МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
имени М.В.ЛОМОНОСОВА

Факультет вычислительной математики и кибернетики

УДК 519.683.8

А. В. Столяров

Парадигмы программирования и алгебраический подход к построению универсальных языков программирования

Авторские права © Андрей Викт. Столяров, 2007

Рукопись депонирована в ВИНИТИ РАН
11.09.2007, №866-В2007

Москва
2007

Оглавление

Глава 1

Введение

1.1 Многообразие языков программирования

Появившиеся в конце 50х годов прошедшего столетия языки программирования высокого уровня показали, что осмысление компьютерной программы возможно в терминах иных, нежели привычные на тот момент номера инструкций процессора и адреса ячеек памяти. Уже самые ранние языки, среди которых были Фортран [10] и Lisp [29], предоставили в распоряжение программиста такие высокоуровневые абстракции, как математические формулы, символические выражения, рекурсивные функции [30] и т.п.

С появлением первых языков программирования высокого уровня связано начало бесконечного процесса поиска «лучшего» (в том или ином смысле) языка. Уже в 1969 году Дженет Самметт (J. Sammett) в своей книге [32] описывает более 120 языков программирования. К настоящему времени общее число разработанных языков программирования оценивается несколькими тысячами.

Несмотря на это, новые языки программирования продолжают создаваться, что само по себе показыват отсутствие среди всего многообразия языков программирования одного «лучшего» языка.

1.2 Выбор языка

Многие авторы отмечают, что правильный выбор языка программирования является ключевой проблемой, от которой существенно зависит успех любого программистского проекта. Попытаемся проанализировать критерии, по которым осуществляется такой выбор.

1.2.1 Выбор по особенностям изобразительных средств

Дейв Баррон (Dave Barron) в книге [12] утверждает:

> «*...для научного работника, использующего ЭВМ, язык программирования - нечто большее, чем просто средство описания алгоритмов: он несет в себе систему понятий, на основе которых человек может обдумывать свои задачи, и нотацию, с помощью которой он может выразить свои соображения по поводу решения задачи*».

Эдсгер Дейкстра в своей лекции лауреата премии Тьюринга, озаглавленной «Смиренный программист» [20], подчеркивает:

> «*Все мы знаем, что единственное мыслительное средство, посредством которого вполне конечный фрагмент рассуждения может охватывать миллионы случаев, называется "абстракцией". Поэтому наиболее важным видом деятельности компетентного программиста можно считать эффективную эксплуатацию его способности к абстрагированию*».

> «*...Инструменты, которые мы пытаемся использовать, а также язык или обозначения, применяемые нами для выражения или записи наших мыслей, являются главными факторами, определяющими нашу способность хоть что-то думать и выражать!*»

Определяющую роль выбора языка программирования с точки зрения изобразительных средств прекрасно иллюстрирует пример, приведенный Тимоти Баддом в [17]. В этом примере два программиста решают задачу выделения повторяющихся последовательностей длиной не менее M в массиве целых размером $N >> M$, при том что число N достаточно велико (порядка нескольких десятков тысяч). Первый программист решает задачу на Фортране с помощью двух вложенных циклов, получая в результате неудовлетворительно медленное решение. Второй же, используя язык APL, строит матрицу из строк, представляющих собой сдвиги основного массива, сортирует строки матрицы с использованием встроенной возможности APL, и, несмотря на то, что APL, в отличие от Фортрана, является языком интерпретируемым и заведомо проигрывает последнему в эффективности, получает решение, на несколько порядков более эффективное по времени исполнения. Автор заостряет внимание читателя на том, как наличие частной встроенной возможности (сортировка произвольных векторов) может натолкнуть программиста на более изящное решение проблемы.

Следует отметить, что предметная область, в которой предстоит работать программе, подлежащей выполнению, оказывает существенное влияние на предпочтения относительно изобразительных возможностей выбираемого языка. Так, если в решаемой задаче предполагается работа со слабо структурированными данными (например, со сложными математическими формулами, с информацией в виде XML-деревьев и т.п.), для такой предметной области хорошо подойдёт язык Lisp благодаря наличию в этом языке S-выражений [30]. В то же время, если необходимо достаточно быстро обработать и преобразовать некий текст (например, произвести генерацию отчета по входным данным, записанным в некотором строгом формате, поддающемся машинному анализу), логично рассмотреть кандидатуры скриптовых языков, таких как Perl или Tcl. В свою очередь, для решения переборной задачи хорошо подошел бы язык Prolog [19].

1.2.2 Уровень абстрагирования языка и его влияние на выбор

Языки программирования традиционно делились на высокоуровневые и низкоуровневые. Следует заметить, что смысл этих терминов с течением времени

менялся. Если изначально под языком низкого уровня понимался исключительно язык ассемблера, а все остальные языки программирования считались языками высокого уровня, причем о сравнении относительной «высоты уровня» двух высокоуровневых языков речь не шла, то со временем концепция уровня языка стала более гибкой. С одной стороны, этому способствовало появление языка программирования C [26], который, не являясь языком ассемблера, позволял, тем не менее, описывать программу в терминах, близких к машинным. С другой стороны, с появлением всё большего количества языков программирования становилась более ясной недостаточная выразительность термина «язык высокого уровня», т.к. языки, являющиеся таковыми, оказывались очевидно по-разному удалены от машинного языка и возможностей машины; так, язык Pascal [43] имеет с возможностями машины гораздо больше общего, чем, скажем, язык Prolog [19] или язык Lisp [36][1].

Наконец, в последние 10–15 лет из программистской практики оказался почти полностью вытеснен язык ассемблера, вместо которого чаще всего используется язык C. Следует заметить, что в задачах, ранее решавшихся на ассемблере, обычно практически невозможно использовать большинство языков высокого уровня; C в этом смысле явно отличается от многих других языков.

Все это сделало термин «язык высокого уровня» неадекватным для классификации языков программирования. В результате концепция уровня языка трансформировалась из бинарной (низкий–высокий) в относительную (язык L_1 имеет уровень более высокий, нежели язык L_2).

Уровень языка является важным, а в некоторых случаях и определяющим критерием при выборе. Так, существуют проблемные области, в которых применение языков достаточно высокого уровня (в частности, языков, имеющих встроенный механизм сборки мусора) оказывается невозможным. К таким проблемным областям относятся, в частности, создание систем реального времени, разработка ядер операционных систем, программирование микроконтроллеров и т.п.

В то же время существуют и такие проблемные области, в которых оказывается неоправданным (хотя и возможным) применение языков программирования сравнительно низкого уровня. К таковым относятся самые разнообразные области, от задач организации документооборота на предприятии до задач искусственного интеллекта.

1.2.3 Практические ограничения

Практическое программирование как вид производственной деятельности часто накладывает дополнительные требования на выбор языка программирования.

Так, подавляющее большинство языков следует отсеять, даже не приступая к их изучению, в силу того, что для этих языков нет ни технических инструментов (систем программирования) под нужную платформу, ни информационной

[1]Это видно хотя бы из того факта, что динамические структуры данных в языках Lisp и Prolog входят в число первичных понятий языка, тогда как в языке Pascal их следует строить в явном виде; также языки Lisp и Prolog имеют механизмы автоматической сборки мусора, каковые в низкоуровневых терминах реализуются достаточно сложным образом; наконец, и сами модели вычисления, т.е. собственно работы программы, в этих языках не имеют ничего общего с низкоуровневыми вызовами функций и машинными командами, тогда как в языке Pascal достаточно очевидны правила, по которым конструкциям языка ставятся в соответствие фрагменты машинного кода.

поддержки в виде учебников и справочников, ни, наконец, возможности найти достаточное количество программистов, владеющих данным языком. Такие языки следует считать мертвыми.

В некоторых специальных случаях возможно приложение усилий для реанимации мертвого языка специально для нужд того или иного проекта, однако такие случаи редки. Дело в том, что практически всегда среди действующих языков программирования можно отыскать язык, близкий по своим характеристикам к данному мертвому, либо, в худшем случае, несколько языков, в совокупности предоставляющих те же возможности. Кроме того, все языки программирования обладают алгоритмической полнотой[2], что означает, что любая программа может быть реализована на любом языке. Возможно, что язык, на котором мы вынуждены в силу каких-либо причин остановиться, потребует бóльших усилий для решения конкретной подзадачи, однако в большинстве случаев эти усилия окажутся меньшими, чем те, что были бы потрачены на реанимацию мертвого языка или тем более создание нового.

В тех редких случаях, когда неадекватность имеющихся средств требованиям предметной области оказывается слишком высокой, как правило, и возникают новые языки. Впрочем, если для начала эры языков высокого уровня такая ситуация была харатерной, то сегодня ее следует рассматривать скорее как нечто исключительное.

Особенности подготовки программистского коллектива также налагают определенные ограничения на выбор. В случае, если некий язык, пусть даже идеально подходящий под поставленную задачу, не известен при этом будущим исполнителям, приходится учитывать еще и сравнительно высокую стоимость переподготовки специалистов, а также и потенциальный риск, связанный с возможностью увольнения некоторых участников проекта, что повлечет необходимость замены ушедшего специалиста новым специалистом, которого, в свою очередь, придется обучать выбранному языку.

1.3 Термин «парадигма» и его применение в программировании

При классификации языков программированиячасто упоминается термин *парадигма программирования*. В литературе этот термин встречается в различных значениях и единого взгляда на то, что же следует называть парадигмой программирования, пока не выработано. Так, Роберт Флойд в своей лекции [22] называет в качестве примера парадигмы структурное программирование. В то же время другие авторы (см, например, [35]) предпочитают считать, что структурное программирование как таковое парадигмой программирования не является. В связи с этим представляется необходимым уделить определенное внимание самому термину.

[2]Разумеется, это верно только в случае, если мы договоримся не рассматривать языки, служащие иным целям, нежели написание программ - например, языки HTML, TeX или SQL

1.3.1 Происхождение

Термин *парадигма* произошел от греческого слова $\pi\alpha\rho\alpha\delta\epsilon\iota\gamma\mu\alpha$, которое можно перевести как *пример* или *модель*. Своим современным значением термин обязан, по-видимому, Томасу Куну и его книге «Структура научных революций» [27]. Кун называл парадигмами устоявшиеся системы научных взглядов, в рамках которых ведутся исследования. Согласно Куну, в процессе развития научной дисциплины может произойти замена одной парадигмы на другую (как, например, геоцентрическая небесная механика Птолемея сменилась гелиоцентрической системой Коперника), при этом старая парадигма еще продолжает некоторое время существовать и даже развиваться благодаря тому, что многие ее сторонники оказываются по тем или иным причинам неспособны перестроиться для работы в другой парадигме.

Роберт Флойд в [22] отмечает, что подобное явление можно наблюдать и в программировании. Тем не менее, в основном парадигмы программирования не являются взаимоисключающими:

> «*Если прогресс искусства программирования в целом требует постоянного изобретения и усовершенствования парадигм,* – пишет Флойд – *то совершенствование искусства отдельного программиста требует, чтобы он расширял свой репертуар парадигм.*»

Иными словами, в отличие от парадигм в научном мире, описанных Куном, парадигмы программирования могут сочетаться, обогащая инструментарий программиста.

1.3.2 Экстенсиональное рассмотрение понятия парадигмы программирования

Прежде чем попытаться обобщить встреченные в литературе определения парадигмы программирования, рассмотрим наиболее часто приводимые примеры парадигм.

Императивное программирование часто рассматривается как наиболее традиционная модель программистского мышления. В основе императивного программирования лежат понятия переменной и присваивания (или, иначе говоря, смены состояния). Джон Бэкус (John Backus) использует для обозначения этой парадигмы термины «стиль фон Неймана» и «языки фон Неймана» [11], отдавая дань концепции, благодаря которой императивное программирование, по-видимому, и возникло.

Понятие императивного программирования часто смешивают с понятием *процедурного программирования*, характерной особенностью которого является разбиение программы на части (процедуры), вызываемые ради побочного эффекта.

В качестве примера чисто императивного языка программирования можно назвать ранние версии языков Basic или Fortran [10]. Примером процедурного языка могут служить Pascal [43], C [26] и другие языки.

Другая часто упоминаемая парадигма – *функциональное программирование* [11, 21] – предлагает осмысливать программу как систему взаимозависимых функций, вычисляющих значение на основании заданных аргументов. В чистом

функциональном программировании побочные эффекты запрещены, что делает его изобразительную мощность недостаточной для создания интерактивных программ[3].

В качестве примера функционального языка часто называют Lisp [36], однако этот пример неудачен. Ни одна версия языка Lisp не была чисто функциональной, хотя, безусловно, программирование на языке Lisp стимулирует мышление в терминах функций. Тем не менее, в качестве примеров чисто функциональных языков правильней будет назвать языки Hope [18] и Miranda [41].

Здесь следует заметить, что термин «функциональное программирование» является весьма общим. Он покрывает, в числе прочих, такие концепции, как *программирование без побочных эффектов*, *функции как объекты данных*, *функции высоких порядков* (функционалы), *ленивые вычисления*, *карринг*, *монады* и т.п. По-видимому, каждая из этих концепций может быть названа парадигмой программирования. Нельзя не упомянуть также такое важное средство, как *рекурсия*, позволяющее говорить о (совсем уже узкой) парадигме *рекурсивного программирования*.

Ясно, что большинство языков программирования позволяет использовать рекурсию, но лишь функциональное программирование активно стимулирует ее применение, не предоставляя возможностей организации простых циклов и попросту не оставляя программисту выбора[4].

Основанное на исчислении предикатов *логическое программирование* [31] также часто называют одной из парадигм программирования. В логическом программировании (например, на языке Prolog [19]) программа рассматривается как набор логических фактов и правил вывода, а выполнение программы состоит в вычислении истинности (попытке доказательства) некоторого утверждения. С логическим программированием связывают понятие *декларативной семантики*, при использовании которой программист задает условия, которым должно удовлетворять решение, а само решение система находит автоматически[5]. Декларативная семантика порождает парадигму *декларативного программирования*.

В. Ш. Кауфман в [25] вводит понятие *ситуационного программирования* («модель Маркова-Турчина», язык Рефал [40]), ключевыми терминами которой являются анализ (исходной структуры) и синтез (результирующей структуры).

Начиная с середины семидесятых годов получило распространение *объектно-ориентированное программирование*. Гради Буч дает следующее определение ООП: *Объектно-ориентированное программирование – это методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования* [16].

---

[3] Строго говоря, это не совсем так, поскольку технология ленивых вычислений дает возможность создавать интерактивные программы, оставаясь в рамках чисто функционального программирования [21], однако такие построения слишком сложны для повседневного практического применения.

[4] Это обстоятельство, кстати, можно считать прекрасным аргументом в пользу изучения языка Lisp в курсах программирования в высшей школе, даже если принять спорный тезис о невостребованности языка Lisp индустрией.

[5] Справедливости ради следует отметить, что декларативная семантика встречается не только в логическом программировании. Примером декларативного языка, не имеющего отношения к логическому программированию, можно (с некоторой натяжкой) считать SQL [9].

Объектно-ориентированный подход позволяет воспринимать программу как набор объектов — «черных ящиков», а выполнение программы — как взаимодействие объектов между собой.

В качестве парадигм программирования часто называют также *продукционное программирование*, в основе которого лежат правила (продукции), по которым преобразуется исходная информация; *ограничительное программирование* (constraint programming), являющееся, по-видимому, разновидностью декларативного программирования; *конкурентное, или параллельное, программирование*, основанное на многопотоковом представлении вычислительного процесса, и другие. Бьерн Страуструп в качестве парадигм программирования, поддерживаемых языком C++, называет процедурное программирование, модульное программирование, парадигму пользовательских типов, парадигму абстрактных типов данных, виртуальные функции, объектно-ориентированное программирование и, наконец, обобщенное программирование (классы-шаблоны) [37].

В статье [34] в качестве парадигмы программирования упоминаются даже регулярные выражения и БНФ-грамматики, используемые во входных языках грамматических процессоров lex [28] и yacc [24].

1.3.3 Варианты интенсиональных определений

К сожалению, далеко не все авторы, использующие термин «парадигма», решаются дать интенсиональное определение используемому термину. Однако и те определения, которые удается найти, серьезно отличаются друг от друга.

Диомидис Спинеллис в работе [35] утверждает: *Слово «парадигма» используется в программировании для обозначения семейства нотаций, разделяющих общий путь описания реализаций программ*[6].

Для сравнения тот же автор приводит определения из других работ. В статье Дэниела Боброва [15] парадигма определяется как *стиль программирования как описания намерений программиста.* Брюс Шрайвер (Bruce Shriver) определяет парадигму как *модель или подход к решению проблемы* [33], Линда Фридман (Linda Friedman) - как *подход к решению проблем программирования* [23, стр. 188]. Памела Зейв (Pamela Zave) дает определение парадигмы как *способа размышления о компьютерных системах*[7] [44].

Питер Вегнер (Peter Wegner) предлагает другой подход к определению термина парадигмы программирования. В работе [42] парадигмы определяются как *правила классификации языков программирования в соответствии с некоторыми условиями, которые могут быть проверены.*

Тимоти Бадд предлагает понимать термин «парадигма» как *способ концептуализации того, что значит «производить вычисления» и как задачи, подлежащие решению на компьютере, должны быть структурированы и организованы* [17].

[6] В оригинале: “The word paradigm is used in computer science to talk about a family of notations that share a common way for describing program implementations”.

[7] В оригинале «way of thinking about computer systems».

1.3.4 Альтернативные термины

Многие авторы предпочитают использовать термин *стиль программирования* (например, [11, 16, 14]). В работе [14] стиль программирования определяется как *способ построения программ, основанный на определенных принципах программирования, и выбор подходящего языка, который делает понятными программы, написанные в этом стиле.* В качестве примеров стилей программирования приводятся, среди прочих, уже знакомые нам процедурно-ориентированный, объектно-ориентированный и логико-ориентированный. Это позволяет нам предположить, что понятия стиля программирования и парадигмы программирования обозначают примерно одно и то же. Отметим, что Бьерн Страуструп в книге [37] неявно отождествляет понятия *парадигмы* программирования, *техники* программирования и *стиля* программирования.

Примерно в том же значении В.Ш.Кауфман применяет термин *модель*, вводя понятия «модель фон Неймана» (императивное, или *операционное* программирование), «модель Маркова-Турчина» (ситуационное программирование), «модель Р» (реляционное программирование) и другие [25].

Для обретения большей уверенности в том, что *модель* по Кауфману означает то же самое, что и термин, рассматриваемый нами, напомним, что слово «модель» - это один из возможных переводов греческого слова «парадигма».

1.3.5 Рабочее определение

Проведенный обзор подтверждает тот факт, что, несмотря на широкое использование, термин парадигмы программирования к настоящему моменту нельзя считать устоявшимся. Тем не менее, ясны общие черты вариантов наполнения смысла этого термина в версиях различных авторов.

Поскольку из числа приведенных определений автор не смог однозначно выделить лучшее, вместо этого необходимо дать свой вариант, который и будет подразумеваться при использовании термина парадигмы программирования в настоящей работе.

Итак, ***парадигма программирования*** – **это набор логически связанных подходов к решению программистских задач вкупе с понятиями и концепциями, характерными для этих подходов.**

Под такое определение подпадают самые различные парадигмы, от объектно-ориентированного программирования до анализа текста по регулярным выражениям.

1.3.6 Языки программирования и парадигмы

Можно заметить, что парадигмы программирования обычно не являются принадлежностью конкретного языка. Рассмотрим, к примеру, парадигму функционального программирования, т.е. парадигму, в рамках которой программа воспринимается как набор взаимозависимых функций, не имеющих побочных эффектов, а выполнение программы представляется как вычисление некоторой функции при заданном значении аргументов.

Некоторые языки (например, Hope [21]) требуют работы именно в таких рамках. Выход за рамки функционального программирования в языке Hope невозможен.

Другие языки, такие как Lisp или Haskell [13], стимулируют применение функционального программирования, однако допускают и отступления от него в виде побочных эффектов функций, применения императивных конструкций и т.п.

Если рассмотреть такие языки, как C или Pascal, можно заметить, что эти языки *допускают* применение функционального программирования, поскольку, вообще говоря, никто не запрещает в этих языках писать функции без побочных эффектов.

Наконец, язык Fortran делает применение функционального программирования невозможным.

Вообще, некоторый язык программирования может находиться с определённой парадигмой в одном из следующих вариантов взаимоотношений:

1. язык **навязывает** примененение парадигмы. Программирование на данном языке без применения данной парадигмы категорически невозможно. Примеры: язык Smalltalk [] и объектно-ориентированное программирование; язык Fortran и присваивания.

2. язык **понуждает** к применению парадигмы. Программирование на данном языке без применения данной парадигмы возможно, но очень неудобно. Примеры: язык Lisp и рекурсия; язык Pascal и присваивания.

3. язык **поощряет** применение парадигмы. Программирование на данном языке без применения данной парадигмы возможно и достаточно удобно, однако при освоении данной парадигмы программист получает вознаграждение в виде резко возрастающего удобства работы. Примеры: язык C++ и механизм исключений; язык Lisp и функции высокого порядка (функционалы).

4. язык **поддерживает** применение парадигмы. Язык включает в себя специальные средства для применения данной парадигмы, расчитанные на программистов, привыкших к её применению, однако допускает и другие, в некоторых случаях более удобные варианты решения аналогичных задач. Примеры: язык C++ и макропроцессирование; язык Lisp и циклы; язык C и рекурсия.

5. язык **допускает** применение парадигмы. Язык не включает никакой специальной поддержки для данной парадигмы, однако при определенных навыках программист все еще может ее применять. Примеры: язык Pascal и функциональное программирование; язык C и обобщенное программирование; язык Pascal и объектно-ориентированное программирование.

6. язык **препятствует** применению парадигмы. Применение данной парадигмы в данном языке теоретически возможно, однако связано с затратами, делающими её применение неоправданным. Примеры: язык C и обработка исключений (возможно с помощью setjump/longjump, но требует серьезных трудозатрат); язык Pascal и виртуальные функции.

7. язык **запрещает** применение парадигмы. В данном языке недостаточно средств для применения данной парадигмы. Примеры: Fortran и функциональное программирование; Hope и императивное программирование; Bourne Shell и объектно-ориентированное программирование.

Следует особо отметить, что определяющим фактором при обсуждении языков программирования и парадигм оказывается не техническая сторона проблемы, а мышление программиста, т.е. психологическая составляющая.

1.4 Проблема универсального языка

Многообразие языков программирования естественным образом приводит к возникновению вопроса о возможности построения *универсального* языка, т.е. такого языка программирования, который подходил бы для реализации любого проекта, причем подходил бы настолько хорошо, чтобы свести на нет преимущества любых других языков.

Иначе говоря, вопрос состоит в том, можно ли создать такой язык программирования, чтобы для программиста, знающего этот язык и любое количество других языков программирования, при решении любой возникающей программистской задачи выбор языка программирования был бы очевиден и всегда оказывался в пользу универсального языка.

1.4.1 Требования к универсальному языку

Сформулируем коротко требования, без выполнения которых язык заведомо не может претендовать на статус универсального.

- Язык должен быть полностью компилируемым. Желательно, чтобы минимальный возможный для данного языка объем библиотеки времени выполнения был строго равен нулю (именно так обстоят дела для ассемблеров).
- Язык должен быть пригоден для низкоуровневого программирования. Это означает, что
  - не должно быть таких возможностей базового вычислителя (аппаратного обеспечения), которые программа на данном языке не могла бы использовать и
  - ядро языка должно включать в себя только такие средства, реализация которых очевидна и не вызывает сомнений; более сложные средства, допускающие разнообразные способы реализации, должны быть вынесены в библиотеку с тем, чтобы позволить программисту использовать собственную реализацию.
- Язык должен как минимум **допускать** (см. §1.3.6) применение всех парадигм программирования, известных на момент создания языка; с другой стороны, язык не должен по возможности **навязывать** использование тех или иных парадигм.

- Язык должен включать средства генерации абстракций достаточно высокого уровня, чтобы удовлетворить запросы программистов-практиков (в идеале должна быть предоставлена возможность генерации абстракций сколь угодно высокого уровня).

Поясним сказанное. Интерпретируемое исполнение недопустимо при программировании микроконтроллеров, в системах реального времени и некоторых других областях. Если язык непригоден к низкоуровневому программированию, нам приходится исключить его из рассмотрения при реализации таких проектов, как ядра операционных систем, системы реального времени и т.п. Если язык включает средства, допускающие различные реализации, это практически заведомо означает, что в том или ином конкретном проекте реализация, предложенная разработчиками компилятора, окажется непригодной.

В случае, если язык не допускает применение какой-либо парадигмы программирования, возможно, что именно эта парадигма будет признана наиболее удобной при реализации конкретной задачи и язык окажется отвергнут.

Наконец, при ограниченных возможностях генерации абстракций высокого уровня реализация той или иной задачи на данном языке может оказаться заведомо многократно более трудоемкой, нежели на языке более высокого уровня.

Отметим, что в настоящее время автору не известен ни один язык, удовлетворяющий всем перечисленным требованиям одновременно. Из известных языков ближе всех к свойству универсальности подошел язык C++, однако включение в этот язык встроенных средств идентификации типов во время исполнения (RTTI, runtime type identification), множественного наследования в общем виде, а также сложных и неочевидных по способу реализации средств обработки исключительных ситуаций сделали этот язык слишком высокоуровневым. Довершило дело введение в стандарт языка библиотеки шаблонных классов STL, которая современными программистами зачастую воспринимается как часть языка (хотя и не является таковой). В итоге разработчики операционных систем от языка C++ отвернулись; в современном сообществе преобладает восприятие C++ как одного из многих языков высокого уровня.

Глава 2

Метод непосредственной интеграции

2.1 Описание метода

Метод непосредственной интеграции впервые предложен в статье [1] в качестве подхода к интеграции разнородных языковых изобразительных средств в одном проекте. Методу также посвящены работы [2], [3], [5], [7] и [8].

В основе метода непосредственной интеграции лежит моделирование средствами базового языка одновременно вычислительной модели (семантики) и изобразительных конструкций (синтаксиса) альтернативного языка. При этом синтаксис альтернативного языка моделируется близкими по визуальному восприятию арифметическими выражениями, записываемыми в соответствии с правилами языка базового. Это достигается введением объектов, соответствующих понятиям альтернативного языка и переопределением символов инфиксных операций для таких объектов; ясно, что для применения метода необходимо наличие в базовом языке соответствующих возможностей.

Поясним сказанное.

2.1.1 Язык Lisp как алгебра

В языке Lisp как программа, так и данные представляются в виде так называемых S-выражений. S-выражение может быть либо атомарно (числовая константа, строковая константа или символ, а также некоторые другие типы значений, например, файл, функция и т.п.), либо представлять собой так называемую точечную пару - выражение вида `(A . B)`, где `A` и `B` в свою очередь являются S-выражениями. Конструкция вида `(A . (B . (C . NIL)))` называется списком и записывается сокращенно как `(A B C)`. `NIL` — это специальный символ для обозначения пустого списка. Запись `()` является синонимом записи `NIL`. Конструкция вида `(A . (B . (C . D)))` также допустима, называется точечным списком и записывается как `(A B C . D)`. Несмотря на то, что возможные значения S-выражений различаются по типам, переменные и формальные параметры функций описываются без указания типов и могут принимать значение S-выражения любого типа. Говорят, что Lisp является языком типизированных данных и нетипизированных имен.

Универсальная сущность S-выражения позволяет говорить о языке Lisp как об *алгебре S-выражений*. Множеством объектов такой алгебры является простран-

ство S-выражений. Основными операциями являются операции композиции (создания списка из отдельных элементов), декомпозиции (выделение отдельных элементов из списка) и *вычисления S-выражения*. Последняя операция представляет интерпретатор языка Lisp и представляет собой отображение множества S-выражений на себя.

Следует заметить, что наличие операции вычисления S-выражения делает ненужным отдельное рассмотрение операций декомпозиции, так как для этого в языке Lisp существуют соответствующие функции. В то же время композицию стоит, по-видимому, рассматривать как отдельный класс операций, поскольку для обращения к функциям в Lisp'е необходимо сначала построить список, являющийся вызовом функции, и лишь затем применить к нему операцию вычисления.

Предложенное представление языка Lisp как алгебры является, бесспорно, упрощенным, так как, например, не учитывает побочных эффектов вычислений; однако для целей настоящего изложения такое рассмотрение приемлемо и, как это станет ясно из дальнейшего текста, весьма удобно.

2.1.2 Алгебра S-выражений как предметная область библиотеки классов

Чтобы представить конструкции языка Lisp средствами C++, достаточно описать класс, способный хранить (инкапсулировать) S-выражение любого поддерживаемого типа и дать набор операций для построения точечных пар и списков из атомарных S-выражений. Используя свойство полиморфности объектов, можно построить иерархию классов с общим предком для хранения различных типов S-выражений, что позволит вводить новые типы S-выражений.

В результате мы получим запись лисповских программ и данных на C++. После компиляции текста транслятором C++ в сегменте данных основной программы появятся конструкции Lisp'а, уже переведенные во внутреннее представление. Чтобы заставить эти конструкции «ожить», необходимо описать функцию (возможно, метод того же класса), производящую вычисление значений S-выражений по правилам языка Lisp. Таким образом, с помощью библиотеки классов, не изменяя собственно язык C++, можно дать возможность программисту использовать парадигматику языка Lisp в программе на C++.

2.1.3 Средства конструирования списков

При реализации алгебры языка Lisp необходимо предоставить пользователю адекватные средства конструирования лисповских списков.

Авторы монографии [39] предлагают строить списки из точечных пар с помощью явного вызова функции `cons`, порождающей точечную пару из двух своих параметров. Такая форма записи, позволяя строить произвольные лисповские списки, при этом требует их записи в форме, совершенно непохожей на запись в Lispe. Например, список `(1 2 3)` предлагается записать в форме

```
cons(new l_int(1), cons(new l_int(2),
     cons(new l_int(3), &nil)));
```

где `l_int` – имя класса для представления лисповских целых констант, `nil` – объект, используемый для представления пустого списка, `cons` – имя функции, создающей новый объект типа «точечная пара».

Для практического применения такое представление неприемлемо из-за низкой наглядности. В самом деле, программист, умеющий писать на Lisp'е, вряд ли сможет эффективно работать вместо привычных списков со столь громоздкими конструкциями.

Ключом к решению проблемы является имеющаяся в языке C++ возможность переопределения стандартных операций. Допустим, мы описали некоторый класс (назовем его `SExpr`), инкапсулирующий любое S-выражение и снабдили его конструкторами для порождения S-выражений типов целая константа, строка и точечная пара[1]:

```
class SExpr {
      . . .
  public:
      . . .
    SExpr(int i);           // для целых
    SExpr(const char *);    // для строк
    SExpr(const SExpr &car, const SExpr &cdr);
                            // для точечных пар
      . . .
};
```

Предположим также, что один из экземпляров класса `SExpr` описывает пустой список.

Наибольшей наглядности при представлении списков можно достичь путем переопределения для класса `SExpr` операции «запятая» (см. [38, стр. 254]). Операцию можно перегрузить таким образом, чтобы, например, лисповский список

```
("The Beatles" "Let It Be" 1969)
```

мог быть представлен[2] в виде

```
(SExpr("The Beatles"), SExpr("Let It Be"), SExpr(1969))
```

или, благодаря автоматическим преобразованиям в C++, более компактно:

```
(SExpr("The Beatles"), "Let It Be", 1969)
```

Преобразования первого элемента списка к типу `SExpr` достаточно для того, чтобы компилятор, обрабатывая всю конструкцию слева направо, применял операцию «запятая», определенную для класса `SExpr`, автоматически преобразуя каждый

[1] Данный пример условен и приводится с целью иллюстрации. Реальная библиотека классов будет рассмотрена в следующих главах

[2] Чтобы это стало возможным, оператор "запятая", будучи примененным к двум операндам типа SExpr, левый из которых является атомарным S-выражением, должен построить список из двух элементов и вернуть ссылку на него, а будучи примененным к двум операндам, левый из которых является списком – добавить к этому уже существующему списку новый элемент (свой правый операнд) и, опять таки, вернуть ссылку на него.

следующий операнд к типу `SExpr` через соответствующий конструктор. Но и такая сравнительно компактная конструкция выглядит неудобочитаемой из-за наличия лишних скобок в вызове конструктора `SExpr`.

Кроме того, возникают сложности, если первым элементом списка будет, в свою очередь, список, например, `((25 36) 49)`. В отличие от Lisp'а, в C++ скобки используются для явного указания последовательности операций и не несут иной смысловой нагрузки, так что конструкция `((SExpr(25),36),49)` будет в точности эквивалентна конструкции `(SExpr(25),36,49)`. Можно, конечно, и здесь написать, например, `(SExpr((SExpr(25),36)),49)`, но такая конструкция оказывается неприемлемой для восприятия человеком.

Наконец, совершенно непонятно, как сконструировать список из одного элемента. В Lisp'е `25` и `(25)` - это две разные конструкции. Введенных средств, очевидно, недостаточно для передачи этого различия.

Выйти из положения позволяет введение отдельной унарной операции, превращающей любое S-выражение в список из одного элемента. Здесь необходимо отметить, что, поскольку из соображений наглядности желательно иметь возможность строить такие одноэлементные списки из числовых и строковых констант языка C++, переопределением одной из унарных операций мы проблему не решим.

Снять проблему можно, описав еще один вспомогательный класс, в котором переопределяется одна из бинарных операций (например, `|`). Описав переменную-объект этого класса с коротким именем (например, `L`), мы сможем представлять лисповские списки достаточно наглядно:

```
(L| "Here I go", 25)        //   ("Here I go" 25)
(L| (L| 25, 36), 49)        //   ((25 36) 49)
(L| (L| 5, 35), (L| 6, 36)) //   ((5 25) (6 36))
```

Аналогичным образом могут быть введены операции для конструирования точечных пар и точечных списков, для представления лисповского символа «апостроф» и т.п.

2.2 Библиотека InteLib и роль языка C++

Практическая применимость метода непосредственной интеграции иллюстрируется библиотекой InteLib. Эта библиотека предоставляет набор классов языка C++, моделирующий усечённый диалект языка Lisp, названный InteLib Lisp. Этот диалект содержит менее 200 функций и специальных форм языка Lisp, но при этом реализует бóльшую часть привычных для Lisp-программиста возможностей, таких как лексическое и динамическое связывание, замыкания, функции высоких порядков, мапперы и т. п. Как отмечается в докладе [6], именно такой диалект, не перегруженный посторонними для языка Lisp возможностями, оказывается идеален при обучении основам программирования на языке Lisp. Практика показывает, что при использовании языка Lisp в качестве дополнительного в проектах, где основным языком является C++, имеющихся в InteLib Lisp возможностей оказывается достаточно (собственно говоря, возможности в диалект добавлялись по мере появления в них потребности).

В разное время предпринимались попытки реализации в рамках InteLib вычислительных моделей других языков, таких как Рефал, Datalog, Prolog и Haskell. До

уровня практической применимости эти экспериментальные реализации доведены в силу различных причин не были, однако в ходе экспериментов было показано, что конструкции Рефала, Prolog'а и Datalog'а вполне можно адекватно смоделировать выражениями C++. К сожалению, синтаксис языка Haskell оказался для выражений C++ слишком сложен; адекватной (хорошо воспринимаемой человеком) модели найдено не было. Тем не менее, ограничения языка C++, не позволившие смоделировать конструкции Haskell, представляются вполне преодолимыми.

Как можно заметить из вышесказанного, применение метода непосредственной интеграции полностью основано на гибких изобразительных возможностях языка C++.

Если бы, во-первых, язык C++ как таковой допускал низкоуровневое программирование (а это условие, заметим, выполнялось для его ранних версий), и, во-вторых, с помощью вышеописанного метода можно было импортировать в C++ вычислительные модели альтернативных языков, представляющих все известные основные парадигмы программирования, можно было бы считать, что задача построения универсального языка программирования решена.

К сожалению, как уже отмечалось выше, язык C++ в его современном виде не может предентовать на принадлежность к языкам низкого уровня и, таким образом, в качестве универсального рассматриваться не может.

Кроме того, язык C++ не был предназначен для использования способом, эксплуатируемым в методе непосредственной интеграции; то, что на его основе удалось построить вышеописанные инструменты — результат удачной случайности. Так, в книге [38] создатель языка C++ Бьерн Страуструп отмечает, что не может себе представить ситуацию, в которой кому-либо потребуется перегружать операцию «запятая».

При реализации на C++ инструментов, использующих метод непосредственной интеграции. разработчик сталкивается с целым рядом проблем. Часто не хватает символов унарных операций; приоритеты операций, задаваемые языком C++, оказываются неудобны при использовании для формирования выражений, представляющих конструкции альтернативных языков.

Из сказанного можно сделать один чрезвычайно важный вывод: **если построить язык программирования низкого уровня, имеющий при этом средства для описания абстрактных типов данных и предоставляющий возможность перекрытия символов операций, то при определённом уровне гибкости этой возможности можно написать библиотеки, моделирующие вычислительные модели языков-носителей альтернативных парадигм программирования, в результате чего задача построения универсального языка программирования будет решена**.

Ключевыми здесь являются требования о низкоуровневой сущности гипотетического языка и о высоком уровне гибкости средств построения абстракций высокого уровня.

Глава 3

О новом языке

Попытаемся сформулировать свойства языка, который предполагается создать в качестве универсального.

3.1 Наследие языка C

Прежде всего, сам по себе этот язык необходимо сделать языком низкого уровня, подобно ANSI C. Следует отметить, что даже язык C в той его версии, которая описывается стандартом C99, является языком чрезмерно высокого уровня. Так, например, C99 допускает следующий код:

```
void f(int n) {
    int a[n];
    int b[2*n];
    int c[3*n];
    /* ... */
}
```

Если при отсутствии возможности задания размерностей массивов подобным образом всегда можно было сказать, что любое имя локальной переменной представляет собой (с низкоуровневой точки зрения) ни что иное как константное смещение относительно базы стекового фрейма, то в приведённом выше примере имя `c` — это сложное адресное выражение, зависящее от переменной `n`, и способов реализации такого кода возможно несколько, причём найти среди них наиболее оптимальный не представляется возможным.

Представляется целесообразным во избежание появления лишних проблем сделать новый язык C-подобным, но лишь до определённого предела. В силу некоторых причин, которые будут рассмотрены позднее, совместимость с языком C на уровне синтаксиса невозможно сохранить без серьёзных компромиссов на пути достижения основных задач.

С другой стороны, задача создания стандартной библиотеки является крайне трудоёмкой и бессмысленной, поскольку для языка C она уже решена. В связи с этим представляется целесообразным, убрав в языке некоторые синтаксические несообразности языка C, тем не менее сохранить модель вызовов языка C, а также его систему типов; это позволит воспользоваться функциями стандартной библиотеки языка C.

Одним из недостатков синтаксиса языка C является чрезмерная перегруженность символа «запятая». Этот символ обозначает арифметическую операцию, однако он же используется для разделения формальных параметров в заголовках функций, фактических аргументов в вызовах функций, имён переменных в описаниях, элементов в сложных инициализаторах и констант в описаниях перечислимых типов. Если в каком-либо из перечисленных контекстов символ запятой требуется использовать для обозначения операции, приходится использовать лишние скобки.

Заметим, что во всех случаях кроме описания переменных запятую можно заменить на точку с запятой, сняв, таким образом, проблему. Что касается описаний вида

```
int a, b, c;
```

то их представляется целесообразным попросту запретить. Заметим, что действующий в языке C совершенно аналогичный запрет на описание нескольких формальных параметров одного типа в заголовке функции никому не мешает. В то же время, требование вместо вышеприведённого описания использовать

```
int a;
int b;
int c;
```

позволяет устранить ещё одну проблему, которую сторонники языка C попросту не замечают, но которая часто возникает у начинающих. Допустим, требуется описать два указателя на `int`. В языке C это можно сделать, например, так:

```
int *p, *q;
```

Для человека, плохо знакомого со спецификой языка C, такая конструкция выглядит нелогично, поскольку символ `*` имеет в данном случае отношение к типу, а не к переменной. Начинающие часто предпочитают записывать объявление указателя, ставя пробел после символа `*`, а не до него, как опытные программисты на C:

```
int* p;
```

Отметим, что язык C вполне допускает такое описание. Усложнив его, получим

```
int* p, q;
```

Для начинающего программиста естественно восприятие, при котором переменная `q` должна иметь тип «указатель на `int`», тогда как на самом деле эта переменная в данном случае оказывается типа `int`. Запрет описания нескольких переменных одного типа без указания типа решает эту проблему.

3.2 Язык и библиотеки

Одним из важнейших условий, необходимых для низкоуровнего программирования, является четкое отделение библиотеки (сколь угодно стандартной) от собственно языка. В частности, компилятор низкоуровневого языка не вправе ничего знать об именах (функций, переменных и т.п.), описываемых в библиотеке. В противном случае в ситуациях, когда использование (стандартной) библиотеки по тем или иным причинам нецелесообразно, язык и его компилятор оказываются неполноценны. Между тем, такие ситуации возникают гораздо чаще, чем можно подумать: стандартная библиотека того же языка C, несмотря на её логичность и высокое качество проектирования, тем не менее не используется при написании ядер операционных систем, при программировании всевозможных микроконтроллеров и т.п.

Отметим, что принцип разграничения языка и библиотеки часто нарушается даже для языка C: в частности, пресловутый стандарт C99 указывает, что операция `sizeof`, встроенная в язык, должна возвращать значение типа `size_t`, при том что этот тип в язык не встроен и должен описываться в библиотеке.

В качестве второго условия следует назвать следующее правило: всё, что *может быть* вытеснено из языка в библиотеку, *должно быть* вытеснено в библиотеку. В крайнем случае, если в языке необходим некий механизм, то лучше предусмотреть в языке не сам этот механизм, а средства, позволяющие его описать. Так, в языках высокого уровня обычно разрешается «складывать» (конкатенировать) строки с помощью операции `+` (это так, например, для языков Pascal, Java и т.п.). В этом плане язык C++ следует признать более удачным, т.к. он, не предоставляя подобных средств в самом языке, при этом позволяет переопределить операцию `+` для произвольных пользовательских типов, что, в свою очередь, позволяет описывать строки, которые можно «складывать», но не только их: с неменьшим успехом операция `+` используется для сложения комплексных чисел, матриц или, например, полиномов, если таковые описаны в виде классов.

Третье условие касается скорее не самого разрабатываемого языка, а предполагаемой политики стандартизации. Как показывает пример языка C++, стандартизация библиотеки в составе стандарта языка может нанести непоправимый урон культуре программирования на этом языке, а также сообществу программистов, использующих данный язык. Присвоение библиотеке STL статуса составной части стандарта C++ поставило другие библиотеки аналогичного назначения в негативные условия, и, кроме того, для подавляющего большинства программистов, использующих C++, введение STL в стандарт послужило сигналом к тому, что при работе на языке C++ следует **всегда** использовать STL; между тем, использование STL резко снижает читаемость кода, многократно усложняет отладку и сопровождение программ, давая при этом весьма сомнительный выигрыш на стадии кодирования. Обучение начинающих программистов с использованием STL приводит к тому, что студенты, завершившие обучение, попросту не понимают принципиальных отличий C++ от других языков и не представляют программирование на C++ иначе как с использованием STL.

Вместе с тем, стандартизация интерфейсов библиотек имеет ряд несомненных достоинств, прежде всего — повышение переносимости программ.

В связи с этим представляется целесообразным проводить стандартизацию биб-

лиотек **без привязки к стандарту языка**, не давая при этом никаким библиотекам статуса «стандартной библиотеки языка». Это позволит, с одной стороны, использовать положительные стороны стандартизации интерфейсов библиотек, и, с другой стороны, избежать катастрофических последствий стандартизации одной конкретной библиотеки.

Вместе с языком можно (хотя и не обязательно) стандартизовать те и только те элементы библиотеки, которые не могут быть в силу тех или иных причин переносимым образом реализованы средствами самого языка. Примером таких элементов является интерфейс для доступа к системным вызовам операционной системы.

3.3 Синтаксическая гибкость

3.3.1 Арифметические выражения как основа синтаксической гибкости

Чтобы сделать возможным библиотечное моделирование альтернативных вычислительных моделей (парадигм), язык должен обладать определённой синтаксической гибкостью. В то же время необходимо сохранять эту гибкость в определённых рамках: попытки создания языков программирования с полностью программируемым синтаксисом в истории известны и к положительным результатам не приводили.

Как показывает пример проекта InteLib, в действительности для моделирования изобразительных возможностей многих альтернативных языков программирования достаточно выразительной мощности арифметических выражений, при условии, что в базовом языке предусмотрен достаточно широкий спектр арифметических операций и имеются возможности их переопределения для введённых пользователем типов.

Отметим, что слова «достаточно широкий» в предыдущем абзаце могут быть истолкованы весьма по-разному в зависимости от конкретной предметной области. Так, для моделирования выражений языка Lisp оказалось достаточно операций, имеющихся в языке C++; более сложный язык, такой как Haskell, столь удачно в имеющемся наборе операций представить не удастся.

3.3.2 Операция «пробел»

Коль скоро символам инфиксных операций выделяется столь важная роль, целесообразно будет сделать набор таких операций возможно более широким. В частности, введение «операции пробел» (то есть соглашения, при котором два выражения произвольных типов, стоящие друг за другом и не разделённые знаком операции, считаются операндами бинарной операции, называемой «операция пробел») способно резко повысить выразительную мощность арифметических выражений.

В частности, при наличии операции «пробел» обозначить применение математического оператора к его аргументу можно будет способом, напоминающим традиционный для математики: `D f`.

Приоритет такой операции следует, видимо, сделать ниже, чем приоритеты всех имеющихся в языке арифметических операций. Действительно, выражение `a+b c+d` воспринимается скорее как два выражения сложения, записанные одно за другим, нежели как сумма из трёх элементов, вторым из которых является вызов операции «пробел», в особенности если вместо пробела использовать, скажем, символ перевода строки. То же самое можно сказать и обо всех остальных операциях, исключая разве что операцию «запятая». Вопрос о соотношении приоритетов операций «запятая» и «пробел» оставим пока открытым.

Также открытым оставим и вопрос об ассоциативности операции «пробел», то есть о том, какой из её аргументов вычисляется первым, левый или правый.

3.3.3 Сочетание операции «пробел» с операцией «вызов функции»

Действие «вызов функции» в языках C и C++ является операцией, обозначаемой как выражение, первый операнд которого представляет собой выражение типа «адрес функции» (имя функции является примером такого выражения), а за первым операндом следуют круглые скобки, воспринимаемые как символ операции; внутри круглых скобок перечисляются фактические параметры вызова.

Круглые скобки, таким образом, играют в языке две совершенно различные роли. С одной стороны, они используются для группировки подвыражений в выражениях с целью изменения порядка применения операций. С другой стороны, круглые скобки обозначают операцию вызова функции.

В языках C и C++ это не создаёт никаких проблем, поскольку синтаксически эти ситуации полностью разнесены: если перед круглой скобкой находится законченное выражение, то скобка обозначает вызов функции, тогда как если перед скобкой никакого выражения нет либо выражение не закончено и необходим ещё один операнд (то есть скобка находится в позиции, где ожидается выражение), то скобка воспринимается как группирующий символ.

При введении в язык операции «пробел» ситуация несколько изменяется. Для случая вызова функции от нуля аргументов проблем не возникает, поскольку при использовании скобок в качестве группирующих символов закрывающая скобка не может оказаться сразу за открывающей. При условии использования для разделения аргументов символа “`;`”, а не запятой (вообще говоря, попросту символа, не являющегося изображением какой-либо операции), проблем не возникает также и в случае вызова функции от двух и более аргументов (вызов отличается от группировки по наличию символа, разделяющего аргументы).

Что касается случая вызова функции от ровно одного параметра, то выражение вида `a(b)` оказывается имеющим два различных трактования: как операция вызова функции `a` от аргумента `b`, либо как вызов операции «пробел» для аргументов `a` и `b` (в этом случае аргумент `b` представляется заключённым в круглые скобки с целью группировки, что вполне имеет смысл, скажем, если `b` представляет собой, в свою очередь, выражение, с операцией, имеющей более низкий приоритет, чем вызов функции).

Возможно выбрать одно из этих толкований на основе контекстных условий и дополнительных соглашений. Подобно этому в языке C++ инициализация контруктором по умолчанию, применённая при описании переменной, оказалась

бы неотличима от описания функции от нуля аргументов, если бы только для этого применялись общие синтаксические правила:

```
A a(25, 26);  // объект класса A,
              // конструктор от двух аргументов
A a2(77);     // объект класса A,
              // конструктор от одного аргумента
A a3();       // прототип функции от 0 аргументов,
              // которая возвращает объект класса A
A a4;         // объект класса A, использован
              // конструктор от нуля аргументов
```

Заметим, что описание отдельно стоящей переменной — это единственный случай, когда конструктор от нуля аргументов вызывается без круглых скобок. Так, в выражении

```
( A(27, 44) + A() )
```

имеет место создание двух анонимных объектов класса `A`, первый из которых создаётся конструктором от двух аргументов, второй — конструктором от нуля аргументов. Заметим, что скобки в данном случае на месте, а выражение

```
( A(27, 44) + A )
```

является синтаксической ошибкой.

Следует отметить, что решение, применённое в C++, нарушает концептуальную целостность синтаксиса языка (т.к. вводит особый случай), обладает определённой неочевидностью и затрудняет восприятие синтаксиса, в особенности для начинающих программистов. В связи с этим для решения проблемы неоднозначности между вызовом функции от одного аргумента и вызовом операции «пробел» предлагается несколько иное решение, основанное на общем правиле без особых случаев и обладающее дополнительными полезными свойствами.

3.3.4 Кортежи

Под кортежем будем понимать синтаксический конструкт языка, состоящий из нескольких выражений произвольных типов, разделённых символом «точка с запятой» и заключенных в круглые скобки, например:

```
(25; "a string"; x+y)
```

Выделим два особых случая, а именно, кортеж, состоящий из нуля элементов и обозначаемый `()`, а также кортеж из одного элемента, который положим семантически эквивалентным самому этому элементу (таким образом, произвольное выражение становится частным случаем кортежа, а именно, кортежем из одного элемента). Введём соглашение, что при записи кортежа из одного элемента скобки можно опустить; таким образом, например, выражения `25` и `(25)` останутся эквивалентными, как и до введения понятия кортежа.

Будем считать, что кортеж с указанием количества и типов параметров является, в свою очередь, типом данных с точки зрения профилей функций (то

есть может, например, выступать в качестве параметра функции), но при этом не является, вообще говоря, структурой данных, то есть не может быть присвоен переменной (как не может и описываться переменная типа «кортеж»). Вызов функции от кортежа из n элементов физически реализуется как вызов функции от n параметров.

Логично разрешить присваивание кортежа выражений кортежу переменных, например:

```
int x;
const char *p;
float f;
(x; p; f) = (25, "string", 3.7);
```

Это позволит пользоваться кортежами имён формальных параметров при описании и вызове функций от кортежей. Например:

```
void f((int a ; int b) ; int c) { /* ... */ }
/* ... */
f((2 ; 3) ; 4);
```

Наконец, определим понятие операции вызова функции как частный случай операции «пробел» от имени функции (или, если угодно, указателя на функцию) и кортежа параметров. Как следствие, вызов функции от одного параметра станет возможно записать без скобок, например, запись `sin(x)` окажется семантически эквивалентна записи `sin x`.

Подчеркнём ещё раз, что вызовы `f(1;2;3;4)` и `f((1;2);(3;4))` можно различать с точки зрения профилей функций (т.е. компилятор должен вызывать при этом разные функции), но с точки зрения реализации ничем, кроме адреса вызываемой функции, эти два вызова различаться не должны.

3.4 Введение новых символов инфиксных операций

Как показал пример библиотеки InteLib, имеющихся в языке символов инфиксных операций может и не хватить, либо может ощущаться их недостаток.

Известны языки программирования, в которых допускается введение новых (не предусмотренных в языке) символов инфиксных операций; так, это возможно в языке Prolog.

Вместе с тем, введение операции «пробел» даёт возможность моделировать новые символы инфиксных операций, вводя их как идентификаторы объектов, для которых переопределена операция «пробел»: так, например, выражение `a in b` можно рассматривать не как операцию `in` от аргументов `a` и `b`, а как два вызова операции «пробел»; в этом случае слово `in` должно быть именем переменной, имеющей некоторый пользовательский тип. В зависимости от ассоциативности операции «пробел» это будет, соответственно, выражение

```
operator space(a, operator space(in, b))
```

или

```
operator space(operator space(a, in), b)
```

Вместе с тем, такой способ моделирования новых инфиксных операций обладает сравнительно низкой гибкостью, т.к. ограничен фиксированным приоритетом и направлением ассоциативности операции «пробел». Кроме того, в качестве символов таких операций можно будет использовать исключительно идентификаторы языка.

Как уже говорилось, к гибкости синтаксиса языка следует относится с определённой осторожностью, поэтому следует оставить открытым вопрос о целесообразности введения в язык возможности определения новых символов операций, подобной имеющейся в языке Prolog. Вернуться к этому вопросу можно будет после того, как язык будет реализован и будет получен определённый опыт работы с ним.

3.5 Наследование

Наследование как таковое является важнейшим средством создания высокоуровневых абстракций, так что необходимость наличия наследования в языке как таковая несомненна. Вместе с тем, вопросы, связанные с множественным наследованием, оказываются достаточно дискуссионными; кроме того, механизм виртуальных функций оказывается достаточно нетривиален, чтобы его нельзя было считать средством низкого уровня.

3.5.1 Множественное наследование

Реализация множественного наследования в общем виде, подобно тому, как это сделано в языке C++, несмотря на кажущуюся простоту, влечёт массу нетривиальных проблем.

Действительно, пока все базовые классы неполиморфны (то есть не содержат виртуальных функций) и не являются сами чьими-то наследниками, реализация множественного наследования остаётся достаточно простой. Единственная привносимая проблема — необходимость пересчёта численного значения указателя или ссылки при преобразованиях от типа «потомок» к типу «предок» (и обратно), если соответствующий предок не является первым в списке базовых классов.

Появление у базовых классов общих предков влечёт появление понятия виртуального класса и множества сомнительных ситуаций, как в случае, если один и тот же класс является виртуальным предком части базовых классов и невиртуальным предком другой части базовых классов.

Появление в двух и более базовых классах таблиц виртуальных функций приводит к резкому усложнению реализации самого механизма виртуальных функций: для каждой виртуальной функции теперь, помимо адреса, требуются ещё, как минимум, смещения для пересчёта указателя `this` и положения указателя на таблицу виртуальных методов, так что вместо одного поля (адреса функции) строка таблицы виртуальных методов оказывается состоящей из трёх полей.

В этой связи представляется разумным компромисс, принятый в языке Java: среди базовых классов может быть не более одного класса, не являющегося *интерфейсом* (то есть классом, состоящим исключительно из чисто виртуальных функций).

В то же время статические преобразования указателей сами по себе не сложны в реализации и вполне укладываются в представления о низкоуровневом программировании, так что (при условии успешного вытеснения механизмов виртуализации в библиотеку) в язык без ущерба его низкоуровневости можно внести и более общие виды множественного наследования. Конечно, при этом следует постулировать, что наследование от нескольких классов, имеющих общий базовый класс, *всегда* приводит к наличию в объекте соответствующего количества экземпляров этого базового класса, поскольку имеющийся в языке C++ механизм виртуальных базовых классов заведомо чрезмерно сложен для низкоуровневого языка.

3.5.2 Возможность библиотечной поддержки механизма виртуальных функций; потребность в гибких макросредствах

Как отмечалось выше, механизм виртуальных функций (в особенности в ситуации множественного наследования) оказывается чрезмерно сложен, чтобы считаться средством низкого уровня.

С другой стороны, библиотечная реализация виртуальных вызовов практически неизбежно повлечет за собой необходимость достаточно развитых макросредств.

Само по себе это, возможно, и не столь плохо. Дело в том, что большинство возражений против применения макропроцессора, высказываемых относительно языков C и C++, основаны на том, что макроимена не подчиняются обычным для языка соглашениям об именах. Вместе с тем, язык C++ вводит т.н. шаблоны, которые отличаются от макросов практически лишь в том, что являются полноценной частью языка, подчиняются всем законам локализации имён и поддерживают проверку типов параметров (ограничение на возможные типы параметров, присутствующее в C++, выглядит скорее реализаторским, чем концептуальным).

Можно привести и другие примеры макросистем, никоим образом не нарушающих концептуальную целостность языка и не создающих никаких проблем. Такова, например, система макросредств в языке Common Lisp.

Отметим еще один момент, непосредственно связанный с виртуальными вызовами. Вообще говоря, виртуальный вызов представляет собой действие совершенно иное, нежели обыкновенный вызов. Возможно, имеет смысл зарезервировать для этого действия отдельные синтаксические соглашения (например, ввести тот или иной символ операции).

При наличии в языке макроподобных механизмов, допускающих перегрузку по типам параметров, возможно также ввести и описание инфиксных операций не (или не только) в виде функций, как это сделано в C++, но и в виде макросов (так, в C++ пришлось вводить весьма нетривиальные правила переопределения операции -> как одноместной; определение её как макроса окажется заведомо проще для восприятия, поскольку такой макрос можно сделать двуместным). Введя

возможность описания инфиксной операции в виде макроса, мы получаем возможность реализовать в виде макросов всё связанное с механизмом виртуальных вызовов, вытеснив, таким образом, эти (заведомо высокоуровневые) средства в библиотеку.

3.6 Обработка исключений и явное управление стеком

Обработка исключительных ситуаций представляет собой крайне важный механизм, использование которого повышает читабельность и надёжность программного кода и существенно снижает трудоёмкость программирования. В особенности это средство полезно в сочетании с автоматическим вызовом деструкторов локальных объектов, как это сделано в языке C++.

С другой стороны, обработка исключений в том виде, в котором она присутствует в современных языках программирования (в том числе и в C++), является средством заведомо высокого уровня. Реализация исключений неочевидна и непрозрачна.

Проблема вытеснения механизмов обработки исключительных ситуаций из ядра языка в библиотеку достаточно интересна. Ключ к решению видится в выработке удобного и логичного интерфейса к явной манипуляции стековыми фреймами; отметим, что это средство должно быть частью языка, т.к. должно позволять, например, при принудительном уничтожении стекового фрейма отработать деструкторы уже сконструированных объектов. Таким образом, механизма, подобного библиотечным функциям `setjmp()` и `longjmp()`, для этого не достаточно.

Вполне возможно, что соответствующее средство языка может быть похожим на «альтернативное тело», подобное блокам `catch` в языке C++; вместо `try`-блока следует, по-видимому, предусмотреть конструкцию, семантика которой сводится к «пометке фрейма» (для такой пометки можно использовать, например, нетипизированный указатель), и операцию «раскрутки стека на один фрейм», которая возвращала бы для помеченных фреймов значение метки, а для фреймов, не помеченных меткой — нулевой указатель.

Наличие таких средств в языке позволит предоставить программисту выбор из нескольких библиотек, обслуживающих обработку исключительных ситуаций и предоставляющих сервис различного уровня сложности. Так, не во всяком проекте требуется возможность использования *произвольного* объекта в качестве «носителя» исключительной ситуации; в некоторых случаях можно обойтись простым кодом ошибки либо фиксированной структурой данных для хранения информации об ошибке, получив при этом выигрыш в эффективности.

Следует отметить, что автору известны случаи сознательного отказа от работы с исключениями, мотивируемые чрезмерной сложностью этого механизма в языке C++. Наличие простой и прозрачной основы для такого механизма потенциально способно исправить ситуацию.

Средства явного управления стеком могут быть полезны также и для других целей, в частности — для организации локальных переменных динамически определяемого размера, что позволит, в частности, реализовать аналог функции `alloca()` без применения ассемблерных вставок. Кроме того, известны, напри-

мер, попытки реализации продолжений (continuations) в языке C, использующие недокументированные особенности функций `setjmp()` и `longjmp()` для манипуляции стеком; наличие средств явного управления стеком устранит необходимость использования таких сомнительных методов.

Глава 4

Заключение

Традиционное восприятие роли языка программирования в вычислительной системе, состоящей из аппаратуры, операционной системы и систем программирования, можно описать приблизительно следующим образом. Имеется некий конгломерат аппаратуры и операционной системы (либо даже попросту виртуальная машина), именуемый обычно *платформой* или *операционной средой*; в рамках этой среды имеется несколько *систем программирования*, каждая из которых построена вокруг того или иного языка программирования или даже некоторой конкретной его реализации. Среди имеющихся (доступных для данной платформы) систем программирования обычно для конкретного проекта выбирается какая-то одна (реже — несколько), причем выбор может быть обусловлен как поставленной задачей, так и личными предпочтениями разработчиков.

Разработка и реализация универсального языка программирования в соответствии с подходом, описываемом в настоящей работе, может несколько изменить представление о взаимоотношениях системы программирования и операционной платформы. Если сформулированные цели будут достигнуты, для заданной платформы может оказаться разумным поддерживать **один** (универсальный) язык программирования, снабженный при этом широкой коллекцией библиотек. Именно многообразие этих библиотек и займёт в этом случае нишу нынешнего многообразия языков программирования; от особенностей задачи и личных предпочтений программистов будет тогда зависеть не выбор языка, а выбор набора библиотек.

Необходимо признать, что две программы, написанные на одном и том же (универсальном) языке, но с использованием принципиально различных библиотек, могут различаться по стилю практически столь же сильно, как сейчас различаются программы, написанные на разных языках программирования. Тем не менее, использование именно одного языка с множеством библиотек вместо традиционного множества языков имеет как минимум одно важное достоинство: трудности интеграции между собой фрагментов программы, написанных в различном стиле, окажутся раз и навсегда сняты именно в силу использования одного языка.

Литература

[1] E. Bolshakova and A. Stolyarov. Building functional techniques into an object-oriented system. In *Knowledge-Based Software Engineering. Proceedings of the 4th JCKBSE*, volume 62 of *Frontiers in Artificial Intelligence and Applications*, pages 101–106, Brno, Czech Republic, September 2000. IOS Press, Amsterdam.

[2] А. В. Столяров. Интеграция изобразительных средств альтернативных языков программирования в проекты на C++. Рукопись депонирована в ВИНИТИ 06.11.2001, №2319-В2001, Москва, 2001.

[3] И. Г. Головин, А. В. Столяров. Объектно-ориентированный подход к мультипарадигмальному программированию. *Вестник МГУ*, сер. 15 (ВМиК), №1, 2002 г., стр. 46–50.

[4] А. В. Столяров. Расширенный функциональный аналог языка Рефал для мультипарадигмального программирования. // Л. Н. Королев, ред., *Программные системы и инструменты. Тематический сборник*, том 2, стр. 184–195. Издательский отдел факультета ВМиК МГУ, Москва, 2001.

[5] А. Столяров. Интеграция разнородных языковых механизмов в рамках одного языка программирования. *Диссертация на соискание степени канд. физ.-мат. наук*, Москва, 2002.

[6] А. Столяров, Е. Большакова, Н. Баева. InteLib Lisp в обучении программированию на Лиспе. // Тезисы докладов конференции «Свободное программное обеспечение в высшей школе», Переславль, 28–29 января 2006 года.

[7] Stolyarov, A. V. A framework of heterogenous dynamic data structures for object-oriented environment: the S-expression model In *Knowledge-Based Software Engineering. Proceedings of the 6th JCKBSE*, vol. 108 of *Frontiers in Artificial Intelligence and Applications*, pages 75–82, Protvino, Russia, August 2004. IOS Press.

[8] А. Столяров. Библиотека InteLib — инструмент мультипарадигмального программирования. // II конференция разработчиков свободных программ «На Протве». Тезисы докладов. Обнинск, 25–27 июля 2005 г., стр. 56–62.

[9] American National Standards Institute. *ANSI X3.135-1992: Information Systems – Database Language – SQL*. American National Standards Institute, 1430 Broadway, New York, NY 10018, USA, 1989.

[10] J. Backus. The history of FORTRAN I, II and III. *ACM SIGPLAN Notices*, 13(8):165–180, Aug 1978.

[11] J. Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, Aug 1978. Русский перевод см. в: Лекции лауреатов премии Тьюринга за первые двадцать лет (1966 - 1985), М.: МИР, 1993.

[12] D. W. Barron. *An introduction to the study of programming languages*. Cambridge University Press, Cambridge, London, New York, Melbourne, 1977. Русский перевод: Д.Баррон, Введение в языки программирования. М.: МИР, 1980.

[13] R. Bird. *Introduction to functional programming using Haskell*. Prentice Hall Press, second edition, 1998.

[14] D. Bobrow and M. Stefik. Perspectives on artificial intelligence programming. *Science*, 231:951, February 1986.

[15] D. G. Bobrow. If Prolog is the answer, what is the question. In *Fifth Generation of Computer Systems*, pages 138–145, Tokyo, Japan, November 1984. Institute for New Generation Computer Technology(ICOT), North-Holland.

[16] G. Booch. *Object-oriented Analyses and Design*. Addison-Wesley, Reading, Massachusets, second edition, 1994.

[17] T. A. Budd. *Multy-Paradigm Programming in LEDA*. Addison-Wesley, Reading, Massachusets, 1995.

[18] R. M. Burstall, D. B. MacQueen, and D. T. Sannella. Hope: an experimental applicative language. In *Conference Record of the 1980 LISP Conference*, pages 136–143, Stanford University, Stanford, California, August 25–27, 1980. ACM Press.

[19] A. Calmerauer, H. Kanoui, and M. van Caneghem. Prolog, bases théoriques et développements actuels. *Technique et Science Informatiques*, 2(4):271–311, 1983. Русский перевод см. в: Логическое программирование. Сборник статей. М.: МИР, 1988.

[20] E. W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, Oct 1972. Русский перевод см. в: Лекции лауреатов премии Тьюринга за первые двадцать лет (1966 - 1985), М.: МИР, 1993.

[21] A. J. Field and P. G. Harrison. *Functional Programming*. Addison-Wesley, Reading, Massachusets, 1988.

[22] R. W. Floyd. The paradigms of programming. *Communications of the ACM*, 22(8):455–460, 1979. Русский перевод см. в: Лекции лауреатов премии Тьюринга за первые двадцать лет (1966 - 1985), М.: МИР, 1993.

[23] L. W. Friedman. *Comparative programming languages: generalizing the programming function*. Prentice Hall, 1991.

[24] S. C. Johnson. Yacc – yet another compiler-compiler. Computer science technical report 32, Bell Laboratories, Murray Hill, NJ, USA, 1975.

[25] В. Ш. Кауфман. *Языки программирования. Концепции и принципы.* Радио и связь, Москва, 1993.

[26] Б. Керниган, Д. Ритчи. *Язык программирования Си* М.: Финансы и статистика, 1992.

[27] T. S. Kuhn. *The structure of scientific revolutions.* University og Chicago Press, Chicago, second, enlarged edition, 1970. Русский перевод: Т. Кун, Структура научных революций. М.: Прогресс, 1997.

[28] M. E. Lesk and E. Schmidt. Lex – a lexical analyser generator. Computer science technical report 39, Bell Laboratories, Murray Hill, NJ, USA, 1975.

[29] J. McCarthy. History of LISP. *ACM SIGPLAN Notices*, 13(8), Aug 1978.

[30] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3(4):184–195, Apr 1960.

[31] J. Robinson. Logic programming - past, present and future. *New Generation Computing*, 1:107–121, 1983. Русский перевод см. в: Логическое программирование. Сборник статей. М.: МИР, 1988.

[32] J. E. Sammett. *Programming Languages: history and fundamentals.* Prentice-Hall, 1969.

[33] B. D. Shriver. Software paradigms. *IEEE Software*, 3(1):2, January 1986.

[34] D. Spinellis, S. Drossoupoulou, and S. Eisenbach. Language and architecture paradigms as object classes: A unified approach towards multiparadigm programming. In J. Gutknecht, editor, *Programming Languages and System Architectures International Conference*, volume 782 of *Lecture Notes in Computer Science*, pages 191–207, Zurich, Switzerland, March 1994. Springer-Verlag.

[35] D. D. Spinellis. *Programming paradigms as object classes: a structuring mechanism for multiparadigm programming.* PhD thesis, University of London, London SW7 2BZ, United Kingdom, February 1994.

[36] G. L. Steele Jr. *Common Lisp the Language.* Digital Press, Burlington MA, second edition, 1990.

[37] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, Massachusets, third edition, 1997.

[38] B. Stroustrup. *The design and evolution of C++.* Addison-Wesley, Reading, Massachusets, 1994. Русский перевод: Бьерн Страуструп, Дизайн и эволюция языка C++, М.: ДМК, 2000.

[39] K. S. Tan, W. Hans Steeb, and Y. Hardy. *Symbolic C++: An introduction to computer algebra using object-oriented programming*. Springer, London, 2000. Русский перевод: Тан К.Ш., Стиб В.-Х., Харди Й. Символьный C++: Введение в компьютерную алгебру с использованием объектно-ориентированного программирования. М.:Мир, 2001.

[40] V. Turchin. *REFAL-5, Programming Guide and Reference Manual*. New England Publishing Co., Holyoke, 1989.

[41] D. A. Turner. Miranda – a non-strict functional language with polymorphic types. In J. P. Jouannaud, editor, *Proceedings of the Conference of Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 1–16, Nancy, France, 1985. Springer-Verlag.

[42] P. Wegner. Concepts and paradigms of object-oriented programming. *OOPS messenger*, 1(1):7–87, August 1990.

[43] K. Jensen and N. Wirth. *Pascal User Manual and Report*. Springer, 1974.

[44] P. Zave. A compositional approach to multiparadigm programming. *IEEE Software*, 6(5):15–25, September 1989.