1966

Синтез алгоритмических систем

А. Дж. Перлис

Институт технологии Карнеги
Питсбург, Пенсильвания

ВВЕДЕНИЕ

Знание и мудрость обогащают человека. Знание ведет к компьютерам, а мудрость — к китайским палочкам для еды. К сожалению, наша ассоциация слишком поглощена знанием. Мудрости придется подождать до более благоприятных дней.

На чем основывается и будет основываться слава Тьюринга? На том, что он доказал теорему о существовании для универсального вычислительного устройства — позднее названного машиной Тьюринга — функций, вычисление которых непосильно этому устройству? Я сомневаюсь в этом. Более вероятно, что она основывается на модели, которую он придумал и разрабатывал: на его формальном механизме.

Эта модель покорила воображение и завладела мыслями целого поколения ученых. Она обеспечила основы для споров, породивших теории. Его модель оказалась столь полезной, что пробудила активность не только математиков, но и представителей ряда инженерных дисциплин. Привлеченные ею доводы не всегда являются формальными, а последующие построения не всегда абстрактные. В сущности, самым плодотворным следствием появления машин Тьюринга стало построение, изучение и вычисление так называемых вычислимых функций, т. е. программирование для компьютеров. Это и неудивительно, потому что компьютеры способны вычислять намного больше того, что мы способны специфицировать.

Я уверен, что все согласятся с тем, что эта модель имела исключительное значение. Историки простят меня за то, что я совсем не уделю в этой лекции внимания тому влиянию, которое Тьюринг оказал на разработку универсального цифрового компьютера, еще более ускорившего наше вовлечение в теорию и практику вычислений.

Хотя, разумеется, не только появление машины Тьюринга способствовало нашей вычислительной деятельности и поощряло ее. Я все же думаю, что столь же мощное воздействие оказал только один формальный механизм, называемый Алголом. Многие сразу возразят, указав, что среди нас слишком мало людей, которые понимали его или пользовались им. Хотя, к сожалению, они окажутся правы, но дело не в этом. Существен

данный Алголом стимул к развитию исследований в информатике, а не число его последователей. К тому же Алгол мобилизовал наше мышление и обеспечил нас основой для дискуссий.

Я долго ломал себе голову над тем, почему Алгол оказался столь полезной моделью в нашей области деятельности. Возможно, к числу причин относятся следующие.

(а) Его международная поддержка.

(б) Ясность опубликованного описания его синтаксиса.

(в) Естественное сочетание в нем важных для программирования понятий сборки и подпрограммы.

(г) Тот факт, что язык можно естественным образом разбить на части, причем возможно предлагать и описывать довольно обширные модификации частей языка, не нарушая впечатляющую гармонию его структуры и нотации. Есть определенный смысл в термине «алголоподобный», который часто используется в рассуждениях о программировании, языках и вычислениях. Алгол, по-видимому, является жизнестойкой моделью и даже расцветает при хирургических вмешательствах — будь то обследования, пластические операции или ампутации.

(д) Тот факт, что он мучительно не приспособлен для многих задач, которые мы хотим программировать.

В одном я уверен: Алгол обязан своими магическими свойствами не процессу своего рождения, т. е. не тому, что это плод коллективного творчества. Итак, мы не должны разочаровываться, когда из таким же образом оплодотворенных яиц выходят более тупые модели. Эти последние, хотя и блещут впечатляющими улучшениями Алгола, вызывают у нас только скуку. Может быть, они и представляют собой улучшения Алгола, но не могут претендовать на роль преемников этой модели.

Конечно, мы должны и будем извлекать пользу из тех усовершенствований, которые позволяют исправлять недостатки Алгола. Нам следует подумать также и о том, почему они не могут стимулировать нашу творческую энергию. Нам следовало бы задаться вопросом, почему при их влиянии на научные исследования и даже на практику программирования они не обеспечивают качественного скачка вперед. Я не претендую на знание исчерпывающего ответа, но уверен, что значительная часть этой ограниченности обусловлена тем, что внимание сосредоточивается не на тех слабостях Алгола, на которые следует его обратить.

СИНТЕЗ ЯЗЫКА И СТРУКТУР ДАННЫХ

Мы знаем, что проектируем язык, чтобы упростить выражение неограниченного количества алгоритмов, создаваемых для важного класса задач. Проектирование языка следует предпри-

нимать только тогда, когда алгоритмы для этого класса предполагают или, вероятно, обещают после некоторого развития значительную нагрузку на компьютеры, а программирование этих алгоритмов в случае использования существующих языков потребует значительных затрат времени. В такой ситуации новый язык должен сократить стоимость некоторого множества выполняемых работ, чтобы окупить затраты на свое проектирование, поддержание и усовершенствование.

Языки-преемники появляются по ряду причин.

(а) Исправление ошибки, недостаточности или избыточности в имеющемся языке *предполагает* естественное перепроектирование, результатом которого явится язык более высокого качества.

(б) Исправление ошибки, недостаточности или избыточности *требует* перепроектирования для получения полезного языка.

(в) На основе двух существующих языков обычно можно создать третий, который (1) содержит возможности обоих этих языков в интегрированной форме и (2) требует менее сложной грамматики и правил вывода, чем простое объединение грамматик и правил вывода, имеющихся в обоих языках.

Имея в виду сказанное выше, как можно было бы приступить к синтезу модели-преемника, которая не только улучшала бы коммерческие характеристики использования компьютеров, но и способствовала бы сосредоточению нашего внимания на важных проблемах внутри самого вычислительного процесса?

Я полагаю, что естественной отправной точкой должна быть организация и классификация данных. Было бы по меньшей мере затруднительно создать алгоритм, не зная природы обрабатываемых им данных. Когда мы пытаемся представить алгоритм на языке программирования, то должны узнать, как представляются на этом языке данные алгоритма, прежде чем сможем надеяться на полезные вычисления.

Поскольку наш преемник мыслится как универсальный язык программирования, он должен обладать общими структурами данных. В зависимости от точки зрения это может оказаться и не столь трудным, и не столь легким, как ожидается. Как следует организовать это представление структур данных? Посмотрим, что было сделано в уже имеющихся языках. Прежний подход состоял в следующем.

(а) В языке описывается несколько «примитивных» структур данных, например целые числа, вещественные числа, однородные по типу массивы, списки, строки и файлы.

(б) На этих структурах обеспечивается «достаточный» набор операций, например арифметические, логические, выделение, присваивание и комбинирование.

(в) Любые другие структуры данных считаются непримитивными и должны представляться через примитивные. Внутренняя организация в непримитивных структурах явно обеспечивается операциями над примитивными данными, например, взаимосвязь между вещественной и мнимой частями комплексного числа реализуется вещественной арифметикой.

(г) Достаточный набор операций для этих непримитивных данных организуется в виде процедур.

Этот процесс расширения нельзя обойти. Всякий язык программирования должен допускать легкое использование, поскольку в конечном счете это всегда требуется. Однако если злоупотреблять этим процессом расширения, то в алгоритмах часто исчезает та ясность структуры, которой они в действительности обладают. Что еще хуже, зачастую они неоправданно медленно выполняются. Первая слабость возникает потому, что для алгоритма язык был выбран неверно, а вторая — из-за того, что язык навязывает избыточную организацию данных и требует во время исполнения такого администрирования, которое могло бы быть выполнено однократно до исполнения алгоритма. В обоих случаях переменные связывались в неподходящее время правилами синтаксиса и вычисления.

Я думаю, все мы осознаем, что в наших языках недостаточно типов данных. Разумеется, в нашей модели преемника нам не следует пытаться возместить этот недостаток добавлением новых, например ограниченного количества новых типов и какой-то общей всеобъемлющей структуры.

Наш опыт описания функций должен подсказывать нам, что нужно делать: не сосредоточиваться на полном множестве описанных функций для общего пользования, а обеспечить в рамках языка такие структуры и управление, из которых следовало бы эффективное описание и использование функций в программе.

Поэтому нам нужно применительно к нашей модели преемника сконцентрировать внимание на обеспечении средств описания структур данных. Но этого недостаточно. В программе, для которой специфицируются структуры данных, должны быть еще представлены «достаточное» множество сопровождающих операций, контексты, в которых они встречаются, а также соответствующие правила вычисления.

Список некоторых возможностей, которые должны быть обеспечены для структур данных, мог бы включать:

(а) описание структуры;

(б) присваивание структуры идентификатору, т. е. придание идентификатору ячеек информации;

(в) правила именования частей для заданной структуры;

(г) присваивание значений для ячеек, отведенных для идентификатора;

(д) правила ссылок на ячейки, отведенные для идентификатора;

(е) правила комбинирования, копирования и стирания как структуры, так и содержимого ячеек.

Конечно, и теперь в большинстве языков эти возможности обеспечиваются в той или иной степени, но обычно слишком фиксированным способом, в виде правил синтаксиса и вычисления.

Мы знаем, что создатели языка не могут устанавливать, сколько информации должно находиться в структуре, а сколько — в данных, переносимых в этой структуре. Для каждой программы должен быть предоставлен естественный для нее выбор достижения баланса между временем и расходом памяти. Мы знаем, что не существует единого способа представления массивов или списковых структур, строк, файлов или их комбинаций. Выбор способа зависит от

(а) частоты доступа;

(б) частоты структурных изменений, в которых задействованы указанные данные, например присоединений к файлу новых структур записей или изменений границ массивов;

(в) затрат на излишнюю машинную память;

(г) затрат на излишнее время доступа к данным и

(д) важности алгоритмического представления, пригодного для упорядоченного наращивания, при котором всегда сохраняется ясность структуры.

Такой выбор, как всем известно, труден для программиста, а на уровне проектирования он совсем невозможен.

Структуры данных нельзя строить из воздуха. В сущности, наш обычный метод состоит в использовании базовой гипотетической машины с фиксированными примитивными структурами данных. Эти структуры — те, которые идентифицируются с реальными компьютерами, но базовая машина может быть более абстрактной применительно к описанию структур данных. После выбора базовой машины требуемые, согласно нашим описаниям, дополнительные структуры должны представляться как данные, т. е. как имя или указатель на структуру. Не все указатели ссылаются на однотипные структуры. Поскольку сегменты программы сами являются структурами, такие указатели, как «идентификатор процедуры с содержимым (x)», устанавливают класс переменных, значениями которых являются имена процедур.

КОНСТАНТЫ И ПЕРЕМЕННЫЕ

На самом деле гибкость языка измеряется тем, *что* программистам разрешается изменять (как в композиции, так и

в исполнении). Систематическое развитие изменчивости в языке является центральной проблемой для программирования, а следовательно, и для проектирования нашего преемника. Всегда практика подсказывает нам особые случаи, на основании которых мы устанавливаем описания новых переменных. Каждый новый опыт сосредоточивает наше внимание на необходимости большей общности. Разделение времени является одним из новых явлений, которое, вероятно, станет повседневным. Разделение времени концентрирует наше внимание на управлении системами и на том, чтобы программисты управляли своими текстами до, во время и после их исполнения. Возрастает гибкость взаимодействия с программой, и наш преемник не должен затруднять этот процесс. Наше представление о диалоговом программировании означает много больше, чем быстрота и удобство отладки: наши наиболее интересные программы никогда не бывают совсем ошибочными, но и никогда не бывают окончательными версиями. Как программисты, мы должны выделить то новое, что вносит диалоговое программирование, прежде чем сможем надеяться обеспечить для него подходящую языковую модель. Я считаю, что новизна состоит в требовании изменяемости того, что прежде считалось фиксированным. Здесь я имею в виду не новые классы данных, а переменные, значениями которых являются программы или части программ, синтаксис или части синтаксиса и режимы управления.

Основное внимание теперь уделяется разработке систем управления файлами, которые улучшают управление всей системой, и относительно немного — улучшению управления вычислениями. Если первое может быть обеспечено помимо языков, на которых мы пишем свои программы, то для второго нам нужно упрочить свой контроль над изменчивостью в рамках языка программирования, используемого нами для решения своих проблем.

При обработке текста программы некий сегмент текста может встретиться один раз, но выполняться неоднократно. Поэтому возникает потребность идентифицировать и постоянство, и изменчивость. В общем случае мы берем то, что по форме является переменной, и делаем это постоянной в процессе инициализации; часто разрешается, чтобы сам этот процесс мог повторяться. Данный процесс инициализации является фундаментальным, и в нашем преемнике должен иметься систематический способ делать это.

Рассмотрим некоторые примеры инициализации и изменчивости в Алголе.

(а) *Вход в блок.* При входе в блок описания обеспечивают инициализацию, но только относительно некоторых свойств идентификаторов. Так, **integer** *x* инициализирует свойство при-

надлежности к типу целых чисел, но нет возможности инициализировать значения x как нечто, что не будет изменяться в области действия блока. Описание **procedure** $P(...)$; ...; выразительно инициализирует идентификатор P, но нет возможности изменять его внутри блока. Описание **array** $A\,[1:n,\ 1:m]$ присваивает начальную структуру. Нет возможности инициализировать значения ячеек этой структуры или изменять структуру, присвоенную идентификатору A.

(б) *Оператор цикла.* Не могут быть инициализированы выражения, которые я называю шаговым и конечным элементами.

(в) *Описание процедуры.* Инициализируется идентификатор процедуры. При вызове процедуры ее формальные параметры инициализируются, как и идентификаторы процедур, и они могут быть даже инициализированы по значению. Впрочем, разные обращения устанавливают различные инициализации идентификаторов формальных параметров, но не разные способы инициализации значений.

Предоставляемый в Алголе вариант связывания формы и содержания с идентификаторами считался подходящим. Однако если мы посмотрим на присваивание формы, вычисление формы и инициализацию как на важные функции, которые должны быть рационально специфицированы в языке, то можем счесть Алгол ограниченным и даже вычурным языком с точки зрения предоставляемого им выбора. Нужно надеяться на то, что язык-преемник окажется гораздо менее произвольным и ограниченным.

Я позволю себе привести тривиальный пример. В операторе цикла использование в качестве шагового элемента такой конструкции, как **value** E, где E — это выражение, должно было бы сигнализировать об инициализации выражения E. **value** — это разновидность оператора, управляющего связыванием значения с формой. Существует естественная область действия, соответствующая всякому применению этого оператора.

Я упоминал, что идентификаторы процедур инициализируются через описания. В таком случае связывание процедуры с идентификатором может быть изменено присваиванием. Я уже отмечал, как это делается с помощью указателей. Разумеется, существуют и другие способы. Простейший из них состоит в том, чтобы вовсе не заменять идентификатор, а пользоваться индексом выбора, который фиксирует одну процедуру из некоего множества. Теперь инициализация описывает массив форм, т. е. **procedure array** $P\,[1:k](f_1, f_2, \ldots, f_j)$; ... **begin** ... ... **end**;...; **begin** ... **end**; Вызов $P\,[j](a_1, a_2, \ldots, a_j)$ выбрал бы для исполнения j-е тело процедуры. Или же можно описать **procedure switch** P: $=A, B, C$ и процедурные указательные вы-

ражения, так что предыдущее обращение стало бы выбирать для исполнения j-е процедурное указательное выражение. Приведенные выше подходы оказываются слишком статичными для некоторых приложений, и им недостает важного свойства присваивания, а именно возможности определять, когда присвоенная форма не является более доступной, и следовательно, ее память можно использовать для иных целей. Возможными применениями таких динамически присваиваемых процедур являются генераторы. Предположим, у нас имеется процедура для вычисления (а) $\Sigma^{n}_{k=0} C_k(N) X^k$ как приближенного значения некоторой функции (б) $f(x) = \Sigma^{\infty}_{k=0} C_k X^k$ при специфицированной целой величине x. Теперь, найдя $C_k(N)$, мы заинтересованы только в вычислении (а) для различных значений x. Потом мы могли бы захотеть описать процедуру, которая подготавливает (а) на основании (б). Эта процедура при своем начальном выполнении присваивает либо себе самой, либо некоторому другому идентификатору процедуру, которая вычисляет (а). Последующие вызовы этого идентификатора только порождали бы созданное таким образом вычисление. Такое динамическое присваивание влекло бы за собой ряд привлекательных возможностей.

(а) В результате второго присваивания могла бы освободиться часть памяти программы.

(б) Память для данных может присваиваться как собственная (**own**) для идентификатора процедуры, описание которой создается.

(в) Начальный вызов может модифицировать результирующее описание; например, вызов по наименованию или вызов по значению формального параметра из исходного обращения может повлиять на вид полученного описания.

Легко видеть, что то, к чему мы пришли, — это необходимость единообразного подхода к инициализации и изменчивости формы и значения, относящихся к идентификаторам. В этом состоит требование вычислительного процесса. Сам по себе наш язык-преемник должен располагать общим способом управления действиями по инициализации и изменениям применительно к классам идентификаторов.

В частности, мы хотим выполнять в диалоговом программировании систематическое, или управляемое, изменение значений данных и текста в отличие от несистематических изменений, которые возникают в процессе отладки. Разумеется, выполнение таких действий означает, что определенные части текста понимаются как переменные. И снова мы достигаем этого с помощью определений, инициализации и присваивания. Поэтому мы можем писать в заголовке блока такие описания:

real x, s;
arithmetic expression t, u;

В сопровождающем тексте появление оператора $s:=x+t$ вызывает сложение значения арифметического выражения, присвоенного переменной t, например, по вводу, со значением x и присваивание результата в качестве значения переменной s. Мы видим, что t может вводиться и храниться как некая форма. В таком случае операция + может выполняться только после применения подходящей функции преобразования. Нас не должно отпугивать то обстоятельство, что в классическое «время трансляции» может быть выполнена лишь частичная трансляция выражения. Настало время систематического подхода к проблеме частичной трансляции. Естественными изменяемыми частями текста являются те части, которые идентифицируются синтаксическими единицами языка.

Несколько более трудно принимать меры в связи с непредумышленным изменением программ. При этом основная проблема состоит в идентификации изменяемой части исходного текста и отыскании для нее соответствия в процессе трансляции с фактически вычисляемым текстом. Легко сказать: проинтерпретируй исходный текст. Но нужно найти удовлетворительный баланс затрат в рамках промежуточных решений между трансляцией и интерпретацией. Я надеюсь выразить в следующем разделе точку зрения, которая позволит пролить некоторый свет на достижение этого баланса в каждой программе, где это требуется.

СТРУКТУРА ДАННЫХ И СИНТАКСИС

Даже если списковые структуры и рекурсивное управление не станут играть центральной роли в нашем языке-преемнике, он будет тесно связан с Лиспом. Этот язык вызывает забавные споры среди программистов, зачастую проклинающих и восхваляющих одни и те же особенности языка. Здесь я хотел бы только отметить, что описание Лиспа точно раскрывает соответствующие компоненты языковых средств с большей ясностью, чем в любом другом известном мне языке. Описание Лиспа включает не только его синтаксис, но и представление его синтаксиса и структуры данных среды в виде структуры данных языка. Фактически такое описание несколько ограничивает дальнейшее описание, но несущественно. На основании предшествующих описаний становится возможным описать процесс вычисления как программу на Лиспе с применением нескольких примитивных функций. Хотя такая завершенность

описания возможна и для других языков, в общем случае она не рассматривается как часть их определяющего описания.

Изучение Алгола показывает, что его структуры данных непригодны для представления текстов на Алголе, по крайней мере тем способом, который годится для описания вычислительной схемы языка. Аналогичное замечание относится и к его непригодности для описания внешних структур данных в программах на Алголе.

Я считаю критическим требование, чтобы наш язык-преемник обеспечил баланс структур данных, подходящих для представления синтаксиса и среды, так чтобы процесс вычисления можно было ясно сформулировать на этом языке.

Почему столь важно дать такое описание? Только ли для придания языку изящного свойства замкнутости, чтобы можно было организовать начальную загрузку? Вряд ли. Это является ключом к систематическому конструированию программных систем, пригодных для диалоговых вычислений.

Язык программирования характеризуется синтаксисом и набором правил вычислений. Последние связаны друг с другом посредством представления программ как данных, к которым применяют правила вычисления. Эта структура данных представляет собой внутренний или вычислительно ориентированный синтаксис языка. Мы составляем программы во внешнем синтаксисе, который фиксируется в целях общения между людьми. Внутренний синтаксис, как правило, считается столь зависимым от машины и транслятора, что он почти никогда не описывается в литературе. Обычно существует процесс трансляции, который переводит текст из внешнего во внутреннее синтаксическое представление. Фактически изменение внутреннего описания более существенно связано с правилами вычисления, чем с машиной, на которой они должны выполняться. Выбор правил вычисления критически зависит от времени связывания переменных языка.

Тем самым указывается подход к организации вычислений, полезной в случае изменяемых текстов. Поскольку внутренняя структура данных отражает изменчивость обрабатываемого текста, пусть подходящее внутреннее представление синтаксиса выбирается в процессе трансляции и пусть общий вычислитель выбирает конкретные правила вычисления на основе предпочтенной синтаксической структуры. Итак, нам нужно задать во внешнем синтаксисе ключи, которые указывают переменные. Например, появление **arithmetic expession** *t*; **real** *u*, *v*; и оператора *u*: =*v*/3***t*; указывает возможность различного внутреннего синтаксиса для *v*/3 и значения *t*. Следует отметить, что *t* по своему поведению весьма напоминает формальный параметр Алгола. Впрочем, здесь менее организован контроль над при-

свиванием. Я думаю, что из этого следует только, что присваивания типа формальный параметр — фактический параметр не зависят от концепции замкнутой подпрограммы и что они объединены в конструкции процедуры как способ спецификации области действия инициализации.

В случае непредусмотренного заранее изменения программы знание внутренней синтаксической структуры позволяет свести к минимуму объем перетрансляции и изменения правил вычисления при изменении текста.

Поскольку нужно изучать и конструировать структуры данных и правила вычисления на некотором языке, представляется целесообразным, чтобы это был сам исходный язык. Можно определить в качестве цели трансляции внутренний синтаксис, в котором цепочки литер являются подмножествами текстов, допустимых в исходном языке. Если такой выбранный синтаксис близок к машинному коду, то его можно потом обрабатывать по правилам, весьма сходным с машинными.

Пока речь шла об изменяемости применительно к идентификаторам языка, и я ничего не сказал об изменяемости управления. В сущности, у нас нет способа описания управления, и поэтому мы не можем определять его организацию. Следует ожидать, что наш язык-преемник будет обладать некой формой управления наподобие принятой в Алголе — и не более того. Много исследований было посвящено такому виду управления, как параллельная работа. Кроме того, недавно в языках начало появляться распределенное управление, которое я буду называть мониторингом. Процесс A непрерывно осуществляет мониторинг процесса B таким образом, что, когда B достигает некоторого состояния, процесс A перехватывает управление дальнейшей деятельностью этого процесса. Управление в рамках процесса A может быть записано в виде **when** P **then** S; P — это предикат, который проверяется всегда в пределах некоторого определенного контекста. Всякий раз, когда P есть истина, подпадзорное вычисление прерывается и выполняется S. Мы хотим механизировать эту конструкцию, проверяя P всякий раз, когда выполняется некое действие, которое, возможно, могло бы сделать P *истиной,* но ни в каких других случаях. В таком случае мы должны при описании языка, среды и правил вычислений включить состояния, которые могут подвергаться мониторингу во время исполнения. На основании этих примитивных состояний можно сконструировать посредством программирования и другие состояния. Зная эти примитивные состояния, можно предусматривать тестовые вставки в возможных точках еще до того, как конкретные предикаты определены в пределах программы. В таком случае мы можем диагностировать наши программы, не нарушая их целостности.

ИЗМЕНЕНИЯ СИНТАКСИСА

В ограниченных рамках одного языка допустимы некоторые вариации. Однако весь наш опыт свидетельствует о том, что наши потребности в изменениях будут оказывать возрастающее давление на сам язык, стимулируя его изменения. Разработчики не в состоянии предугадать точные характеристики этих изменений, потому что те явятся следствиями еще ненаписанных программ для еще нерешенных задач. Увы, как раз наиболее полезные и успешные языки оказываются наиболее подверженными такому давлению. К счастью, довольно предсказуемы те изменения, которых нужно ожидать на ранних этапах. Например, в научных вычислениях представление чисел и арифметические действия над числами изменяются, но природа выражений не подвержена изменениям, кроме как через операнды и операции. Вызываемые этим модификации синтаксиса обозримы. В результате синтаксис и правила вычисления арифметических выражений остаются неопределенными в языке. Вместо этого в языке обеспечиваются правила синтаксиса и вычисления для программирования описаний арифметического выражения и для задания области действия таких описаний.

При этом единственная реальная трудность связана со спецификацией правил вычислений. При их задании следует соблюдать осторожность. Например, при введении таким образом арифметики матриц нужно вычислять матричные выражения с аккуратным использованием временной памяти, избегая необязательных итераций.

Естественный способ поупражняться в работе с описаниями состоит в том, чтобы начать с языка X, рассмотреть описания как расширение синтаксиса до синтаксиса языка X' и задать правила вычислений как процесс редукции, который преобразует любой текст на языке X' в эквивалентный текст на языке X.

Следует заметить, что изменение синтаксиса требует представления этого синтаксиса предпочтительно в виде структуры данных самого языка X.

ЗАКЛЮЧЕНИЕ

Языки программирования строятся вокруг переменной — операций над ней, структур управления и данных. Поскольку эти понятия являются общими для всего программирования, общий язык должен сосредоточиваться на их последовательном развитии. Хотя мы в большом долгу перед Тьюрингом за предложенную им образцовую модель, нас не смущает работа с машинами и данными, превосходящими по своей сложности тот

уровень, который Тьюринг считал необходимым. Программисты никогда не удовлетворятся языками, позволяющими им программировать что угодно, но не позволяющими удобно и легко делать то, что их действительно интересует. Поэтому наш прогресс оценивается достигаемым нами балансом между эффективностью и общностью. С изменением характера использования вычислений (а это происходит) изменяется пригодное для наших целей определение языка и наиболее актуальными становятся иные проблемы. Я предчувствую, что наша модель преемника будет отражать такое изменение. Информатика — непоседливое дитя; ее прогресс столь же зависит от изменений в подходе, как и от последовательного развития наших современных концепций.

Ни одна из представленных здесь идей не нова; просто время от времени о них забывали.

Хочу поблагодарить Ассоциацию за предоставленную мне честь прочитать эту первую Тьюринговскую лекцию. И лучше всего ее закончить словами, что, если бы Тьюринг был сегодня с нами, он сказал бы о другом в лекции с иным названием.

ПОСТСКРИПТУМ

А. Дж. Перлис
Факультет информатики
Йельский университет

В столь динамично развивающейся проблематике, как программирование, не следует ожидать, что статья двадцатилетней давности останется последним словом науки. Поэтому я был приятно удивлен, обнаружив очевидные интерпретации ее содержания, согласующиеся с тем, что происходило и все еще происходит с языками программирования.

Мы по-прежнему придаем огромное значение модели вычислений, которая предстает перед нами в обличье языка программирования. Большинство новых языков, овладевших нашим воображением, придают этим моделям сладость синтаксиса и обогащают их семантику, так что теперь нас соблазняют такие честолюбивые эксперименты, которые мы зарекались проводить прежде. Рассмотрим четыре такие модели: конвейеры (APL), распределенное программирование (более известное под названием объектно-ориентированное программирование, примером может служить язык Смолток), редукционное программирование (функциональное программирование, примеры — языки Лисп, EP и ML) и непроцедурное программирование (например, логическое программирование на Прологе). Эти модели завладели нашими умами примерно так же, как Алгол 25 лет назад. У нас нет оснований полагать, что это последние модели, способные побудить нас попытаться достичь еще более грандиозных обобщений.

Моя лекция была сосредоточена на важности структур данных в программировании, а следовательно, в языке программирования. Одно из направлений развития языков состоит в возрастающем усложнении описаний и управления структурами данных. Причем Алгол и его прямые «отпрыски» ориентировались на четко определенную изменяемость структур данных, ведущую к записям и к теориям типов, а упомянутые нами выше модели опирались на свою зависимость от одной составной структуры данных, напри-

мер списка или массива. Разумеется, по мере расширения их использования забота об изменчивости структур данных этих моделей должна возрасти.

Рабочая станция, персональный компьютер и сеть тогда еще не стали общепринятым инструментарием, в который надлежало интегрировать системы языков программирования. Редакторы были примитивными и никоим образом не выглядели как волшебная дверь, через которую мы попадаем в страну вычислений. Тем не менее в лекции нашла отражение важность диалоговых вычислений и языков для поддержки таких вычислений. Было отмечено, что представление программ как данных являлось критическим для подобного программирования. Оказывается, что для такого представления списки подходят лучше, чем массивы, поскольку синтаксис представляет собой набор вложенных и подставляемых ограничений, что совпадает с правилами модификации списковых структур. Язык APL и конвейерная организация пострадали из-за этого неравноправия между массивами и списками. В новых версиях APL предпринимаются попытки преодолеть это неравноправие.

Программирование становится повсеместной деятельностью, и предпринимаются энергичные усилия, чтобы стандартизировать небольшое количество языков (моделей и средств). До сих пор мы сопротивлялись такому ограничению, и это было мудро с нашей стороны. Новые архитектуры и проблемные области, безусловно, подскажут новые вычислительные модели, которые потрясут наше воображение. От них и от нынешнего состояния программирования произойдет новое видение языка, играющего роль проекта Ада. Как всегда, мы по-прежнему будем избегать ловушки Тьюринга: необходимости пользоваться языками, которые позволяют делать все, но не делают простым и удобным ничего из того, в чем мы действительно заинтересованы.