**Московский государственный университет**
**имени М. В. Ломоносова**

На правах рукописи

**Столяров** Андрей Викторович

# Интеграция разнородных языковых механизмов в рамках одного языка программирования

Специальность 05.13.11 -
математическое и программное обеспечение
вычислительных машин, комплексов и компьютерных сетей

Диссертация на соискание ученой степени
кандидата физико-математических наук

*Защита диссертации состоялась 29 ноября 2002 г. на заседании диссертационного совета Д 501.001.44 в МГУ им. М.В.Ломоносова*

Научный руководитель:
к.ф.-м.н. Головин И. Г.

Москва - 2002

# Оглавление

# Предисловие

Эффективность работы программиста во многом зависит от адекватности используемых изобразительных средств по отношению к решаемой задаче. Так, было бы сложно решать задачу символьного дифференциирования, имея в распоряжении только представление текста как массива символов. С другой стороны, весьма страно было бы использовать для написания интерактивной программы язык, допускающий только «чистые» функции, не имеющие побочных эффектов (хотя это и возможно при наличии ленивых вычислений).

Ограничиваясь при реализации проекта одним конкретным языком, мы, как правило, вынуждены довольствоваться наработанными вокруг этого языка изобразительными приемами, что при решении частных подзадач может оказаться неудобным за неимением в избранном языке адекватных средств. Поэтому у многих программистов часто возникает потребность воспользоваться при решении той или иной частной подзадачи средствами языка программирования, отличного от основного языка проекта.

В то же время попытка использования в проекте нескольких языков может обернуться столь высокими накладными расходами, что преимущества применения альтернативных изобразительных средств окажутся сведены на нет.

В настоящей работе предложено решение задачи предоставления программистам, владеющим альтернативными языками (такими, как Лисп, Пролог, Рефал и т.п.), удобных возможностей применения своих навыков в рамках проектов, реализуемых на индустриальных языках программирования (в частности, на языке Си++). Предложенный метод позволяет избежать большинства проблем, характерных для многоязыковых проектов.

# Введение

## Разнообразие языковых средств и проблема выбора

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 [15] и Лисп [40], предоставили в распоряжение программиста такие высокоуровневые абстракции, как математические формулы, символические выражения, рекурсивные функции [38] 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 [43] описывает более 120 языков программирования. К настоящему времени общее число разработанных языков программирования оценивается несколькими тысячами.

Многие авторы отмечают, что правильный выбор языка программирования является ключевой проблемой, от которой существенно зависит успех любого программистского проекта.

Дейв Баррон (Dave Barron) в книге [16] 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» [24], 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 прекрасно иллюстрирует пример, приведенный Тимоти Баддом в [20]. 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, 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.

# О классификации языков программирования

Очевидно, что попытка рассмотрения всех или почти всех существующих языков с целью выбора наиболее подходящего обречена на провал, т.к. ни один человек не в состоянии изучить все те тысячи созданных человечеством языков не только в достаточной мере, чтобы сделать аргументированный выбор, но и вообще в сколь бы то ни было осмысленной степени.

К счастью для программистов-практиков, 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 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[1], 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 бо́льших усилий для решения конкретной подзадачи, однако в большинстве случаев эти усилия окажутся меньшими, чем те, что мы потратили бы 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, так как многообразие уже наработанных средств практически всегда позволяет найти инструмент, адекватный поставленой задаче.

Существуют различные способы классификации языков программирования. С точки зрения выбора языка для решения той или иной задачи наиболее релевантной оказывается классификация, опирающаяся на основные методики и подходы к описанию алгоритмов (концепции), стимулируемые данным языком [59].

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

# Термин «парадигма» и его применение в программировании

В настоящей работе при описании, сравнении и классификации изобразительных средств языков программирования будет активно употребляться 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 [26] называет в качестве примера парадигмы структурное программирование. В то же время другие авторы (см, например, [47]) предпочитают считать, что структурное программирование парадигмой программирования не является.

В связи с этим представляется необходимым уделить определенное внимание самому термину.

## Происхождение термина

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» [35]. 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 [26] 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.

## 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» [14], 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 Бейсика или Фортрана [15]. Примером процедурного языка могут служить Паскаль [30], Си [34] и другие языки.

Другая часто упоминаемая парадигма - *функциональное программирование* [14, 25] - 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[2].

В качестве примера функционального языка часто называют Лисп [50], однако этот пример неудачен. Ни одна версия Лиспа не была чисто функциональной, хотя, безусловно, программирование

[2]Строго говоря, это не совсем так, поскольку технология ленивых вычислений дает возможность создавать интерактивные программы, оставаясь в рамках чисто функционального программирования [25], однако такие построения слишком сложны для повседневного практического применения.

на Лиспе стимулирует мышление в терминах функций. Тем не менее, в качестве примеров чисто функциональных языков правильней будет назвать Хоуп [21] и Миранду [57].

Здесь следует заметить, что термин «функциональное программирование» является весьма общим. Он покрывает, в числе прочих, такие концепции, как *функции высоких порядков* (функционалы), *функции как объекты данных*, *ленивые вычисления* и т.п. Повидимому, 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[3].

Основанное на исчислении предикатов *логическое программирование* [42] также часто называют одной из парадигм программирования. В логическом программировании (например, на языке Пролог [22]) 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[4]. Декларативная семантика порождает парадигму *декларативного программирования*.

В.Ш.Кауфман в [11] вводит понятие *ситуационного программирования* («модель Маркова-Турчина», язык Рефал [55]), 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*

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

[4]Справедливости ради следует отметить, что декларативная семантика встречается не только в логическом программировании. Примером декларативного языка, не имеющего отношения к логическому программированию, можно считать SQL [13].

*представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.* [19]

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 Си++, называет процедурное программирование, модульное программирование, парадигму пользовательских типов, парадигму абстрактных типов данных, виртуальные функции, объектно-ориентированное программирование и, наконец, обобщенное программирование (классы-шаблоны) [53].

В статье [46] в качестве парадигмы программирования упоминаются даже регулярные выражения и БНФ-грамматики, используемые во входных языках грамматических процессоров lex [36] и yacc [31].

## 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 [47] утверждает: *Слово «парадигма» используется в программировании для обозначения семейства нотаций, разделяющих общий путь описания реализаций программ*[5].

Для сравнения тот же автор приводит определения из других работ. В статье Дэниела Боброва [18] парадигма определяется как *стиль программирования как описания намерений программиста.* Брюс Шрайвер (Bruce Shriver) определяет парадигму как *мо-*

[5]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".

*дель или подход к решению проблемы* [44], Линда Фридман (Linda Friedman) - как *подход к решению проблем программирования* [27, стр. 188]. Памела Зейв (Pamela Zave) дает определение парадигмы как *способа размышления о компьютерных системах*[6] [60].

Питер Вегнер (Peter Wegner) предлагает другой подход к определению термина парадигмы программирования. В работе [59] 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* [20].

## Альтернативные термины

Многие авторы предпочитают использовать термин *стиль программирования* (например, [14, 19, 17]). В работе [17] 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 [53] 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 [11].

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».

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

## Выводы

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. Для нужд дальнейшего обсуждения введем еще одно определение.

***Стилем программирования*** **мы будем называть комплекс парадигм, сложившийся вокруг определенного языка или семейства языков программирования в силу его (их) отличительных особенностей**.

Таким образом, под стилем программирования мы будем понимать весь комплекс понятий и приемов программирования, характерных для какого-то конкретного языка или семейства языков программирования.

В дальнейшем ограничимся рассмотрением следующих пяти стилей программирования:

- Императивно-процедурное программирование (Бейсик, Фортран, Паскаль, Си, Ада);

- Объектно-ориентированное программирование (Смоллток, Эйфель);

- Функциональное программирование (Лисп, Миранда, Хоуп, Хаскел);

- Логическое программирование (Пролог, Дейталог, Логлисп);

- Ситуационное программирование (Рефал).

Данный список не претендует на полноту или законченность, однако для целей дальнейшего изложения достаточен[7].

## Технологическая мотивация мультипарадигмального подхода

Как уже говорилось, парадигмы программирования, в отличие от парадигм в естественной науке (парадигм по Куну), не являются взаимоисключающими. Напротив, умение программиста на практике применять разнообразные парадигмы и целые стили дает возможность выбора наиболее удобных парадигм для каждой конкретной задачи.

В идеале, для каждой частной подзадачи следовало бы применять свой набор парадигм программирования. Для иллюстрации рассмотрим задачу, к которой сводится большинство современных информационных проектов. Допустим, имеется некий банк данных, имеющий сложную организацию - например, реляционная база данных с несколькими сотнями отношений. Необходимо обеспечить взаимодействие пользователя с этой базой данных, по возможности приблизив язык запросов к естественному языку.

Прежде всего необходимо проанализировать введенный пользователем запрос (произвести лексический и синтаксический анализ). Затем необходимо понять смысл пользовательского запроса (анализ семантики), сформулировать и послать собственно запрос к базе данных, после чего вернуть результаты пользователю (возможно, также после некоторой обработки).

Для лексического и синтаксического анализа чрезвычайно удобен язык Рефал, тем более что анализируются сравнительно короткие запросы пользователя и эффективность в данном случае не критична. Затем, когда запрос разобран на лексемы и переведен во внутреннее представление, провести семантический анализ было бы удобнее на Лиспе. Наконец, для формулировки запроса к базе данных как нельзя лучше подходит Пролог или другой язык логического программирования, модифицированный для работы с внешними по отношению к программе отношениями (таблицами базы данных) [23]. Каждое отношение базы данных на уровне пролог-машины может быть представлено как n-арный (по числу атрибутов) предикат,

[7] Исследование вопроса о границах множества стилей программирования, по-видимому, выходит за рамки настоящей работы.

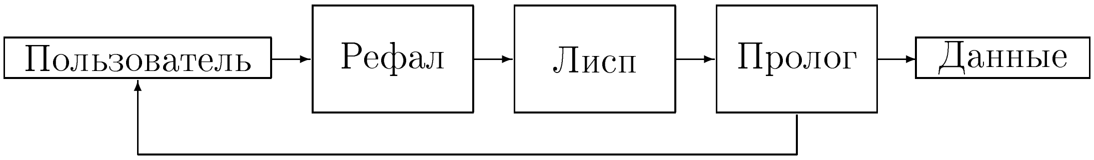


Рис. 1: Упрощенная схема примерной системы

все предложения которого являются фактами (не содержат целей, подлежащих вычислению).

Итак, поток данных было бы удобно организовать, как показано на рис. 1.

Данная диаграмма, однако, не учитывает некоторых деталей, существенных в реальной задаче. Во-первых, базу данных нельзя рассматривать лишь как набор «абстрактно существующих» порций информации. Информация хранится в реальных файлах на реальном диске, вставленном в реальный компьютер. Диски имеют свойство переполняться или портиться, компьютер может давать сбои, необходимо осуществлять резервное копирование, журнализацию, поддерживать механизм гранулированных захватов, если база данных позволяет многопользовательский доступ, и т.п. Все эти так называемые «системные» задачи следует решать на императивном языке, приближенном к языку машины (например, на языке Си).

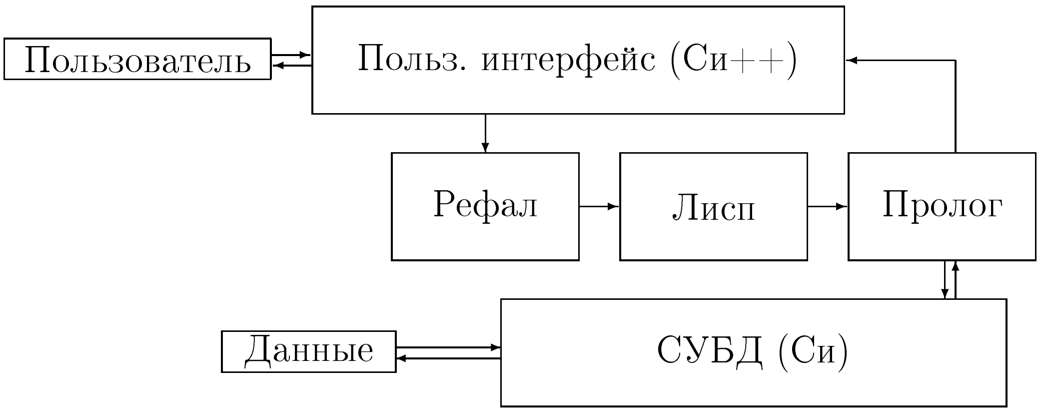


Рис. 2: Реалистическая схема примерной системы

Кроме того, интерфейс пользователя в современных программных продуктах, как правило, существенно сложнее, чем приглашение командной строки. Несмотря на то, что в данном случае запросы, скорее всего, будут вводиться с клавиатуры, это не исключает некоторого интерактивного графического интерфейса, облегчающего работу (например, результаты запросов удобнее видеть в отдельном окне, оснащенном скроллером). Со времен проекта Smalltalk [32] известно, что оконные интерфейсы лучше всего разрабатывать с использованием объектно-ориентированного стиля – к примеру, на Си++. То же касается вообще любых событийно-управляемых программ, к классу которых относятся, например, сервера, способные одновременно обслуживать несколько клиентов, реализованные в рамках одного процесса.

На рис. 2 представлена схема, лучше отвечающая реальности.

В условиях реального проекта реализация подобной схемы практически никогда не производится. Основная причина, как уже сказано, усматривается в трудностях интеграции кода, написанного на языках существенно различной природы.

# Мультипарадигмальный подход и человеческий фактор

Рассматривая перспективы мультипарадигмального программирования, не следует упускать из вида также и навыки, свойственные конкретным программистам. Программист, владеющий тем или иным альтернативным языком программирования, концептуально отличным от базового языка проекта, в некотором роде является носителем альтернативной культуры программирования. Как ясно из вышеизложенного, в ряде случаев это обстоятельство может оказаться полезным для проекта в целом, избавив команду разработчиков от рутинной работы.

Если рассматривать ситуацию в контексте человеческого фактора, вышеописанные проблемы встают под несколько иным углом. Так, в случае, если в команде разработчиков отсутствуют носители альтернативных культур (что, к сожалению, в современных реалиях не редкость), то такая команда не станет использовать возможности альтернативных стилей программирования по причине неосведомленности технических руководителей об этих возможностях.

Таким образом, говоря об использовании разнородных (т.е. пришедших из различных языков программирования) приемов и понятий в рамках одного проекта, следует ориентироваться преимущественно на программистов, одновременно с базовым языком проекта владеющих альтернативными языками[8].

Это означает, что средства, предоставляемые в распоряжение программистов, должны быть приближены по своим выразительным свойствам к оригиналу некоторого существующего языка программирования, играющего роль базиса для импортируемого стиля программирования. В противном случае программисты, имеющие опыт работы на определенном языке, не станут использовать предлагаемые средства из-за сложностей в их освоении.

## Проблема сочетания разнородных языковых механизмов

Ясно, что практически в каждом языке программирования можно обнаружить поддержку различных парадигм. Так, язык Паскаль поддерживает императивное программирование, процедурное программирование, рекурсивное программирование, генерацию пользовательских типов данных и т.д. Кроме того, Паскаль допускает применение техники функционального программирования. Действительно, все, что для этого нужно - это наличие в языке функций, возвращающих значение, и допустимость рекурсивных вызовов. При наличии этих механизмов ничто не мешает нам построить программу или какую-то ее часть в виде набора функций, не имеющих побочного эффекта. Мы будем говорить, что язык Паскаль *поддерживает* парадигмы императивного, процедурного, рекурсивного программирования, а также генерацию пользовательских типов данных, и в то же время *допускает*, но при этом не стимулирует использование функциональной парадигмы.

Некоторые достаточно простые парадигмы могут быть искусственно внесены практически в любой язык независимо от его природы. Так, парадигма разбора строк по регулярным выражениям может быть внесена (и вносится) в язык Си с помощью библиотеки из нескольких функций. Ясно, что никакой проблемы здесь нет, несмотря на то, что парадигма регулярных выражений, являющаяся «родной» для, например, языка Перл [58], изначально никак не

[8]Следует отметить, что при традиционной организации работы с разделением программистов на аналитиков и кодировщиков от программистов-кодировщиков требуется практическое владение как базовым, так и альтернативным языком, тогда как аналитику, владеющему базовым языком, достаточно иметь лишь общее представление о возможностях альтернативного языка.

поддерживается языком Си и даже (в отсутствие соответствующей библиотеки) не может считаться допускаемой этим языком.

С качественно иной ситуацией мы сталкиваемся при попытке сочетания двух и более *стилей программирования* в рамках одной программы, как это понадобилось бы нам для реализации схемы, приведенной на рис. 2. Комплексы понятий и приемов, характерных для того или иного языка, рассматриваемые целиком, существенно превосходят отдельные парадигмы по сложности, ведь каждый такой комплекс развился вокруг реального языка программирования (а чаще - нескольких языков) и включает в себя множество достижений специфической культуры, возникающей в сообществе пользователей каждого сколько-нибудь интересного языка программирования.

Проблема интеграции воедино понятий и приемов программирования, характерных для разнородных языков программирования, - это и есть проблема, нуждающаяся в решении, и именно решению этой проблемы посвящена настоящая работа. Если быть более точным, проблема состоит в том, чтобы предоставить программисту, владеющему, наряду с базовым языком проекта, еще и каким-либо альтернативным стилем программирования, возможность применения своих знаний и навыков в проекте, базовый язык которого рассматриваемый стиль не поддерживает. При этом решение должно быть пригодным к практическому применению в инустриальной практике, для чего необходимо снизить, насколько это возможно, накладные расходы как на внедрение нового метода в практику (барьер внедрения), так и на применение метода для решения текущих задач.

# Существующие подходы к построению программ с использованием различных стилей

Проблема сочетания разнородных языковых механизмов в рамках одного проекта (проблема мультипарадигмального программирования) изучается достаточно давно, и к настоящему моменту известно достаточно большое количество подходов к ее решению. Рассмотрим наиболее популярные из них.

## Пакеты взаимосвязанных программ

Наиболее простым с технической точки зрения подходом к решению проблемы многоязыковых проектов можно считать использование нескольких не связанных между собой систем программирования, каждая из которых поддерживает один из используемых языков. В большинстве случаев, на уровне объектного кода такие системы между собой не совместимы, так что сборка модулей воедино не представляется возможной либо оказывается неоправданно трудоемкой. Кроме несовместимости по сборке, сложности возникают в соглашениях о связях, в способах представления данных и т.д. Поэтому проект реализуется в виде набора (пакета) отдельных программ, каждая из которых может быть написана на своем языке программирования.

При этом проблема интеграции языковых средств уступает место проблеме интеграции разнородных частей пакета программ. Известны различные попытки создания единого подхода к связыванию таких компонент, в частности - технологии CORBA и COM. Однако такие технологии достаточно сложны сами по себе. Создание соответствующих CORBA-объектов и COM-компонент по накладным трудозатратам может быть сравнимо с выигрышем от применения альтернативных языков программирования, что делает многоязыковой подход неоправданным. То же можно сказать и относительно дополнительных затрат технических ресурсов; неизбежно усложняются runtime-библиотеки, возникают потери по быстродействию и объему памяти. Во многих случаях расходы, связанные с CORBA/COM, сравнимы с расходами на решение самой задачи.

Наиболее простым подходом к построению интегрированных пакетов программ является, очевидно, так называемый Unix-стиль, при котором каждая программа имеет поток стандартного ввода, стандартного вывода и имеется возможность перенаправлять эти потоки в файлы или замыкать на другие программы. Единообразие интерфейсов программ позволяет выстраивать системы произвольной сложности из элементов в виде отдельных программ, которые не предназначались для такого использования специально, а лишь следовали соглашению о наличии стандартных потоков ввода-вывода. Чтобы избежать проблем, связанных с различием внутренних представлений данных, используется универсальное представление информации в виде текстов. Это позволяет использовать одни и те же программы как в качестве элементов системы, так и по отдельности. Unix в качестве «мультипарадигмальной среды» упоминается, например, в работе [47].

Но и этот подход налагает серьезные ограничения на разработку. Так, программы вынуждены обращаться к другим программам через потоки ввода-вывода, что не всегда удобно, не говоря уже о необходимости постоянно выполнять конверсию из внутреннего представления данных во внешнее и обратно.

## Встраиваемые интерпретаторы

Возможен и несколько иной подход к интеграции между собой систем программирования на разных языках. Назовем этот подход методом встраиваемого интерпретатора. В рамках этого подхода для каждого проекта избирается язык-лидер, на котором пишется большая часть кода, а прочие языки (которые по условиям должны позволять интерпретируемое исполнение) тем или иным образом встраиваются в него.

Допустим, основной проект написан на Си++ и необходимо использовать модуль, написанный, например, на языке Лисп. Рассмотрим Лисп-интерпретатор, написанный на Си++ (очевидно, что это возможно). Оформим такой интерпретатор в виде отдельного модуля, а необходимый в проекте код на языке Лисп - в виде текстовой константы языка Си++. При инициализации программы передадим Лисп-код в интерпретирующий модуль, в результате чего получим готовую к работе виртуальную Лисп-машину с уже заданной программой. В процессе работы можно передавать в текстовом виде запросы к полученной Лисп-машине, получая в ответ результаты, опять таки, в текстовом виде.

Сформулируем основные недостатки рассмотренной методики.

- При таком решении модуль, написанный в альтернативной системе программирования, не может вызывать функции основной программы, и наоборот. Также отсутствует возможность совместного использования глобальных переменных.

- При применении интерпретаторов на этапе выполнения приходится выполнять лексический и синтаксический анализ кода, что снижает эффект от применения в проекте компилируемых языков.

- При организации обмена данными через текстовое представление мы вынуждены анализировать текстовые результаты во всех программах пакета, в том числе и на базовом языке (например, Си++). Необходимо отметить, что лексический и синтаксический анализ - это отдельный класс задач, для которых был бы удобнее, например, язык Рефал. При таком же решении вынести анализ текста в код на другом языке, очевидно, невозможно.

Существуют менее тривиальные варианты этого же решения. Например, базовый язык проекта может быть расширен конструкциями для обращения ко встроенному интерпретатору. Перед компиляцией текст на таком расширении базового языка прогоняется через специфический препроцессор, дающий на выходе код на базовом языке, свободный от дополнительных конструкций и содержащий вместо них обращения к соответствующим функциям встроенного интерпретатора. Препроцессор фактически заменяет конструкции встроенного языка вызовами интерпретатора этого языка.

Эта технология широко применяется, в частности, для встраивания языка запросов на SQL в императивные языки. Также существуют встраиваемые варианты языков Лисп, Схема [33] и других.

Хорошо спроектированные средства встраивания способны существенно облегчить программирование, избавив программиста от описания преобразований данных и т.п. Некоторые системы могут позволять также включать в код на альтернативном языке обращения к функциям базового языка, преобразуя соответствующие конструкции при препроцессировании к приемлемому для компилятора виду. Однако полностью снять все перечисленные недостатки метода встроенного интерпретатора, по-видимому, невозможно.

## Расширяемые интерпретаторы

В рамках этого подхода в качестве базового выступает некоторый интерпретируемый язык, причем интерпретатор может быть расширен программистом за счет написания дополнительных функций. Такие функции, как правило, пишутся на том же языке, на котором реализован сам интерпретатор, и могут как требовать перекомпиляции интерпретатора для добавления новых функций, так и не требовать (в случае, если используется тот или иной механизм оверлейных модулей, динамических библиотек и т.п.) Примером такой системы может служить интерпретатор TCL.

Основным недостатком подхода является необходимость использовать в качестве базового языка интерпретируемый язык программирования, что не всегда приемлемо. Следует отметить, что подход проявляет уже известные нам проблемы с разделением данных между базовым языком и дополнительными модулями интерпретатора, а также трудности в вызове функций основной программы изнутри дополнительных модулей.

Наконец, подход ограничен двумя языками - базовым интерпретируемым и дополнительным, на котором пишутся дополнительные модули к интерпретатору (язык реализации интерпретатора).

## Компиляция из одного языка в другой

Некоторые системы программирования построены на компиляции исходного текста не в объектный код, как это обычно делается, а в код на другом языке высокого уровня, чаще всего – Си [34]. В частности, так построены некоторые компиляторы языка Схема.

Ясно, что такой подход к трансляции позволяет без особых проблем сопрягать между собой модули, написанные на различных языках, с помощью обыкновенного связывания.

Применение этого метода затрудняется тем, что в реализации языка более высокого уровня организация данных, а равно и процесса выполнения программы оказывается достаточно нетривиальной, чтобы породить сложности для стороннего программиста. Поясним сказанное на примере языка Лисп, транслируемого в код на Си. Понятие S-выражения компилятор Лиспа может реализовывать одним из многих возможных способов. Также самыми различными способами можно реализовать отображение лисповской функции в код на языке Си. Программисту, желающему сопрячь код, написанный на чистом Си, с кодом, оттранслированным в Си из Лиспа, пришлось бы досконально изучить организацию S-выражений в конкретной реализации транслятора Лиспа, а также методы, используемые транслятором для отображения функций. Такое изучение не всегда оправдано по трудозатратам, поскольку вопросы реализации традиционно рассматриваются как внутренние и не включаются в пользовательскую документацию, не говоря уже о том, что в последующих версиях транслятора реализация может измениться.

## Создание нового языка

Другим подходом к решению проблемы мультипарадигмального программирования является создание некоторого нового языка программирования, в который вносятся изобразительные средства большого числа различных парадигм (в идеале - всех или почти всех концепций программирования). В качестве характерных примеров можно назвать языки Леда [20], Оз [41] и другие.

Как показывает практика, такие языки не достигают уровня промышленных инструментов, натыкаясь в своем развитии на уже упоминавшийся барьер внедрения. Создание нового языка даже при наличии эффективной реализации не влечет само по себе его широкого внедрения в практику. Специалисты начинают изучать новые инструменты только в случае очевидности и несомненной значительности их преимуществ перед существующими и изученными, что само по себе еще необходимо донести до сообщества. Кроме того, необходимо создание адекватной среды разработки, включающей системы программирования для различных платформ, широкие библиотеки и т.д. Наконец, язык, в котором на базовом уровне присутствуют возможности, относящиеся к нескольким парадигмам, неизбежно оказывается чрезмерно сложным.

## Расширение существующего языка

При этом подходе выбирается некоторый существующий язык программирования и производится его расширение путем введения дополнительных изобразительных средств, соответствующих парадигмам, в исходном языке отсутствовавшим. Удачным примером такой эволюции может служить язык Си, давший начало языку C with classes, а он, в свою очередь - языку Си++ [52]. Однако Си++ объединяет две в некотором смысле родственные концепции – процедурную и объектно-ориентированную; кроме того, Си++ возник в ответ на потребность индустрии, работающей в-основном на императивных языках, в объектно-ориентированных средствах разработки, чем, видимо, и объясняется его успех.

В то же время автор Си++ с недоверием относится к перспективе дальнейшего расширения парадигматики Си++:

> *«Не каждую задачу нужно решать на Си++ и не каждая проблема, имеющаяся в Си++, настолько серьезна, чтобы заниматься ею. Например, не следует включать*

*в сам язык средства для сопоставления с образцом или доказательства теорем...»* [52, стр. 122] [9]

Расширение какого-либо из существующих языков с целью одновременного охвата большинства существующих парадигм неизбежно столкнется с определенными проблемами. Как и при создании совершенно нового языка, основным препятствием к использованию таких модифицированных языков оказывается все тот же барьер внедрения.

Кроме того, расширяя язык чуждыми для него парадигмами, мы рискуем превратить его в бессвязное собрание возможностей, не объединенных общей концепцией. Если же этого не произойдет, то полученный язык будет иметь весьма мало общего с начальным языком, со всеми вытекающими отсюда последствиями в виде барьера внедрения.

В той же книге Бьерн Страуструп дает рекомендации тем, кто желает видеть в языке Си++ дополнительные возможности:

> *«Мы предпочитаем языковым расширениям специальные приемы программирования и библиотечные функции, где только это возможно. Многие группы программистов хотели бы, чтобы любимая ими конструкция или библиотечный класс стали частью языка. Однако включение средств, полезных той или иной части сообщества пользователей, превратило бы Си++ в конгломерат не связанных между собой возможностей».* [52, стр. 207]

Итак, по мнению автора Си++, дальнейшее расширение возможностей языка следует проводить путем создания дополнительных библиотек и освоения специальных приемов программирования.

# Новый подход: постановка задачи

Основной мотивацией постановки задачи настоящей работы является устранение отмеченных выше недостатков рассмотренных существующих подходов к сочетанию разнородных языковых механизмов в рамках одного проекта.

[9] Здесь и далее имеется в виду номер страницы в русском переводе, если в списке литературы такой перевод указан

Отметим, что предлагаемый метод должен подходить для индустриального программирования и быть адекватным реальным потребностям. Как отмечает Бьерн Страуструп,

> *«Недостаточно просто дать пользователю некоторое средство или порекомендовать способ решения определенной задачи. Предлагаемый вариант не должен требовать слишком больших затрат. В противном случае совет звучит, как издевательство»*[52, стр. 127].

Требование применимости в индустриальном программировании наводит на мысль о том, что мультипарадигмальная среда должна базироваться на одном из современных императивно-объектных языков, поскольку именно эти языки наиболее востребованы индустрией.

В пользу выбора объектно-ориентированной концепции говорят следующие факторы. Объектно-ориентированная парадигма популярна и имеет эффективные реализации в рамках различных языков и для различных платформ. Современные объектно-ориентированные языки программирования имеют мощные средства развития, богатые библиотеки классов для различных проблемных областей.

Помимо общих достоинств объектно-ориентированного программирования, отдельные языки обладают достаточно мощными синтаксическими возможностями, которые, как станет ясно из дальнейшего изложения, играют важную роль в предлагаемом решении проблемы.

Учитывая вышесказанное, сформулируем условия, которым должен удовлетворять новый метод решения проблемы интеграции основных парадигм.

1. **Универсальность**. Метод должен позволять использование нескольких различных стилей программирования (как можно большего их количества) в рамках одного проекта (одной программы).

2. **Языковая целостность**. При импорте очередного стиля должно быть возможно использование всех наиболее заметных достижений программистской культуры, сформировавшейся вокруг языка (языков), породивших данный стиль; иными словами, необходимо импортировать не абстрактный теоретиче-

ский стиль программирования, а изобразительные возможности конкретных языков программирования.

3. **Удобство**. Средства разработки кода в альтернативном стиле должны быть удобны для человека, владеющего одновременно базовым языком проекта и языком-носителем альтернативного стиля.

4. **Консервативность**. Метод должен предполагать использование существующих систем программирования. В частности, метод не должен быть связан с созданием очередного «принципиально нового» языка программирования, так как этот путь в условиях современного индустриального программировнаия бесперспективен. Кроме того, в качестве базового языка должен использоваться язык, широко используемый в индустрии, чтобы избежать необходимости широкой переориентации команд разработчиков на новые инструменты. Базовый язык должен сохраняться в неизменном виде, то есть не должен претерпевать каких-либо изменений (расширений и т.п.). Это означает, что код, написанный в рамках мультипарадигмального проекта на базовом языке, должен обрабатываться существующими стандартными средствами (компилятором, компоновщиком и т.п.). Код, написанный программистом на базовом языке проекта и, вполне возможно, содержащий обращения к подпрограммам и данным, принадлежащим альтернативным парадигмам (и даже фрагменты кода, выполненные целиком в рамках альтернативной парадигмы), не должен, тем не менее, требовать какого-либо дополнительного препроцессирования, кроме предусмотренного стандартным компилятором базового языка.

5. **Прозрачность**. Метод должен позволять естественное разделение данных между частями программы, написанными в рамках различных стилей, а также обращения к подпрограммам, написанным в одном стиле, из кода, написанного в другом стиле, для любых двух интегрированных стилей.

6. **Компилируемость**. Метод не должен быть связан с полной интерпретацией каких-либо частей исходного кода во время исполнения программы. Это означает, что на этапе исполнения не должны производиться лексический и синтаксический

анализ никаких частей кода программы, независимо от того, к какой парадигме относятся эти части кода.

Следует отметить, что ни один из подходов, рассматривавшихся в предыдущем параграфе, не удовлетворяет всем приведенным требованиям одновременно. Это автоматически означает, что решение, если таковое будет найдено, обязано качественно отличаться от рассмотренных мультипарадигмальных решений.

Необходимо отметить, что мы не требуем интеграции языков программирования как таковых – то есть, например, не требуем поддержки синтаксиса языка, изобразительные возможности которого мы импортируем ради поддержки очередной альтернативной парадигмы. Прекрасное обоснование этого подхода можно найти в книге Бьерна Страуструпа [52, стр. 215]:

*Из представления о синтаксисе как об интерфейсе между языком и пользователем следует, что возможны и другие интерфейсы. Единственная фундаментальная константа - это базовая семантика языка.*

Тем не менее, желательно, чтобы импортированная парадигма была реально доступна программисту, знающему оригинальный язык. Иными словами, синтаксис кода, разрабатываемого в альтернативной парадигме, желательно сделать естественным отображением кода, который мог бы для аналогичных целей быть разработан на оригинале альтернативного языка. Это позволяет программисту мыслить в привычных терминах; в противном случае мы вновь наткнемся на барьер внедрения.

# Глава 1

# Предлагаемый подход

Предлагаемый подход к решению проблемы мультипарадигмального программирования основывается на выборе языка-лидера как основного языка проекта. В качестве языка-лидера предлагается использовать объектно-ориентированный язык программирования, предоставляющий возможность переопределения символов стандартных операций для пользовательских типов данных. Вместо синтаксического расширения языка средствами препроцессора и тому подобных инструментов мы рассматриваем семантику альтернативного языка программирования как специфическую предметную область, подлежащую моделированию объектно-ориентированными изобразительными средствами языка-лидера (в виде соответствующей библиотеки классов). Введением набора инфиксных операций нам удается добиться возможности задания в базовом языке выражений, семантически эквивалентных и при этом синтаксически близких соответствующим конструкциям альтернативного языка.

## 1.1 Основные предпосылки

Рассмотрим подробнее, что имеется в виду. В качестве иллюстративного материала изберем Си++ (как основной язык) и Лисп (как альтернативный).

### 1.1.1 Си++ как язык моделирования алгебр

Язык Си++ позволяет программисту вводить собственные типы данных со своими наборами операций над ними («классы»), внутренняя реализация которых скрыта от остальной программы. Для классов можно описывать как обычные функции, так и операторы[1], определяя функционирование того или иного оператора в случае, если он применен не к данным предопределенных типов, а к данным, тип которых введен пользователем. Так, можно описать класс Vector3d для хранения элемента трехмерного евклидова пространства и определить для этого класса операции "+", "−", "унарный −" для сложения, вычитания векторов и умножения вектора на −1 соответственно, операцию "*" над двумя векторами для вычисления скалярного произведения векторов, ту же операцию над числом и вектором для умножения вектора на число и, например, операцию "&" (обычно использующуюся для обозначения логического И) для вычисления векторного произведения. Можно также определить, например, унарный оператор "~" (тильда) для вычисления нормы вектора, бинарный оператор "^" для вычисления расстояния между векторами и т.п. В число бинарных операций в языках Си и Си++ входит и "," (операция «запятая»), которая тоже может быть переопределена для конкретных типов операндов.

Аналогичным образом, благодаря большому количеству унарных и бинарных операций в языке Си++, допускающих переопределение для пользовательских классов, на этом языке может быть удобным образом смоделирована практически любая алгебра. Более интересные примеры приложения Си++ к алгебре можно найти в монографии [54].

### 1.1.2 Язык Лисп как алгебра

В языке Лисп 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 `()` являет-

[1] При описании языка Си++ Бьерн Страуструп, а за ним и другие авторы предпочитают использовать английское слово *operator* для обозначения символов арифметических и псевдоарифметических операций; наряду с ним используется термин *statement* для обозначения законченных конструкций языка. Это создает определенные трудности при переводе на русский язык, поскольку традиционно оба слова переводились как *оператор*. Во избежание путаницы в некоторых случаях первый термин переводится как *оператор*, а второй – как *инструкция*.

ся 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 Лисп является языком типизированных данных и нетипизированных имен.

Универсальная сущность S-выражения позволяет говорить о языке Лисп как об *алгебре S-выражений*. Множеством объектов такой алгебры является пространство 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 Лисп 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 Лисп существуют соответствующие функции. В то же время композицию стоит, по-видимому, рассматривать как отдельный класс операций, поскольку для обращения к функциям в Лиспе необходимо сначала построить список, являющийся вызовом функции, и лишь затем применить к нему операцию вычисления.

Предложенное представление языка Лисп 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.

## 1.2 Алгебра S-выражений как предметная область библиотеки классов

Чтобы представить конструкции языка Лисп средствами Си++, 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 Си++. После компиляции текста транслятором Си++ в сегменте данных основной программы появятся конструкции Лиспа, 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 Лисп. Таким образом, с помощью библиотеки классов, не изменяя собственно язык Си++, можно дать возможность программисту использовать парадигматику языка Лисп в программе на Си++.

## 1.3 Средства конструирования списков

В качестве одного из условий постановки задачи упоминается удобство работы с кодом, написанным в альтернативных парадигмах. В связи с этим при реализации алгебры языка Лисп необходимо предоставить пользователю адекватные средства конструирования лисповских списков.

Авторы монографии [54] 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 Лиспе. 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 Лиспе, вряд ли сможет эффективно работать вместо привычных списков со столь громоздкими конструкциями.

Ключом к решению проблемы является уже рассмотренная в § 1.1.1 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[2]:

```
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. [52, стр. 254]). Операцию можно перегрузить таким образом, чтобы, например, лисповский список

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

мог быть представлен[3] в виде

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

или, благодаря автоматическим преобразованиям в Си++, 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 следующий операнд к

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

[3]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.

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 Лиспа, в Си++ 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 Лиспе 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 Си++, 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.

Даже если избранный базовый язык не поддерживает перегрузку операции «запятая», можно, не утратив наглядности, использовать другие операции. Так, аналогичные усилия в языке Ада могли бы привести к такому примеру:

```
(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))
```

При этом можно было бы не вводить отдельный класс для L, а описать имя L как константу перечислимого типа и описать соответствующую функцию *.

Предложенный подход получил название «метода непосредственной интеграции» [1].

## 1.4 Выбор базового языка

Перечислим требования, которым должен удовлетворять конкретный язык программирования, избираемый в качестве базового в методе непосредственной интеграции.

Во-первых, этот язык должен включать понятие класса, поддерживающего основные свойства объектно-ориентированного программирования – инкапсуляцию, наследование и динамический полиморфизм. Все это необходимо для построения адекватной модели альтернативной парадигмы как предметной области.

Во-вторых, такой язык должен быть достаточно универсальным и известным, чтобы его использование было допустимо в большинстве проектов.

Наконец, избранный язык-лидер должен иметь возможность переопределения стандартных операций, таких как арифметические знаки, и достаточно широкий набор самих таких операций. Это необходимо для наглядности отображения кода с альтернативного языка на основной.

Этим требованиям удовлетворяют, в частности, такие языки, как Си++ и Ада 95. В то же время Java, Objective C и некоторые другие языки удовлетворяют первым двум требованиям, но не дают возможности перекрытия стандартных символов операций.

Язык Си++ имеет серьезные недостатки, в-основном унаследованные от языка Си; в частности, в Си++ отсутствуют средства модульного программирования, которые программист вынужден имитировать с помощью заголовочных файлов, препроцессора и явного управления пространствами имен. Язык Ада 95 таких недоработок лишен.

В то же время, язык Си++ предоставляет большое количество символов операций, допускающих переопределение для пользовательских классов (в том числе операцию «запятая», перекрытие которой позволяет заметно повысить наглядность средств построения списочных конструкций), тогда как в языке Ада95 множество таких операций ограничивается символами арифметических действий.

Кроме того, в силу исторических причин язык Ада и его наследник Ада 95, несмотря на несомненные достоинства обоих языков, оказались невостребованы в индустрии программного обеспечения. В то же время язык Си++ можно назвать одним из самых популярных в современном индустриальном программировании.

В силу вышесказанного в настоящей работе в качестве базового языка мультипарадигмальных проектов рассматривается именно Си++.

## 1.5 Итоги

Итак, предложен метод, позволяющий, не выходя за рамки базового языка (например, Си++), программировать в стиле, характерном для альтернативных языков, таких как Лисп, Рефал, Пролог и т.п. Метод заключается в том, чтобы штатными средствами базового языка задавать выражения, семантически эквивалентные и синтаксически близкие конструкциям соответствующих альтернативных языков. Основное отличие метода от ранее существовавших состоит в том, что при разработке программы используется только один язык (базовый) без каких-либо изменений, при этом программисту дается возможность мыслить в терминах альтернативных языков.

Намечен подход к построению объектно-ориентированной модели алгебры S-выражений языка Лисп, поддерживающей предложенный подход.

В качестве базового языка избран язык Си++.

# Заключение

## Краткий обзор проделанной работы

Описанный в настоящей работе *метод непосредственной интеграции* решает проблему мультипарадигмального программирования, предоставляя инструментарий, позволяющий в рамках одного проекта наряду с традиционными концепциями базового языка применять концепции альтернативных языков. При этом предложенное решение свободно от некоторых недостатков, свойственных другим способам решения проблемы.

Предложенная объектно-ориентированная модель лисп-машины, расширенной рефал-машины и дэйталог-машины позволяют, не выходя за рамки языка Си++ и используя, соответственно, существующие системы программирования без специальных модификаций, применять в проектах понятия и приемы, наработанные в языках Лисп, Рефал и Дэйталог, а при необходимости и расширять возможности этих приемов, как это сделано при программировании рабочей версии транслятора Рефала.

Предложенная модель порождает среду для мультипарадигмального программирования в рамках проектов, основным языком которых является Си++.

Поскольку применение предложенных методов не требует ни модификации существующих систем программирования, ни внедрения новых, барьер внедрения для предложенной методологии оказывается достаточно низок и сводится к освоению библиотеки классов InteLib, имеющей сравнительно простую структуру. Это позволяет ожидать, что предложенный метод может найти применение в индустриальном программировании.

С точки зрения изучения изобразительных возможностей языков программирования представленная в работе библиотека InteLib иллюстрирует тот факт, что язык Си++ и другие языки с аналогичными возможностями сами по себе могут использоваться в качестве мультипарадигмальных языков программирования, причем, по-видимому, способны поддерживать целиком системы парадигм, возникшие в других языках.

# Основные результаты

К основным результатам работы можно отнести следующее:

1. Предложен метод, позволяющий, не выходя за рамки базового объектно-ориентированного языка программирования, допускающего переопределение стандартных операций, программировать, используя понятия и приемы, характерные для некоторых альтернативных языков.

   Метод заключается в построении средствами базового языка модели семантики альтернативного языка, позволяющей формировать выражения, семантически эквивалентные и синтаксически близкие конструкциям альтернативного языка.

2. В рамках предложенного метода разработаны объектно-ориентированные модели алгебры S-выражений языка Лисп, Рефал-машины и Дэйталог-машины. Каждая модель вводит специфический набор операций для конструирования выражений, синтаксически близких конструкциям соответствующего языка.

3. Создана библиотека классов Си++ InteLib, реализующая вышеперечисленные модели и позволяющая практически осуществлять программирование в стиле языков Лисп, Рефал и Дэйталог, оставаясь при этом целиком в рамках языка Си++ и используя существующие для этого языка системы программирования.

# Перспективы

Среди возможных направлений дальнейшей работы следует назвать прежде всего дальнейшее совершенствование созданных библиотек. В частности, естественным развитием лисповской части библиотеки InteLib может стать более полная реализация таких возможностей языка Лисп, как частичное квотирование, иерархическая система типов, многозначные функции и т.п., что повлечет за собой расширение введенного языка Intelib Lisp, в том числе, возможно, до степени совместимости с каким-либо из существующих диалектов Лиспа.

Часть InteLib, отвечающую за парадигмы языка Дэйталог, было бы логично развить в направлении сопряжения с существующими СУБД, такими как MySQL, Postgresql, Oracle и др., применяя Дэйталог (точнее говоря, дейталогоподобные конструкции Си++) в качестве языка формирования запросов.

Наконец, что немаловажно, необходимо тщательное исследование возможностей мультипарадигмальной среды и классов задач, решение которых облегчается за счет появления в языке Си++ возможности использования альтернативных парадигм программирования. Требуется выработать рекомендации о том, в каких случаях следует применять альтернативные парадигмы и как это следует делать.

Разумно ожидать, что широкое экспериментальное и практическое применение библиотеки позволит выявить ее сильные и слабые стороны и, соответственно, более четко расставить приоритеты дальнейшей разработки.

# Благодарности

Автор считает своим приятным долгом поблагодарить своего научного руководителя Игоря Геннадьевича Головина за помощь и участие.

Автор также признателен Елене Игоревне Большаковой, в соавторстве с которой вышло две публикации по теме данной работы, за неослабевающий интерес к данному направлению исследований и практическое содействие в теоретических аспектах работы.

Автор благодарен Ирине Анатольевне Волковой, Владимиру Геннадьевичу Абрамову и Николаю Павловичу Трифонову, в разное время в ходе работы оказавших помощь советами, замечаниями и рекомендациями.

Особую признательность автор хотел бы высказать в адрес Михаила Георгиевича Мальковского, оказавшего неоценимую поддержку и в течение последнего года не позволявшего автору ни на минуту сбавить темп работы. Если бы не усилия и забота Михаила Георгиевича, работа не была бы окончена еще долго и вряд ли имела бы столь целостный вид.

# 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] E. I. Bolshakova and A. V. Stolyarov. Extending an object-oriented language with Lisp programming techniques. In *Memoria del 3er Encuentro Internacional de Ciencias de la Computacion ENC'01*, volume 2, pages 903–911, Aquascalientes, Mexico, September 2001.

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

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

[5] 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.

[6] А. П. Крюков, А. Я. Родионов, А. Ю. Таранов, Е. М. Шаблыгин. *Программирование на языке R-Лисп.* Радио и Связь, Москва, 1991.

[7] В. Ф. Турчин. *Метаязык для формального описания алгоритмических языков // Цифровая вычислительная техника и программирование* стр. 116–124. Сов. радио, Москва, 1966.

[8] В. Ф. Турчин. Метаалгоритмический язык. *Кибернетика*, 4:45–54, 1968.

[9] В. И. Сердобольский. Язык РЕФАЛ и его использование для преобразования алгебраических выражений. *Кибернетика*, 3:45–51, 1969.

[10] В. Ф. Турчин. Эквивалентные преобразования программ на Рефале. // *Автоматизированная система управления строительством*, стр. 36–68. ЦНИПИАСС, Москва, 1974.

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

[12] J. Alger. *C++ for real programmers.* AP Professional, Boston, 1998. Русский перевод: Джефф Элджер, C++: библиотека программиста. СПб., Питер, 2001.

[13] 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.

[14] 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.

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

[16] 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.

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

[18] 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.

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

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

[21] 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.

[22] 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.

[23] S. Ceri, G. Gottlob, and L. Tanka. *Logic Programming and Databases.* Springer-Verlag, Berlin, 1990. Русский перевод: С.Чери, Г.Готлоб, Л.Танка, Логическое программирование и базы данных, М.:МИР, 1992.

[24] 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.

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

[26] 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.

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

[28] E. Hyvönen and J. Seppänen. *LISP-MAALMA.* Kirjayhtymä, Helsinki, Finland, 1986. Русский перевод: Э.Хювёнен, Й.Сеппянен. Мир Лиспа. В двух томах. М.:МИР, 1990.

[29] IEEE Standard 1178-1990. *IEEE Standard for the Scheme Programming Language.* IEEE, New York, 1991.

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

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

[32] A. C. Kay. The early history of Smalltalk. *The second ACM SIGPLAN history of programming languages conference (HOLP-II), ACM SIGPLAN Notices*, 28(3):69–75, March 1993.

[33] R. Kelsey, W. Clinger, and J. Rees. Revised$^5$ report on Algorithmic Language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.

[34] B. W. Kernigan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, New Jersey, 1988. Русский перевод: Б.Керниган, Д.Ритчи. Язык программирования Си. Москва, Финансы и статистика, 1992.

[35] 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.

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

[37] W. D. Maurer. *The programmer's introduction to Lisp*. American Elsevier Inc., New York, 1972. Русский перевод: У.Маурер. Введение в программирование на языке Лисп. М.:МИР, 1976.

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

[39] J. McCarthy. *LISP 1.5 programmer's manual*. MIT Press, Cambridge, Massachusets, 1965.

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

[41] M. Müller, T. Müller, and P. Van Roy. Multiparadigm programming in Oz. In D. Smith, O. Ridoux, and P. Van Roy, editors, *Workshop on the Future of Logic Programming*. International Logic Programming Symposium, 1995.

[42] 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.

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

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

[45] D. C. Smith. *MLISP*, volume 135 of *Memo AIM*. Stanford Artificial Intelligence Project, Stanford, California, October 1970.

[46] 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.

[47] 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.

[48] G. L. Steele. An overview of common lisp. In *Proc. 1982 ACM Symposium on Lisp and Functional Programming*, pages 98–107, Pittsburgh, Pennsylvania, August 1982. ACM SIGPLAN/SIGACT/SIGART.

[49] G. L. Steele. *Common Lisp the Language*. Digital Press, 1984.

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

[51] G. L. Steele Jr. and G. J. Sussman. The revised report on Scheme, a dialect of Lisp. MIT Artificial Intelligence Memo 452, MIT, January 1978.

[52] 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.

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

[54] 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.

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

[56] V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.

[57] 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.

[58] L. Wall and R. L. Schwartz. *Programming Perl*. O'Reilly and associates, Sebastopol, CA, USA, 1990.

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

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