и ростом актуальности средств для реализации

ее идей, Lisp претерпел свое второе рождение. Появилось множество его диалектов практически для всех платформ и операционных систем, и в том числе — два диалекта, которые стали основоположниками сегодняшних стандартов.

Первый из них — *Scheme Lisp*, который унаследовал от своего «родоначальника» наиболее чистые черты оригинальной идеологии. Пройдя глубокую математическую переработку, эта версия, по-прежнему ограничиваясь небольшим количеством базовых примитивов (полное описание языка занимает всего 50 страниц), позволила сосредоточиться на ключевых деталях при решении ряда математических задач, требующих некоего формального описательного аппарата. Например, оригинальной и многообещающей оказалась идея *engines* — параллельных процессов. Поэтому в большинстве научных групп используется именно эта версия Lisp.

Второй диалект — *Common Lisp* (*CL*) — наоборот, отличался очень большой библиотекой различных функций, чуть ли не превосходящей по их количеству аналогичные библиотеки Fortran (!). Его, конечно, было бы значительно удобнее использовать для реализации конкретных проектов, требующих, кроме простого анализа списочных структур, еще и значительных объемов вычислительной работы и организации хорошего графического интерфейса. Описание этого диалекта занимает уже около 1300 страниц, в него введено довольно много возможностей, характерных для обычных процедурных языков (типа C), — например строгая типизация, которая в оригинальном Lisp отсутствовала вообще. Версия CL сильно отличается от «классического» языка Lisp 1.5 времен 1960-х гг., и хотя она включает в себя все базовые возможности, в реальных проектах обычно используется более подходящая для человеческой психологии и более близкая к привычным языкам линейная структура программы, а не рекурсивная. Однако из-за отказа от оригинальной идеологии, когда требовалось очень четко формализовывать задачу в почти математических терминах, сразу возникли проблемы, характерные для обычных задач проектирования и реализации крупных проектов.

После активного распространения ОС UNIX в 1980-х гг. получила широкое распространение версия *Portable Stan-*

*dard Lis*p, реализованная на большинстве платформ, и наконец фактическим стандартом стал Common Lisp. А 8 декабря 1994 г. в Американском Институте Национальных Стандартов было зарегистрировано официальное описание этого языка ANSI X3.226:1994 (X3J13), которое действует и сегодня.

Всплеск интереса к объектно-ориентированному программированию не обошел и Lisp: в него были добавлены понятия объекта, метода, наследования. Быстро появился *объектный стандарт Common Lisp Object System* (*CLOS*), но, к сожалению, его создатели не понимали (или не хотели понять), что такое искусственное расширение языка, не соответствующее его идеологии, лишь усложняет Lisp и лишает его оригинальной ясности и эффективности.

Современные реализации Lisp представляют собой большие программные комплексы, близкие к CASE-системам. Сегодня этот язык относится скорее к 4GL-классу, несмотря на то, что он был придуман около 40 лет назад. Манипулирование объектами на абстрактном уровне хотя и требует подчас конкретного (не визуального) кодирования, но, тем не менее, делает ненужным программирование рутинных операций, а наличие больших библиотек, обеспечивающих быструю реализацию множества примитивов, позволяет получить более эффективный и надежный код, чем при ручном программировании аналогичных задач на C++. Например, совершенно абсурдным может показаться программирование на C задач автоматического интеллектуального перевода, тогда как для Lisp это — типичная сфера применения. (Кстати, интересно — на чем пишут свои «переводчики» наши программисты?) Имеется и большое число «компиляторов», переводящих текст задачи на Lisp в C-код.

В профессиональных Lisp-системах имеются специальные библиотеки для поддержки графического интерфейса. Не обошлось здесь и без объектно-ориентированных версий с описаниями таких классов как «окно», «кнопка», «меню», «скроллбар» и т. д. Присутствуют в них и символьные отладчики, профилировщики, а также прочий инструментарий. Конечно, сто́ят такие системы недешево. Например, цена многоплатформенной версии Allegro Lisp 4.2 составляет $4500, а Golden Common Lisp для DOS, Windows (в том

числе NT) и OS/2 обойдется в $2000. Впрочем, имеется не меньшее количество и некоммерческих компиляторов, например 32-разрядная версия Allegro Common Lisp 3.0 for Windows, GNU CL для UNIX и т. д.

В будущем можно ожидать появления версий Lisp и для Интернет. Простой интерпретатор Lisp на Java уже распространяется бесплатно, а вскорости наверняка появятся и более мощные сетевые диалекты. Благодаря своей простоте и эффективности этот красивый, но подзабытый в России язык заслуживает не меньшего внимания, чем те же C и C++.

Особенности языка Lisp

Основные отличия языка Lisp

От других языков программирования Lisp отличается следующими свойствами:

1) одинаковая форма данных и программ;
2) хранение данных, не зависящее от места их размещения в оперативной памяти компьютера;
3) автоматическое и динамическое управление памятью;
4) функциональная направленность;
5) нежесткое разграничение данных по типам;
6) интерпретирующий и компилирующий режимы работы;
7) пошаговое программирование;
8) единый системный и прикладной язык программирования.

Теперь рассмотрим эти свойства более подробно.

1. Одинаковая форма данных и программ.

В Lisp способы представления программы и обрабатываемых ею данных одинаковы. И то, и другое представляется как списочная структура, имеющая одинаковую форму. Поэтому Lisp-программы могут обрабатывать и преобразовывать другие Lisp-программы и даже самих себя. В процессе трансляции введенное и сформированное в результате вычислений выражение данных можно проинтерпретировать в качестве программы и непосредственно выполнить ее. Это свойство имеет не только теоретическое, но и большое практическое значение.