1978

Парадигмы программирования

Роберт Флойд

Станфордский университет

Премия Тьюринга 1978 г. Ассоциации по вычислительной технике (ACM) была вручена Роберту Флойду председателем Комитета по присуждению премий Уолтером Карлсоном 4 декабря на ежегодной конференции Ассоциации в Вашингтоне.

Осуществляя отбор претендентов, Подкомитет по присуждению премий за крупные научные достижения (бывший Подкомитет по присуждению премии Тьюринга) назвал имя профессора Флойда в связи с его «вкладом в создание следующих важных разделов информатики: теория синтаксического анализа, семантика языков программирования, автоматическая проверка правильности программ, автоматический синтез программ и анализ алгоритмов».

Профессор Флойд, получивший степени бакалавра гуманитарных наук и бакалавра естественных наук в Чикагском университете соответственно в 1953 и 1958 гг., в области информатики является самоучкой. Он начал заниматься этими проблемами в 1956 г., когда, работая ночным оператором на ЭВМ IBM 650, в перерывах между загрузкой карманов для перфокарт смог найти время, чтобы обучиться программированию.

Флойд создал один из первых компиляторов Алгола-60, закончив свою работу над этим проектом в 1962 г. В рамках проекта он выполнил ряд новаторских работ по оптимизации компилятора. Потом вплоть до 1965 г. он занимался систематизацией синтаксического анализа языков программирования. Для этого им были разработаны метод предшествования, метод ограниченного контекста и синтаксический анализ с помощью языка продукций.

В 1966 г. профессор Флойд предложил математический метод доказательства правильности программ. За годы работы он создал ряд полезных быстрых алгоритмов. Среди них — древовидный алгоритм разрядовой сортировки, алгоритмы поиска кратчайших путей в сетях, а также алгоритм нахождения медиан и выпуклых оболочек. Он также определил предельную скорость цифрового суммирования и предельные скорости размещения информации в памяти ЭВМ.

Ему же принадлежит существенный вклад в методы машинного доказательства теорем и создание блоков орфографического контроля.

В последние годы профессор Флойд занимается разработкой и реализацией языка программирования, в первую очередь предназначенного для использования студентами. Язык можно будет использовать при систематическом обучении новичков структурному программированию, и по своим возможностям он будет почти универсален.

* * *

Сегодня я хочу поговорить о парадигмах программирования, об их влиянии на наши успехи как создателей компьютерных программ, о том, как нужно обучать парадигмам и как их следует включать в наши языки программирования.

Известным примером парадигмы программирования служит методика *структурного программирования*, которая, видимо, является доминирующей в большинстве современных подходов. Структурное программирование, как оно определяется Дейкстрой [6], Виртом [27, 29], Парнасом [21] и другими, содержит две фазы.

В первой фазе, представляющей собой нисходящее проектирование или пошаговую детализацию, задача разбивается на весьма незначительное число более простых подзадач. Например, при написании программы решения системы линейных уравнений первым уровнем разбиения станет этап приведения системы к треугольной форме с последующим этапом, заключающимся в обратной подстановке в систему, приведенную к треугольной форме. Это последовательное разбиение продолжается до тех пор, пока возникающие подзадачи не станут достаточно простыми, чтобы их можно было решать непосредственно. В примере с системой линейных уравнений операция обратной подстановки должна подвергаться дальнейшему разбиению в виде обратных итераций некоторого процесса поиска и запоминания значения i-й переменной из i-го уравнения. Дальнейшее разбиение должно привести к полностью детализированному алгоритму.

Вторая фаза парадигмы структурного программирования предусматривает движение снизу вверх, от конкретных объектов и функций данной машины к более абстрактным объектам и функциям, используемым в модулях, созданных путем нисходящего проектирования. Если в примере с линейным уравнением коэффициенты уравнений являются рациональными функциями одной переменной, то сначала можно построить арифметическое представление с многократной точностью и соответствующие процедуры, а затем на этой основе — полиномиаль-

ное представление с собственными арифметическими операциями и т. д. Этот подход получил название метода *уровней абстракции* или *сокрытия информации.*

Парадигму структурного программирования никак нельзя считать повсеместно признанной. Наиболее ярые ее сторонники должны признать, что самой по себе ее недостаточно, чтобы превратить все сложные проблемы в простые. Сохраняют свое значение и другие более специализированные парадигмы высокого уровня, такие, как метод ветвей и границ [17, 20] или метод «разделяй и властвуй» [1, 11]. Тем не менее парадигма структурного программирования действительно помогает расширить возможности программиста, позволяя создавать программы, которые слишком сложны, чтобы их можно было писать эффективно и надежно без методологической базы.

Я считаю, что современное состояние дел в программировании отражает неадекватность наших запасов парадигм, нашего знания существующих парадигм, наших методов обучения парадигмам программирования и способа, которым наши языки программирования поддерживают или не поддерживают парадигмы сообществ их пользователей.

Нынешнее состояние искусства программирования было недавно охарактеризовано Робертом Бэлзером [3] следующими словами: «Хорошо известно, что программное обеспечение находится в жалком виде. Оно ненадежно, появляется с опозданием, не реагирует на происшедшие перемены, неэффективно и дорого. Более того, поскольку в настоящее время его создание трудоемко, то ситуация будет и дальше ухудшаться по мере возрастания спроса и увеличения расходов на рабочую силу». Если это напоминает знаменитый «кризис программного обеспечения» примерно десятилетней давности, то тот факт, что на протяжении 10 или 15 лет мы пребываем в одном и том же состоянии, подсказывает, что термин «упадок программного обеспечения» был бы более подходящим.

Томас Кун в книге «Структура научных революций» [16] описывает научные революции за последние несколько столетий как возникавшие из изменений в доминировавших парадигмах. Некоторые из наблюдений Куна выглядят применимыми к нашей области. Относительно учебников, излагающих современные научные знания студентам, Кун пишет:

> «Эти тексты, например, часто наводят на мысль, что содержание науки можно единственным образом проиллюстрировать описываемыми на их страницах наблюдениями, законами и теориями».

Аналогично большинство учебников программирования исходит из допущения, что содержание программирования заключается только в знании алгоритмов и определений языков, описываемых на их страницах.

Кун также отмечает:

«Именно изучение парадигм, в том числе парадигм гораздо более специализированных, чем названные мною здесь в целях иллюстрации, служит основным фактором, подготавливающим студента к членству в том или ином научном сообществе. Поскольку в дальнейшем он оказывается в окружении людей, которые изучали основы предмета на тех же самых конкретных моделях, что последующая практика в научных исследованиях не часто будет побуждать его к резкому расхождению с фундаментальными принципами...»

В информатике можно видеть несколько таких сообществ, каждое из которых говорит на собственном языке и использует собственные парадигмы. Фактически каждый из языков программирования обычно поощряет использование одних парадигм и затрудняет использование других. Существуют четко определенные школы программирования на языке Лисп, APL, Алгол и т. д. В качестве исходной структурной информации о программе одни рассматривают поток данных, а другие — поток управления. Рекурсия и итерация, копирование и совместное использование структур данных, вызов по имени и вызов по значению — все они имеют своих сторонников.

И еще одна цитата из Куна:

«...Старые школы постепенно исчезают. Исчезновение этих школ частично обусловлено обращением их членов к новой парадигме. Но всегда остаются ученые, верные той или иной устаревшей точке зрения. Они просто выпадают из дальнейших совокупных действий представителей их профессии, которые с этого времени игнорируют все их усилия».

В информатике не существует механизма, который исключал бы таких людей из сферы их профессиональной деятельности. Я подозреваю, что они преимущественно становятся менеджерами в сфере разработки программного обеспечения.

Бэлзер в своем плаче о состоянии разработки программного обеспечения пророчествовал, что автоматическое программирование спасет нас. Я желаю успеха сторонникам автоматического программирования, но, пока они не вычистят «конюшен», наша главная надежда связана с расширением собственных возможностей. Наилучшим шансом, которым мы располагаем для улучшения общей практики программирования, я считаю внимательное рассмотрение наших парадигм.

В начале 60-х годов синтаксический анализ бесконтекстных языков представлял собой особенно острую проблему как для разработки компиляторов, так и для лингвистики. Опубликованные алгоритмы были и медленными, и некорректными. Джон Кок (как утверждают, приложив весьма незначительные усилия) нашел быстрый и простой алгоритм [2], основывающийся на теперь уже стандартной парадигме, которая представляет собой вычислительную форму динамического программирования [1]. Парадигма динамического программирования

решает некую задачу для данной входной информации, решая ее вначале итерационно для всех отрезков вводимой информации меньшей длины. Алгоритм Кока успешно осуществлял любые синтаксические анализы всех подцепочек ввода. В этих концептуальных рамках данная проблема становилась почти тривиальной. Полученная процедура стала первым алгоритмом, стабильно завершающим работу при любом входе за полиномиальное время.

Приблизительно тогда же, после публикации нескольких некорректных нисходящих синтаксических анализаторов, этой проблемой занялся и я. Для этого я создал парадигму нахождения иерархической организации процессоров, напоминающей организацию людей-работодателей, нанимающих и увольняющих сотрудников для решения конкретных задач, и затем смоделировал ее работу [8]. Моделирование таких множественных рекурсивных процессов привело меня к использованию в качестве управляющей структуры рекурсивных сопрограмм. Позже я выяснил, что другие программисты, столкнувшись со сложными комбинаторными проблемами, например Гелернтер со своей машиной для доказательства геометрических теорем [10], очевидно, изобрели такую же управляющую схему.

Опыт Джона Кока и мой собственный наглядно показывают вероятность того, что для обеспечения прогресса в области программирования необходимо будет продолжать изобретать, совершенствовать и публиковать новые парадигмы.

Примером эффективной тщательно разработанной парадигмы является работа Шортлиффа и Дэвиса над программой MYCIN [24], которая блестяще диагностирует и дает рекомендации по лечению бактериальных инфекций. MYCIN представляет собой систему на основе правил продукции, в которую заложен большой набор независимых правил с проверяемыми условиями применимости каждого правила и простым действием, осуществляемым каждым правилом в случае выполнения соответствующего условия. Программа TERESIAS [5] Дэвиса модифицирует программу MYCIN, позволяя пользователю-эксперту улучшать ее рабочие характеристики. Программа TERESIAS уточняет парадигму, прослеживая ответные действия в обратной последовательности — от ошибочного результата через правила и условия, разрешившие его применение, до того момента, когда будет обнаружено неудовлетворительное правило, дающее неверные результаты из правильных предположений. За счет этого стало технически возможным специалисту-медику, не знакомому с программированием, улучшать диагностические способности программы MYCIN. Хотя в программе MYCIN нет ничего, что не могло бы быть закодировано в виде традиционного ветвящегося дерева решений с использованием условных

переходов, именно использование парадигмы, основанной на системе правил, с последующим ее усовершенствованием добавлением возможности самоизменения сделало возможным интерактивное улучшение программы.

Если прогресс искусства программирования в целом требует постоянного изобретения и усовершенствования парадигм, то совершенствование искусства отдельного программиста требует, чтобы он расширял свой *репертуар* парадигм. В рамках собственного опыта построения сложных алгоритмов я выработал определенную технику, оказавшуюся наиболее плодотворной для расширения моих возможностей. После того как поставленная задача решена, я повторно решаю ее с самого начала, прослеживая и повторяя только суть предыдущего решения. Я проделываю эту процедуру до тех пор, пока решение не становится настолько четким и ясным, насколько это для меня возможно. Затем я ищу общее правило решения аналогичных задач, которое *заставило бы* меня подходить к решению поставленной задачи наиболее эффективным способом с первого раза. Часто такое правило приобретает непреходящее значение. Поиски такого общего правила привели меня от ранее упоминавшегося алгоритма синтаксического анализа, основывающегося на рекурсивных программах, к общему методу написания недетерминированных программ [9], которые затем с помощью макрорасширения преобразуются в обычные детерминированные программы. Эта парадигма позднее, оказавшись включенной в такие языки программирования, как PLANNER [12, 13], MICROPLANNER [25] и QA4 [23], нашла применение в казалось бы далекой области — автоматическом решении задач, относящемся к вопросам искусственного интеллекта.

Приобщению отдельного программиста к новым парадигмам может способствовать чтение программ, написанных другими. Но такой подход не очень продуктивен, поскольку материал для чтения заимствуется у программистов, работающих, как правило, в той же фирме и отобранных по их умению использовать те же языки и парадигмы. Подтверждением этому служит то, сколь часто наша промышленность адресует предложения о найме не программистам вообще, а программистам, пишущим на Фортране или Коболе. Правила Фортрана можно выучить за несколько часов, связанные с ними парадигмы требуют значительно большего времени, как для того, чтобы их усвоить, так и для того, чтобы от них отвыкнуть.

Помочь может знакомство с программированием, ведущимся по незнакомым правилам. Работая в этом году во время творческого отпуска в МТИ (MIT), я столкнулся с многочисленными примерами широких возможностей, которые извлекают программисты на Лиспе из единственной структуры дан-

ных, используемой также в качестве универсальной синтаксической структуры для всех встречающихся в программах функций и операций и позволяющей им манипулировать программами как данными. Хотя ранее я был энтузиастом синтаксически богатых языков вроде Алгола, сейчас я ясно и наглядно вижу глубокий смысл тьюринговской лекции М. Минского 1969 г. [19], в которой он утверждал, что однородность структуры Лиспа и рекурсия предоставляют программистам такие возможности, содержание которых вполне компенсирует утраты наглядности визуальной формы программ. Я хотел бы получить какой-то разумный синтез этих подходов.

То, что каждый стремится создать новый язык программирования, остается справедливым для настоящего момента в той же степени, что и для 1956 г., когда я начал заниматься информатикой. «Я предпочитаю писать программы, которые помогают писать программы, а не писать их самому» — выбито на стене одного из зданий Станфордского университета. Оценивая ежегодный «урожай» новых языков программирования, полезно классифицировать их по тому, в какой степени они допускают и поощряют использование эффективных парадигм программирования. Когда мы делаем наши парадигмы четкими, то выясняется, что их очень много. Как показал Корделл Грин, механическая генерация алгоритмов поиска и сортировки, таких, как сортировка слиянием и быстрая сортировка, требует свыше ста правил, большинство из которых, вероятно, это парадигмы, знакомые большей части программистов. Часто наши языки программирования не только не помогают, а и мешают нам в использовании даже парадигм низкого уровня. Приведу несколько примеров.

Допустим, мы моделируем динамику популяций системы хищник — жертва, например волков и кроликов. Мы имеем два уравнения

$$W' = f(W, R),$$
$$R' = g(W, R),$$

которые определяют число волков и кроликов к концу какого-то промежутка времени как функцию их количества в его начале.

Общая ошибка начинающих состоит в том, что они пишут:

```
FOR I := - - - DO
    BEGIN
    W := f(W, R);
    R := g(W,R)
    END
```

где g ошибочно вычисляется с использованием измененного значения W. Для того чтобы программа заработала, мы должны написать

```
FOR I := ---DO
    BEGIN
    REAL TEMP;
    TEMP := f(W,R);
    R := g(W,R);
    W := TEMP
    END
```

Новичок прав, полагая, что нет необходимости делать это. Одна из наиболее распространенных наших парадигм, как и в случае моделирования системы хищник — жертва, состоит в одновременном присваивании новых значений всем компонентам векторов состояний. Тем не менее едва ли в каком-либо языке существует оператор для одновременного присваивания. Вместо этого нам приходится выполнять чисто техническую, связанную с потерей времени и возможными ошибками операцию заведения одной или нескольких временных переменных и присваивать новые значения через них.

Еще раз обратимся к простой на первый взгляд задаче:

Вводить строки текста до тех пор, пока не обнаружится строка, состоящая только из пробелов. Устранить лишние пробелы между словами. Напечатать текст по тридцать литер в строке, не разрывая слова между строками.

Поскольку ввод и вывод удобно записываются с помощью множественных уровней итерации и поскольку итерации на входе не вкладываются в итерации на выходе и не включают их, то данная задача для программирования на большинстве языков оказывается на удивление сложной [14]. У новичков решение этой задачи займет в 3—4 раза больше времени, чем рассчитывают преподаватели, и завершится либо полным хаосом, либо кустарной структурой управляющей логики программы, в которой используются явные приращения и условное выполнение для моделирования нужных итераций.

Задача естественным образом формулируется с помощью разбиения на три связанные сопрограммы [4] для ввода, преобразования и вывода потока литер. Однако, за исключением языков моделирования, немногие из наших языков программирования содержат структуры управления сопрограммами, позволяющие запрограммировать данную задачу естественным образом.

Когда язык удобен для реализации парадигмы, я буду говорить, что он *поддерживает* данную парадигму. А в случае если на этом языке парадигма реализуема, но сделать это непросто,

я буду говорить, что он слабо поддерживает парадигму. Как показывают два предыдущих примера, большинство наших языков лишь слабо поддерживают одновременное присваивание и абсолютно не поддерживают сопрограмм, хотя требуемые механизмы значительно проще и полезнее, чем, скажем, процедуры рекурсивного вызова по имени, реализованные семнадцать лет назад в семействе языков Алгол.

Даже парадигма структурного программирования в лучшем случае слабо поддерживается многими из наших языков программирования. Чтобы записать программу решения системы уравнений, нужно иметь возможность написать:

```
MAIN___PROGRAM:
  BEGIN
  TRIANGULARIZE;
  BACK___SUBSTITUTE
  END;
BACK___SUBSTITUTE:
  FOR I := N STEP -1 UNTIL 1 DO
    SOLVE___FOR___VARIABLE(I);
SOLVE___FOR___VARIABLE(I):
  - - -
  - - -
TRIANGULARIZE:
  - - -
  - - -
```

Процедуры для арифметических операций с многократной точностью

Процедуры для арифметических операций с рациональными функциями

Описания массивов

В большинстве современных языков записать в указанной последовательности главную программу, процедуры и описания данных нельзя. Требуется некоторая предварительная перекомпоновка текста, выполняемая вручную, которая, вообще говоря, вполне поддается автоматизации. Далее, любые переменные, используемые более чем в одной из процедур с многократной точностью, должны быть глобальными для каждой части программы, где могут использоваться арифметические операции с многократной точностью, что позволяет вопреки принципу сокрытия информации осуществлять случайную модификацию. И наконец, детальное разбиение задачи на иерархию процедур обычно приводит к очень неэффективному коду, хотя большинство процедур, поскольку они вызываются только в одном месте главной программы, могут быть реализованы эффективно с помощью макрорасширения.

Парадигма какого-то еще более высокого уровня абстракции, чем парадигма структурного программирования, пред-

ставляет собой иерархию языков, где программы на языке наивысшего уровня оперируют с наиболее абстрактными объектами и транслируются в программы на языке следующего, более низкого уровня. Примерами тому служат многочисленные предназначенные для работы с формулами языки, которые построены на базе Лиспа, Фортрана и других языков. Большинство из наших языков более низкого уровня не могут полностью поддерживать указанные суперструктуры. Например, их системы диагностирования ошибок обычно «отлиты из бетона», так что диагностические сообщения делаются понятными только посредством ссылок транслированный текст программы на языке более низкого уровня.

Я считаю, что дальнейшее развитие программирования как искусства требует разработки и распространения языков, поддерживающих основные парадигмы сообществ их пользователей. Созданию языка должно предшествовать перечисление этих парадигм, включая изучение слабостей программного обеспечения, связанных с неадекватной поддержкой некоторых парадигм. Я не удовлетворен такими расширениями наших языков, как вариантные записи и множества подмножеств Паскаля [15, 28], пока парадигмы, о которых я говорил, и многие другие не поддерживаются или слабо поддерживаются. Если когда-нибудь появится наука о проектировании языков программирования, то, вероятно, она в основном будет состоять из сопоставления языков с методами конструирования, которые в них поддерживаются.

Я не хочу утверждать, что поддержка парадигмы ограничивается только возможностями наших языков программирования. Вся среда, в которой мы пишем программы — системы диагностики, системы файлов, редакторы и т. д., — может быть рассмотрена как поддерживающая или не поддерживающая набор методов создания программ. Есть надежда, что это начинает осознаваться. Например, в последней работе в IRIA во Франции и некоторых других используются редакторы, учитывающие структуру программы, которую они редактируют [7, 18, 26]. Оценить это может всякий, кто пытался решить даже столь простую задачу, как изменение всех X, появляющихся в задаче в качестве идентификатора, без случайного изменения всех других X.

Теперь я хотел бы поговорить о том, что мы *преподаем* под видом компьютерного программирования. Часть нашей злосчастной одержимости формой в ущерб содержанию, о чем М. Минский сожалел в своей тьюринговской лекции [19], проявляется обычно в выборе того, чему учить. Если спрашиваю какого-либо профессора о том, что он читает в вводном курсе программирования, то ответит ли он с гордостью «Паскаль»

или застенчиво «Фортран», я все равно знаю, что он обучает грамматике, набору семантических правил и некоторым законченным алгоритмам, предоставляя студентам самостоятельно открывать некоторые процессы написания программ. Даже учебники, основанные на парадигме структурного программирования, хотя и содержат «режиссуру» на высшем уровне, который мы можем назвать уровнем «сюжета» в написании программы, часто не оказывают никакой помощи на промежуточных уровнях, которые мы можем назвать уровнем «эпизода».

Я считаю, что можно ясно изложить набор систематических методов для всех уровней построения программы и что студенты, курс обучения которых был построен таким образом, имеют значительное преимущество перед теми, кто обучался стандартным образом, только лишь изучая уже готовые программы.

Ниже дается несколько примеров того, чему мы можем учить.

Когда я знакомлю студентов с возможностями ввода, свойственными какому-либо языку программирования, то использую стандартную парадигму интерактивного ввода в форме макрокоманды, названной мною PROMT_READ_CHECK_ECHO, которая осуществляет считывание до тех пор, пока вводимые данные удовлетворяют проверке на правильность, а затем осуществляет их эхопередачу в выходной файл. Эта макрокоманда является сама по себе на некотором уровне парадигмой итерации и ввода. В то же время, поскольку она чаще считывает один раз, чем выдает диагноз «неверные данные», она вводит более общую, изученную ранее парадигму цикла, выполняемого «$n+1/2$ раз».

```
PROMPT_READ_CHECK_ECHO: аргументы являются цепочкой
PROMPT, переменная V, которая должна быть считана, и усло-
        вие BAD, характеризующее неверные данные;
PRINT_ON_TERMINAL (PROMPT);
READ_FROM_TERMINAL (V);
WHILE BAD (V) DO
  BEGIN
  PRINT_ON_TERMINAL («НЕВЕРНЫЕ ДАННЫЕ»),
  READ_FROM_TERMINAL (V)
  END;
  PRINT_ON_FILE (V)
```

Она также, на более высоком уровне, иллюстрирует ответственность программиста по отношению к пользователю данной программы, включая следующий принцип: каждый компонент программы должен быть защищен от ввода, для которого он не предназначался.

Говард Шрауб и другие члены Группы практического обучения программированию [22] в Массачусетском технологическом институте успешно обучают своих студентов-новичков парадигме, полезной во многих случаях, которую они называют «создавать/фильтровать/накапливать». Студенты учатся распознавать многие внешне несхожие задачи как состоящие из перечисления элементов некого множества, отфильтровывания подмножества и накапливания некой функции от элементов этого подмножества.

Язык MACLISP [18], используемый студентами, поддерживает эту парадигму, студенты создают только генератор, фильтр и накапливающий сумматор.

Ранее упоминавшаяся мною модель популяционной динамики хищник—жертва также представляет собой пример общей парадигмы, — парадигмы конечных автоматов. Обычно в парадигме конечных автоматов состояние вычисления представляется с помощью множества хранимых переменных. Если состояние сложное, то функция переходов требует создания механизма, способного поддерживать парадигму одновременного присваивания, особенно в силу того, что большинство языков лишь слабо поддерживают одновременное присваивание. Чтобы проиллюстрировать это, предположим, что мы хотим вычислить

$$\frac{\pi}{6} = \arcsin\left(\frac{1}{2}\right) = \boxed{\frac{1}{\boxed{2}\cdot\boxed{1}}} + \boxed{\frac{1}{\boxed{2^3\cdot 2}\cdot\boxed{3}}}$$

$$+ \boxed{\frac{1\cdot 3}{\boxed{2^5\cdot 2\cdot 4}\cdot\boxed{5}}} + \boxed{\frac{1\cdot 3\cdot 5}{\boxed{2^7\cdot 2\cdot 4\cdot 6}\cdot\boxed{7}}} + \cdots$$

где в каждом из слагаемых заключены в рамочку части, которые окажутся полезными при вычислении следующего члена, стоящего правее. Без описания всей парадигмы построения для таких процессов часть построения смены состояния заключается в постоянном нахождении способа перейти от

$$Q = \frac{1\cdot 3}{2^5\cdot 2\cdot 4}, \quad C = 5$$

$$S = \frac{1}{2} + \frac{1}{2^3\cdot 2\cdot 3} + \frac{1\cdot 3}{2^5\cdot 2\cdot 4\cdot 5}$$

к

$$Q' = \frac{1 \cdot 3 \cdot 5}{2^7 \cdot 2 \cdot 4 \cdot 6}, \qquad C' = ?$$

$$S' = \frac{1}{2} + \cdots + \frac{1 \cdot 3 \cdot 5}{2^7 \cdot 2 \cdot 4 \cdot 6 \cdot 7}.$$

Опытный программист усвоил этот шаг, и во всех, за исключением самых сложных, случаях автоматически прибегает к нему. Новичку явная демонстрация парадигмы позволяет решать более сложные задачи теории конечных автоматов, чем он мог бы без этой помощи, и, что еще более важно, поощряет его самостоятельно находить другие полезные парадигмы.

Большинство классических алгоритмов, помещаемых в учебниках по программированию, могут рассматриваться в качестве примеров более широких парадигм. Формула Симпсона является примером экстраполяции к пределу. Метод исключения Гаусса представляет собой решение задачи методом рекурсивного спуска, преобразованным в итеративную форму. Сортировка слиянием является примером парадигмы «разделяй и властвуй». В отношении каждого такого классического алгоритма можно задать вопрос: «Каким образом я мог бы придумать его?» и восстановить, какой должна быть эквивалентная классическая парадигма.

В итоге мое обращение к серьезным программистам состоит в следующем: отводите часть вашего рабочего дня анализу и уточнению своих собственных методов. Даже несмотря на то, что программистам всегда приходится укладываться в сроки, которые уже либо прошли, либо приближаются, методологическая абстракция является разумным долгосрочным капиталовложением.

Преподавателю программирования я могу сказать даже больше этого: определите используемые вами парадигмы настолько полно, насколько вы можете, и затем ясно излагайте их. Они пригодятся вашим студентам, когда Фортран станет образцом мертвого языка-прототипа вроде латинского или санскрита.

Разработчику языков программирования я скажу: пока вы не можете поддерживать парадигмы, которые я использую при написании программ, или по меньшей мере поддерживать *мое* расширение вашего языка до такого, который действительно поддерживает мои методы программирования, я не нуждаюсь в ваших блестящих новых языках; как старый автомобиль или дом, старый язык обладает ограничениями, с которыми я научился уживаться. Чтобы убедить меня в достоинствах вашего языка, вы обязаны продемонстрировать мне, как на нем

писать программы. Я не хочу отбивать охоту к созданию новых языков, я хочу подтолкнуть разработчика языков к серьезному изучению деталей процесса разработки.

Я благодарю членов ACM за то, что они включили меня в число знаменитых людей — моих предшественников по чтению тьюринговских лекций. Никто не может достичь такого положения без помощи других. Я должен выразить свою признательность многим, но особенно это относится к четверым: Бену Миттмэну, который на начальном этапе моей карьеры помогал мне и поощрял научную и преподавательскую стороны моего интереса к информатике; Гербу Саймону — человеку эпохи Ренессанса в нашей профессии, беседы с которым представляли собой подлинную школу; Джорджу Форсайту, снабдившему меня парадигмой обучения информатике, и моему коллеге Дональду Кнуту, создавшему общепризнанный образец интеллектуальной целостности. Мне также сопутствовала удача и в том отношении, что у меня было много прекрасных аспирантов, от которых, как мне кажется, я почерпнул столько же, сколько и дал им.

Я благодарен и глубоко признателен всем вам.

ЛИТЕРАТУРА

1. Aho A. V., Hopcroft J. E., and Ullman J. D. The Design and Analysis of Computer Algorithms. Addison-Wesley, Reading, Mass., 1974. [Имеется перевод: Ахо А., Хопкрофт Дж., Ульман Дж., Построение и анализ вычислительных алгоритмов. — М.: Мир, 1979.]
2. Aho A. V. and Ullman J. D. The Theory of Parsing, Translation, and Compiling. Vol. 1: Parsing Prentice-Hall, Englewood Cliffs, New Jersey, 1972. [Имеется перевод: Ахо А., Ульман Дж., Теория синтаксического анализа и трансляции. Т. 1. — М.: Мир, 1978.]
3. Balzer R. Imprecise Program Specification. Report ISI/RR—75—36, Inform. Sciences Inst., Dec. 1975.
4. Conway M. E. Desing of a separable transition-diagram compiler Comm. ACM 6, 7 (July 1963), 396—408.
5. Davis R. Interactive transfer of expertise: Acquisition of new inference rules. Proc. Int. Joint. Conf. of Artif. Intell. MIT, Cambridge, Mass., August 1977, pp. 321—328.
6. Dijstra E. W. Notes on structured programming. In Structured Programming, O. J. Dahl, T. W. Dijkstra and C. A. R. Hoare, Academic Press, New York, 1972, pp. 1—82. [Имеется перевод: Дал У., Дейстра Э., Хоор К. Структурное программирование. — М.: Мир, 1975.]
7. Donzeau-Gouge V., Huet G., Kahn G., Lang B., and Levy J. J. A structure oriented program editor: A first step towards computer assisted programming. Res. Rep. 114, IRIA, Paris, April 1975.
8. Floyd R. W. The syntax of programming languages—A survey. IEEE ES—13. 4 (Aug. 1964), 346—353.
9. Floyd R. W. Nondeterministic algorithms. J. ACM 14, 4 (Oct. 1967), 636—644.
0. Gelernter. Realisation of a geometry-theorem proving machine. In Computers and Thought, E. Feigenbaum and J. Feldman, Eds., McGraw-Hill, New York, 1963, pp. 134—152.

11. Green C. C. and Barstow D. On program synthesis knowledge. Artif. Intel. 10, 3 (June 1978), 241—279.
12. Hewitt C. PLANNER: A language for proving theorems in robots. Proc. Int. Joint Conf. on Artif. Intel., Washington D. C. 1969.
13. Hewitt C. Description and theoretical analysis (using schemata) of PLANNER... AI TR-258, MIT, Cambridge, Mass., April 1972.
14. Hoare C. A. R. Communicating sequential processes. Comm. ACM 21, 8 (Aug. 1978), 666—677.
15. Jensen K. and Wirth N. Pascal User Manual and Report. Springer-Verlag. New York, 1978. [Имеется перевод: Йенсен К., Вирт Н., Паскаль. Руководство для пользователя и описание языка. — М.: Финансы и статистика, 1982.]
16. Kuhn T. S. The Structure of Scientific Revolutions. Univ. of Chicago Press, Chicago, Ill., 1970. [Имеется перевод: Кун Т. Структура научных революций. — М.: Прогресс, 1977.]
17. Lawler T., and Wood D. Branch and bound methods: A survey. Operations Res. 14, 4 (July—Aug. 1966), 699—719.
18. MACLISP Manual. MIT, Cambridge, Mass., July 1978.
19. Minsky M. Form and content in computer science. Comm. ACM 17, 2 (April 1970), 197—215.
20. Nilsson N. J. Problem Solving Methods in Artificial Intelligence. McGraw-Hill, New York, 1971. [Имеется перевод: Нильсон Н. Искусственный интеллект. Методы поиска решений. — М.: Мир, 1973.]
21. Parnas D. On the criteria for decomposing systems into modules. Comm. ACM 15, 12 (Dec. 1972), 1053—1058.
22. Rich C. and Shrobe H. Initial Report on a LISP programmer's apprentice. IEEE J. Software Eng. SE-4, 6 (Nov. 1978), 456—467.
23. Rulifson J. F., Derkson J. A. and Waldinger R. J. QA4: A procedural calculus for intuitive reasoning. Tech. Note 73, Stanford Res. Inst. Menlo Park, Calif., Nov. 1972.
24. Shortliffe E. H. Computer-Based Medical Consultations: MYCIN. American Elsevier, New York, 1976.
25. Sussman G. J., Winograd T. and Charniak C. MICROPLANNER reference manual. AI Memo 203A, MIT, Cambridge, Mass., 1972.
26. Teitelman W., et al. INTERLISP manual. Xerox Palo Alto Res. Ctr., 1974.
27. Wirth N. Program development by stepwise refinement. Comm. ACM 14 (April 1971), 221—227.
28. Wirth N. The programming language Pascal. Acta Informatica 1, 1 (1971), 35—63.
29. Wirth N. Systematic Programming, an Introduction. Prentice-Hall, Englewood Cliffs, New Jersey, 1973. [Имеется перевод: Вирт Н., Систематическое программирование. Введение. — М.: Мир, 1977.]