МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
имени М.В.ЛОМОНОСОВА

Факультет вычислительной математики и кибернетики

УДК 519.683.8

А.В.Столяров

# Интеграция изобразительных средств альтернативных языков программирования в проекты на C++



Рукопись депонирована в ВИНИТИ РАН
06.11.2001, №2319-B2001.

Москва

2001

# Содержание


**1 Введение** **3**

**2 Постановка задачи и существующие пути решения** **4**

**3 Предлагаемый метод** **7**

**4 Библиотека InteLib** **8**

4.1 Основные классы архитектуры . . . . . 8
4.2 Списки и их представление . . . . . 8
4.3 Средства конструирования списков . . . . . 9
4.4 Лисповский символ и его представление . . . . . 11
4.5 Лексическое и динамическое связывание . . . . . 12
4.6 Поддержка реализации функции SETF . . . . . 14
4.7 Функциональные типы данных . . . . . 16
4.8 Средства описания библиотечных функций . . . . . 17
4.9 Применение лисп-функций . . . . . 18
4.10 Дополнительные типы S-выражений . . . . . 18
4.11 Пример кода . . . . . 19
4.12 Вспомогательный транслятор . . . . . 20

**5 InteLib Lisp как диалект Лиспа** **22**

**6 Соображения по поводу языка Рефал** **25**

6.1 Краткий обзор языка Рефал . . . . . 25
6.2 Средства представления рефал-выражений на языке Си++ . . . . . 26
6.3 Интерпретация в терминах Рефала . . . . . 28
6.4 Трансляция с языка Рефал . . . . . 29

**7 Заключение** **29**

**8 Благодарности** **30**

**Литература** **31**

# 1 Введение

Различные языки программирования предоставляют программисту разнообразные подходы к описанию алгоритмов. В зависимости от преобладающих в данном языке традиций язык часто относят к одному из стилей программирования. Наиболее популярными в настоящее время являются императивный стиль (языки Фортран, Си[4], Паскаль и др.) и его сочетание с объектно-ориентированным стилем (Си++[13], Ада95[7] и др.).

В то же время существуют языки, для которых характерен совершенно иной поход к осмыслению проектируемой программы, нежели в императивных языках. Среди них выделяют группу языков искусственного интеллекта. К этой группе относятся такие языки, как Лисп[17] и Пролог[1], а также Рефал[14], Плэнер[9] и другие.

Говорить о принадлежности языков программирования к некоторому стилю можно лишь с определенной степенью условности. Так, языки Рефал и Лисп относят к функциональному стилю [15], основываясь на том, что в обоих языках программа представляется как набор функций, вызывающих друг друга и в большинстве случаев не имеющих побочных эффектов. Однако, эти языки предлагают совершенно разные подходы к описанию алгоритмов. В связи с этим представляется целесообразным говорить не о стилях программирования, а о более узких *парадигмах*, каждая из которых являет собой конкретную особенность изобразительных средств данного языка.

Характерными особенностями языка Лисп (составными частями его парадигматики) являются прежде всего:

- описание алгоритма как системы взаимнорекурсивных функций;
- возможность использования лямбда-исчисления как средства представления алгоритмов, возможности построения функционалов, функция как объект данных и конструирование функций программным путем на этапе счета;
- гетерогенные структуры данных произвольной топологии.

Язык Пролог характерен использованием логики предикатов для получения результата (декларативная семантика). Для отыскания результата используется поиск с возвратами (backtracking). В Прологе присутствуют, как и в Лиспе, нетипизированные списки, причем для описания данных сложной структуры возможно использование функторов, заимствованных из математической логики. Еще одной важной возможностью Пролога является унификация (unification).

Основой языка Рефал являются текстовые сопоставления с образцом (pattern matching) и текстовые подстановки, что делает язык очень удобным для обработки текстовой информации (в частности, для быстрого построения трансляторов). Введение в текстовые данные дополнительных элементов, таких как символы-метки, числа, функциональные вызовы и структурные скобки, облегчает работу с данными. Вообще, легкость в работе с данными произвольной структуры и сложности можно считать отличительной чертой всех языков искусственного интеллекта.

Несмотря на то, что каждая из перечисленных парадигм имеет сферу применения, где она была бы удобна, в реальных коммерческих проектах используются, в основном, императивный и императивно-объектный стили, а остальным стилям не уделяется должного внимания.

Преобладание императивного стиля можно объяснить отчасти его универсальностью. Кроме того, программы, написанные на императивных языках, позволяют добиться гораздо большей эффективности по сравнению с языками искусственного интелекта, т.к. последние практически не поддаются компиляции и очень далеки от аппаратных возможностей машины (разумеется, если речь идет о классической фон-неймановской архитектуре). В числе причин можно назвать и то обстоятельство, что большинство современных программ интерактивны, а для программирования интерактивности императивный стиль подходит как нельзя лучше.

Все это, однако, не объясняет ситуации, сложившейся с языками искусственного интеллекта. Практически все они, за исключением разве что Лиспа, забыты и никогда не используются в серьезных проектах. Последняя широко известная попытка реального использования языка Пролог - японский проект «компьютеров пятого поколения». О других языках вообще ничего не слышно. Лиспу повезло больше: силами многочисленных энтузиастов появился достаточно мощный стандарт Common Lisp[11]. Известны и серьезные программные продукты, написанные на Лиспе - например, профессиональный редактор текстов Emacs и знаменитый пакет AutoCad. Наконец, достаточно популярен в последние годы диалект Лиспа, получивший название «Схема» (Scheme, [3]).

Причина кроется в том, что реализации этих языков замкнуты сами на себя и практически не поддаются интегрированию в проекты, написанные на других языках. Почти в каждом достаточно сложном проекте есть отдельные части, которые было бы удобно реализовать на одном из языков искусственного интеллекта, однако в условиях, когда язык может быть только один, выбор однозначно падает на императивные языки.

Решению этой проблемы и посвящена настоящая работа.

# 2 Постановка задачи и существующие пути решения

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, нам 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).

Как уже отмечалось, 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, модифицированный для работы с внешними по отношению к программе отношениями (таблицами базы данных). Каждое отношение базы данных на уровне пролог-машины может быть представлено как n-арный (по числу атрибутов) предикат, 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 вещей, необходимых в реаль-

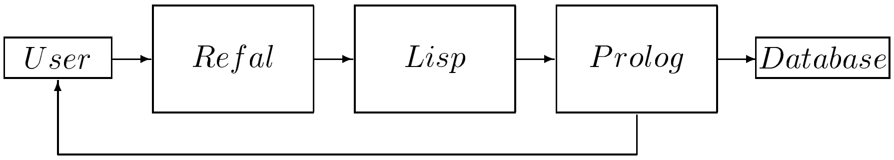


Рис. 1: Упрощенная схема примерной системы

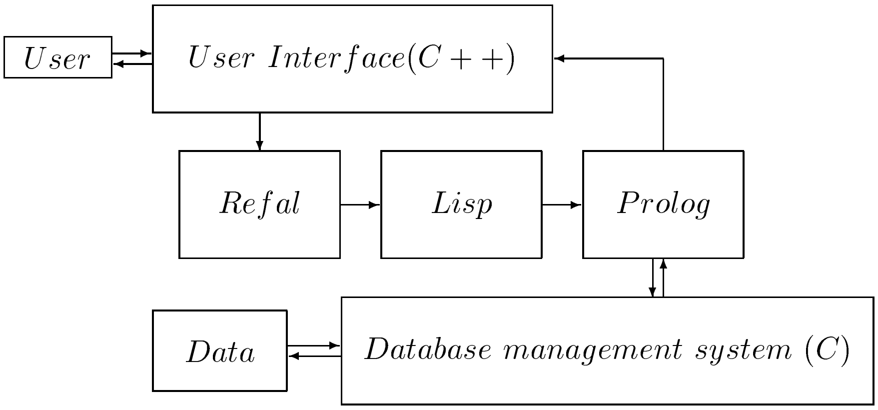


Рис. 2: Реалистическая схема примерной системы

ной 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).

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 известно, что оконные интерфейсы лучше всего писать с использованием объектно-ориентированной парадигмы - к примеру, на Си++.

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 искусственного интеллекта, с кодом на других языках.

Когда мы имеем дело с несколькими императивными языками, например, с Си и Фортраном, мы можем обычно выбрать трансляторы таким образом, чтобы они оказались совместимы на уровне объектных модулей, и связать код на уровне редактора связей, соблюдая, естественно, определенные соглашения о связях. К сожалению, языки искусственного интеллекта настолько отличны от императивных, что этот очевидный путь оказывается на практике неприменим к случаю интеграции императивного кода с кодом на одном из языков искусственного интеллекта.

Часто делаются попытки смоделировать отдельные парадигмы из дургих языков средствами базового языка. Например, гетерогенные структуры данных, доступные в Лиспе, можно смоделировать на Си++ с помощью библиотеки родовых классов (темплейтов) с использованием полиморфной иерархии контейнеров с общим предком. Собственно к интеграции языков это отношения не имеет. Вообще говоря, этот подход заведомо лишает программиста возможности использования большинства достижений специфической культуры, возникающей вокруг каждого языка программирования с его характерными особенностями.

Известны попытки создания новых языков программирования, позволяющих работу с большим, нежели обычно, множеством парадигм (языки Leda[2], Oz[8] и другие). Однако индустрия программного обеспечения консервативна в выборе инструментов, и убедить программистское сообщество в необходимости использования нового языка - задача достаточно нетривиальная, особенно в случае, если язык еще не поддержан учебной литературой, средами программирования, широкими библиотеками и т.п.

Вернемся к проблеме интеграции языков программирования как таковых. Проблему интеграции можно решить тривиальным образом. Допустим, основной проект написан на Си++ и необходимо использовать модуль, написанный на одном из ИИ-языков, например, на Лиспе. Можно взять интерпретатор Лиспа, написанный на Си++ (очевидно, что это возможно), оформить его в виде отдельного модуля, а подлежащий интеграции лисповский код - в виде текстовой константы, при инициализации программы передать интерпретатору код на Лиспе, а в процессе работы передавать в текстовом виде запросы к полученному модулю и получать обратно результаты - опять же, в текстовом виде. Если операционная система, в которой работает программа, многозадачна и поддерживает обмен данными между процессами, нет даже необходимости в написании интерпретатора именно на Си++; можно взять любой интерпретатор и вынести его в отдельный процесс.

Данное решение при его простоте имеет определенные недостатки. Вот основные из них:

- При таком решении модуль, написанный на Лиспе, не может вызывать функции основной программы.

- Лисповский код полностью интерпретируется, то есть на этапе счета приходится выполнять лексический и синтаксический анализ кода, что заметно снижает эффективность.

- Результаты обращений к лисповскому коду основная программа получает в текстовом виде. Соответственно, мы вынуждены анализировать текстовые результаты в программе на Си++. Необходимо отметить, что лексический и синтаксический анализ - это отдельный класс задач, для которых был бы удобнее, например, язык Рефал. При таком же решении реализовать анализ текста на более удобном языке, очевидно, не получится.

Очевидно, что рассмотренное решение неудовлетворительно. Существуют и иные решения, так или иначе связанные с сопряжением кода, написанного на разных языках, использующие те или иные соглашения о связях. Однако богатые синтаксические возможности современных императивно-объектных языков (прежде всего - языков Си++ и

Ада95) позволяют предложить принципиально новое решение поставленной проблемы, а именно - не изменяя исходный язык как таковой (то есть не внося никаких изменений в транслятор), ввести в программу конструкции, эквивалентные по своей семантике конструкциям других языков. Предварительное сообщение об этом методе можно найти в [12][1].

## 3 Предлагаемый метод

Рассмотрим для примера языки Си++ и Лисп. 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 «`()`» 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 Лисп является языком типизированных данных и нетипизированных имен.

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, 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 «~» 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.

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++. Пос-

[1]За время, прошедшее с момента представления статьи [12] к публикации, архитектура рассматриваемой библиотеки претерпела значительные изменения

ле 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++.

# 4 Библиотека InteLib

## 4.1 Основные классы архитектуры

При проектировании библиотеки InteLib в качестве предметной области рассматривалась виртуальная машина, оперирующая абстракциями языка Лисп (то есть S-выражениями). Очевидно, что ключевой абстракцией такой предметной области является собственно S-выражение.

Как уже отмечалось, S-выражения бывают различных типов, имеющих как различия, так и общие для всех S-выражений свойства, что логично приводит к идее использования иерархии полиморфных классов с общим предком. Исторически этот предок в библиотеке InteLib называется LTerm. Класс LTerm является абстрактным, так как имеет чисто виртуальные функции. Наследниками LTerm являются классы, представляющие конкретные типы S-выражений. Например, для представления лисповских текстовых констант используется класс LTermString, а для представления точечной пары - класс LDotPair.

Хранение данных в языке Лисп организовано динамически, причем предусмотрена сборка мусора. В связи с этим предполагается, что объекты, представляющие S-выражения, создаются всегда динамически (с помощью оператора new) и располагаютася в динамической памяти. Сборку мусора обеспечивает механизм счетчиков ссылок, организованный следующим образом. Для работы с объектами S-выражений вводится класс LReference. Поведение объекта класса LReference аналогично поведению указателя на LTerm благодаря переопределенным соответствующим образом операторам `->` и `*`. Строго говоря, предполагается, что в пользовательской программе класс LReference будет единственным способом ссылки на объекты S-выражений. Класс LTerm поддерживает счетчик ссылок на данный экземпляр. Экземпляры класса LReference сообщают объектам, на которые они ссылаются, о возникновении и ликвидации ссылки. Если счетчик ссылок доходит до нуля, объект самоуничтожается.

Для удобства использования класс LReference имеет большое количество конструкторов для преобразования к S-выражениям всех числовых типов, имеющихся в языке Си++, а также строковых констант и некоторых других типов.

Для удобства отладки предусмотрена возможность получения текстового представления S-выражений в традиционном синтаксисе Лиспа. Для этого класс LTerm определяет виртуальный метод TextRepresentation().

## 4.2 Списки и их представление

В языке Лисп списки и другие подобные структуры данных строятся с помощью так называемых точечных пар. Точечная пара - это S-выражение, представляющее два указателя на S-выражения, называемые в силу исторических причин CAR (для левого указателя) и CDR (для правого). Так, `(A . B)` - это точечная пара, CAR которой указывает на символ A, а CDR - на символ B.

Для представления точечных пар в библиотеке InteLib используется класс LDotPair, являющийся прямым потомком класса LTerm. Класс имеет два члена типа LReference, служащие для хранения указателей CAR и CDR и называющиеся car и cdr соответственно.

Как уже отмечалось, для построения линейного списка используется цепочка точечных пар, причем CAR каждой из них указывает на очередной элемент списка, а CDR - на остаток цепочки (т.е. на следующую точечную пару). Для обозначения окончания списка используется указатель на специальное S-выражение, обозначающее «пустой список». В языке Common Lisp и других диалектах Лиспа это символ NIL, в языке Scheme - специальное S-выражение «пустой список», не являющееся символом как таковым. Необходимо отметить, что в качестве метки конца списка нельзя использовать нулевой указатель, как это часто делается в других языках. Дело в том, что в языке Лисп нулевой указатель означает «неопределенное значение», что в большинстве случаев свидетельствует о произошедшей ошибке.

В библиотеке InteLib в качестве представления пустого списка обычно используется объект класса LTermSymbol, представляющий свойства символа NIL. Возможно, однако, использовать и любой другой объект иерархии LTerm, в том числе экземпляр специально определенного для этой цели класса LTermNil (это может быть использовано, например, при представлении альтернативного диалекта Лиспа, например, подмножества языка Схема). Важно лишь, чтобы объект, представляющий пустой список, был один и только один и существовал в памяти в течение всего периода выполнения программы.

При анализе списков для проверки на конец списка используется простое сравнение указателей, но указатель сравнивается не с нулем (как это обычно делается при построении списков), а с адресом объекта, избранного для представления пустого списка.

## 4.3 Средства конструирования списков

Для построения правильных списков (т.е. списков, оканчивающихся ссылкой на пустой список) класс LReference снабжен оператором «,» (запятая), выбранным из соображений наглядности. Так, лисповский список

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

может быть представлен в виде

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

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

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

Преобразования первого элемента списка к типу LReference уже достаточно для того, чтобы компилятор, обрабатывая всю конструкцию слева направо, применял везде не предопределенную операцию «запятая», а операторы, определенные в классе

LReference, автоматически преобразуя каждый следующий операнд к типу LReference через соответствующий конструктор. Но и такая сравнительно компактная конструкция выглядит неудобочитаемой из-за наличия лишних скобок в вызове конструктора LReference.

Кроме того, есть и еще одна проблема - как быть, 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 `((LReference(25),36),49)` будет в точности эквивалентна конструкции `(LReference(25),36,49)`. Можно, конечно, и здесь написать, например, `(LReference((LReference(25),36)),49)`, но такая конструкция оказывается уже совершенно 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 еще одного класса, названного LListConstructor. Этот класс не является структурой данных как таковой, поскольку не имеет ни одного члена. LListConstructor вводится ради дополнительного набора операций. В программе обычно существует ровно один экземпляр класса LListConstructor. Его имя определяется перед компиляцией библиотеки как compile-time option. Автор предпочитает использовать имя «`L`» (мнемонически оно представляет собой сокращение от слова List). Для класса определяется единственный оператор - `|` - для правого операнда типа LReference. Задача этого оператора - создать список из одного элемента, в качестве которого выступает правый операнд.

Кроме того, для представления пустых списков введено преобразование типа LListConstructor к типу LReference, возвращающее S-выражение «пустой список». Вот несколько примеров построения лисповских списков с помощью экземпляра класса LListConstructor:

```
(L| "Here I go", 25)         //   ("Here I go" 25)
(L| (L| 25, 36), 49)         //   ((25 36) 49)
(L| 99, (L))                 //   (99 ())
(L| (L| 5, 35), (L| 6, 36))  //   ((5 25) (6 36))
(L| (L| (L| (L))))           //   ((()))
```

Для конструирования точечных пар и точечных списков класс LReference снабжается оператором `||`. Такой выбор обусловлен тем, что приоритет у этого оператора должен быть ниже приоритета оператора `|`, используемого для построения списков с помощью LListConstructor. Семантика оператора `||` состоит в замене «()» на конце существующего списка, переданного как левый операнд, на терм, переданный как правый операнд. Таким образом, лисповская точечная пара (10 . 20) может быть представлена как `(L| 10 || 20)`.

К сожалению, в языке Си++ не существует операций с приоритетом ниже, чем у операции «запятая». Поэтому представление точечных списков содержит лишние скобки, указывающие на порядок применения операндов. Например, лисповский список `(1 2 3 4 . 5)` представляется как `((L|1,2,3,4)||5)`.

Кроме этого, для удобства работы введен оператор `^`, который строит точечную пару из своих операндов. Отличие между операторами `^` и `||` можно проиллюстрировать на следующих примерах:

```
(L| 1) || 2          // (1 . 2)
(L| 1) ^ 2           // ((1) . 2)
Symb1 ^ Symb2        // (Symb1 . Symb2)
Symb1 || Symb2       // Ошибка, Symb1 - не список
```

В языке Лисп важную роль играет специальная форма QUOTE, блокирующая вычисление своего аргумента. Поскольку форма используется очень часто, для нее введено сокращение - одинарный апостроф. Т.е. '<S-выр.> означает то же самое, что и (QUOTE <S-выр.>).

В библиотеке InteLib для этой цели используется унарный оператор ~ (тильда). Он определен для классов LReference и LSymbol и при обращении конструирует список из двух элементов - символа QUOTE и объекта, для которого он вызван. Это позволяет ставить тильду перед символами и списками (в том числе точечными). Константы в языке Лисп при вычислении дают сами себя, поэтому нет причин для применения к ним блокировки вычисления.

## 4.4 Лисповский символ и его представление

В языке Лисп важную роль играет абстракция, в первых работах по Лиспу [6] называвшаяся атомом.[2] В описании языка Common Lisp [11] аналогичное понятие обозначается термином «символ» (symbol), а при описании языка Scheme обычно используется термин «идентификатор». В настоящей статье мы используем термин «символ».

Символ в Лиспе - это S-выражение, имеющее имя, и, кроме того, способное иметь значение и ассоциированную функцию. В большинстве диалектов Лиспа символы обладают и другими свойствами, однако в ныне существующей версии библиотеки они отсутствуют.

Как можно заметить, символы в Лиспе используются в двух различных ипостасях – как имена переменных и как метки, для которых важна лишь уникальнось.

Для представления лисповских символов в InteLib предусмотрены классы LTermLabel (наследуется от LTerm) и LTermSymbol (наследуется от LTermLabel). Печатное имя символа задается параметром конструктора класса LTermLabel. LTermSymbol отличается от LTermLabel тем, что, в отличие от последнего, может иметь значение и ассоциированную функцию.

Объекты класса LTermLabel могут применяться в ситуациях, когда требуется некоторое S-выражение, особые свойства которого определяются им самим (в терминах реализации - его адресом в памяти), а не его значением. В Common Lisp'е все такие выражения реализованы как символы (это, например, символы NIL, T и LAMBDA). Однако при попытке реализации диалекта, подобного языку Scheme, потребуются такие S-выражения, как (), #F и #T, которые и могут быть реализованы классом LTermLabel.

В отличие от виртуальной машины языка Лисп, где имя символа (текстовая строка) является основным средством доступа к символу, при работе с компилируемым языком (Си++) логично осуществлять доступ по адресу. При этом необходимо при написании программы иметь возможность именовать символы обычными идентификаторами языка Си++.

[2] Современное значение термина «атом» несколько иное, под ним обычно понимается атомарное S-выражение (т.е. не список)

Наиболее простым решением кажется использование экземпляров класса LReference. Однако то обстоятельство, что символы используются в Лиспе в качестве переменных, заставляет подойти аккуратнее к этому вопросу. Для символов имеют смысл некоторые достаточно важные операции, которые при этом не имеют смысла для других типов S-выражений. Желательно иметь способ ссылаться на S-выражение типа «символ», гарантирующий, что работа ведется именно с этим типом S-выражения.

Проблема решается введением специального класса LSymbol, являющегося наследником класса LReference. Экземпляры этого класса описываются статически и используются при формировании лисповских конструкций. Конструктор класса LSymbol принимает в качестве параметра печатное имя символа и создает в динамической памяти соответствующий объект класса LTermSymbol, делая порождаемый объект ссылкой на него. Таким образом, становятся возможными, например, такие фрагменты кода:

```
(L| SETQ, A, 25).Evaluate();        // (setq a 25)
(L| SETQ, B, (L| LIST, A, A)).Evaluate();
                                    // (setq b (list a a))
```

Предполагается, что SETQ, LIST, CAR, A и B - это экземпляры класса LSymbol, причем в процессе инициализации с символами SETQ, LIST и CAR ассоциированы соответствующие функции.

В библиотеке InteLib эксплуатируется принцип минимизации описанных самой библиотекой объектов. В частности, библиотека описывает только четыре лисповских символа - `NIL`, `T`, `QUOTE` и `LAMBDA`. Описание (и именование) остальных символов оставлено за пользователем. Три из четырех вышеперечисленных символов необходимы самой библиотеке: символ `NIL` - как представление пустого списка, а также для возврата значения «ложь» из предикатов; символ `T` - для возврата значения «истина» из предикатов; символ `QUOTE` - для работы оператора ~, переопределенного для моделирования лисповского апострофа (см. §4.3). Что касается символа `LAMBDA`, то, в отличие от большинства остальных символов, роль этого символа определяется не только и не столько его значением и ассоциированной с ним функцией (эти признаки у него обычно пусты), сколько его конкретным экземпляром[3].

Свойства остальных символов целиком определяются их значениями и ассоциированными с ними функциями и, следовательно, нет необходимости описывать эти символы в библиотеке. Таким образом, за программистом сохраняется выбор, какие именно символы описать и под какими именами. Так, символ, именующий функцию `APPEND`, можно назвать `APPEND`, `LISP_APPEND`, `TheFunctionAppend` и т.п., а можно и вообще не описывать его, если функция `APPEND` в программе не используется.

## 4.5 Лексическое и динамическое связывание

Изменить значение, связанное с тем или иным символом, можно путем присваивания (например, с помощью формы SETQ) либо путем так называемого связывания (binding). Связывание выполняется для формальных параметров функции при входе в нее; также связывание можно выполнить явно, с помощью, например, формы LET. Изменения, вызванные связыванием, имеют в некотором смысле ограниченное действие.

[3]Как можно заметить, то же справедливо и в отношении символов NIL и T, что является еще одной причиной описания этих символов в библиотеке

Вид ограничения зависит от типа связывания, которое может быть лексическим или динамическим.

При динамическом связывании значение связывается с символом на период выполнения (вычисления) данной формы. При выходе из формы значение восстанавливается.

Чаще, однако, используется так называемое лексическое связывание. Действие лексического связывания ограничено рамками конструкции языка, для которой он введен. В отличие от динамического связывания, действие лексического связывания не распространяется на период выполнения функций, вызванных из данной конструкции, если они описаны вне ее. При этом, вообще говоря, возможно, что действие лексического связывания не будет ограничено временем однократного вычисления данной конструкции. Например, если конструкция порождает S-выражение функционального типа (это можно сделать, применив форму FUNCTION к Lambda-списку), то при обращении к такой порожденной функции на время вычисления ее тела должны действовать лексические связи, действовавшие на момент ее порождения. Это достигается с помощью так называемых лексических замыканий, т.е. объектов, представляющих совокупность вычислительной конструкции и лексических связей.

В библиотеке InteLib с каждым символом связана ссылка на S-выражение, представляющее «глобальное», или динамическое, значение символа на данный момент. Эта ссылка реализована как переменная-член класса LTermSymbol типа LReference.

Для данного символа в данный момент может также существовать так называемое лексическое значение, или лексическая связь. В случае, если найдено активное лексическое значение (символ лексически связан), используется это значение, в противном случае используется глобальное значение.

Механизм динамических связываний реализован как стек, хранящий информацию о глобальных значениях символов до выполнения связывания. Следует отметить, что стек динамических связываний можно считать в программе глобальным, если не учитывать возможности многотредового (multithread) программирования. В последнем случае стек становится принадлежностью треда. Поэтому реализация стека динамических связываний вынесена в класс LLispThread, инкапсулирующий все структуры данных, локальные в каждом треде. В поддержку стека динамических связываний класс предоставляет операции начала континуации (установить метку стека), создания связи (добавить запись о старом глобальном значении заданного символа в стек, затем изменить глобальное значение символа) и завершения континуации (просмотреть стек до ближайшей метки, восстановив соответствующие значения переменных, и удалить просмотренные записи из стека). Механизм становится работоспособным, если при входе в каждую вычислительную конструкцию установить метку стека, а при выходе из нее - вызвать операцию завершения континуации.

Лексические связывания поддерживаются объектами вспомогательного класса LLexicalContext. В отличие от динамического связывания, при лексическом связывании текущее значение символа никак не изменяется; вместо этого заводится новая запись о лексической связи данного символа. Поддерживается понятие вложенных лексических контекстов. Каждый объект лексического контекста имеет ссылку на объект объемлющего контекста. Все контексты считаются вложенными в так называемый контекст верхнего уровня, не имеющий ни одной связи (этот контекст не представлен объектом и реализуется как нулевой указатель). Если связь для данного символа не найдена в текущем (активном) контексте, производится поиск в объемлющем контексте, затем в следующем объемлющем контексте, и так до тех пор, пока не будет достигнут контекст верхнего уровня.

Лексическое замыкание реализуется путем снабжения объектов S-выражений, соответствующих функциям и макросам, ссылкой на лексический контекст (см. §4.7).

Объекты лексических контекстов могут существовать в памяти неограниченно долго, до тех пор, пока данный контекст используется функциональными объектами либо другими объектами контекстов (существуют контексты, вложенные по отношению к данному). В связи с этим для объектов лексических контекстов поддерживается механизм счетчика ссылок, аналогичный поддерживаемому в классе LTerm. За поддержание счетчика ссылок отвечают объекты класса PLexicalContext, которые семантически аналогичны указателям на LLexicalContext и используются вместо них.

Тип связывания, который необходимо применять к данному символу, определяется значением поля isdynamic в классе LTermSymbol. По умолчанию это поле имеет значение false, т.е. используется лексическое связывание. Сменить тип связывания для данного символа на динамический можно вызовом метода SetDynamicBinding либо с помощью лисповской функции DEFVAR.

## 4.6 Поддержка реализации функции SETF

В стандарте Common Lisp [11] описана функция SETF[4], представляющая обобщенное присваивание. Первый аргумент функции SETF задает «слот в памяти», значение которого необходимо изменить, второй - собственно присваиваемое значение. Слот задается формой, вычисление которой дало бы значение слота. Например, возможны следующие конструкции:

```
(SETF X 25)                ; (SETQ X 25)
(SETF (CAR A) 25)          ; (RPLACA A 25)
(SETF (SECOND B) 25)       ; (RPLACA (CDR A) 25)
(SETF (GETHASH 6 TBL) 36)  ; find key 6 in the table TBL
                           ; and bind it with value of 36
```

При этом следующие конструкции являются ошибочными, поскольку первый аргумент функции не представляет никакого существующего слота в памяти, а конструирует новый:

```
(SETF (CONS 'A 'B) C)         ; error
(SETF (+ 2 3) 23)             ; error
```

С реализацией функции SETF связана некоторая сложность. Дело в том, что обычный вызов формы, стоящей в первом аргументе, возвратит указатель на S-выражение, тогда как для осуществления присваивания необходим *адрес указателя*, который следует изменить. Проиллюстрируем это на примере. Допустим, значением символа `A` является список `(1 2 3)` и нам необходимо обработать вызов `(SETF (SECOND A) 22)` Вызов `(SECOND A)` даст нам указатель на S-выражение 2. Однако изменять значение этого S-выражения было бы некорректно, так как оно может использоваться в других конструкциях. Для осуществления запрошенной операции необходимо изменить значение указателя CAR второй точечной пары в исходном списке, заставив его ссылаться на S-выражение 22. Но адрес этого указателя мы не можем узнать, произведя простой вызов `(SECOND A)`, для этого необходимы некоторые нестандартные действия.

[4]Если говорить точнее, SETF является не функцией, а макросом

Создатели Common Lisp решили проблему, описав SETF как макрос, который по виду первого аргумента (точнее, по символу, стоящему в его CAR) определяет, какую функцию следует применить для присваивания и преобразует исходную форму в соответствующий вызов. Чтобы сделать возможным описание пользовательских функций, к которым можно было бы применять SETF, введено понятие «сложного имени функции». В качестве имени функции в Common Lisp может выступать, наряду с символом, список вида `(SETF X)`. Подразумевается, что так описывается функция, которую необходимо вызвать, чтобы произвести присваивание, соответствующее функции `X`.

Недостатком такого решения можно назвать громоздкость реализации функции SETF, которая вынуждена знать обо всех символах, связанных с функциями, допустимыми в первом аргументе SETF. Кроме того, при описании пользовательской функции, допустимой в SETF, приходится описывать две разные функции, производящие практически одни и те же действия выборки и различаются только конечной операцией (чтение либо запись), что приводит к дублированию кода и повышает вероятность ошибок при внесении изменений.

В библиотеке InteLib применен другой подход к реализации функции SETF. Механизмы возвратов при вычислении значений выражений предусматривают возможность возврата *ссылок* на объект класса LReference. В случае, если возврат по ссылке невозможен (то есть необходимо вернуть только что построенную конструкцию), используется объект LReference, специально предусмотренный для этой цели в классе LLispThread. Этот класс позволяет также проверить, имел ли место действительно возврат по ссылке, и в случае, если требуется именнно ссылка, а возврат произведен по значению, возбудить соответствующее исключение.

Способностью возврата по ссылке (если не принимать во внимание возврат через поле класса LLispThread) обладают символы (возвращается ссылка на LReference, хранящий значение данного символа), а также функциональные вызовы, если вызываемая функция реализована как допускающая возврат ссылки. В число таких функций входят CAR, CDR и их производные (такие как CDDDAR), FIRST, ... TENTH, GETHASH и т.п. Следует отметить, что в сочетании с механизмом оптимизации остаточной рекурсии это дает возможность использовать в первом аргументе функции SETF не только вызовы предопределенных функций, для которых предприняты специальные меры, но и другие лисп-выражения. Выражение оказывается допустимо на роль первого аргумента функции SETF тогда и только тогда, когда его значение вычислено как значение выражения, допускающего возврат по ссылке.

Рассмотрим пример:

```
(defun f1 (a) (if (symbolp a) a '(none)))
```

При таком определении функции `f1` вызов ее с символом в качестве параметра будет допустим в функции SETF, а с любым другим параметром - недопустим:

```
(setf (f1 'b) 25)        ; то же, что и (set 'a 25)
(setf (f1 25) 36)        ; ошибка
```

Следует обратить внимание, что, в отличие от Common Lisp, у нас нет необходимости описывать функцию, парную к `f1` (в Common Lisp'е такая функция называлась бы `(setf f1)`).

## 4.7 Функциональные типы данных

В языке Лисп функция рассматривается как объект данных, может передаваться как параметр в другие функции и конструироваться во время исполнения программы. Обычно такое использование допускается только для обыкновенных функций (т.е. функций, всегда вычисляющих все свои параметры) и запрещено для специальных форм и макросов. Для удобства реализации в библиотеке InteLib такое ограничение отсутствует.

Библиотека поддерживает следующие классы для представления функциональных S-выражений:

- LForm - абстрактный класс, представляющий общие свойства всех функциональных объектов. От этого класса наследуются все прочие классы функциональных S-выражений. Также от этого класса непосредственно наследуются классы, описывающие специальные формы;

- LCFunction - абстрактный класс, описывающий свойства обыкновенной функции языка Лисп, реализованной на Си++;

- LLispForm - абстрактный класс, описывающий общие свойства всех функциональных объектов, реализованных на языке Лисп (lambda-функций, nlambda-функций и макросов);

- LLambda - класс, представляющий обыкновенную функцию, реализованную на языке Лисп;

- LNLambda - класс, представляющий функцию типа nlambda (функцию, не вычисляющую свои аргументы;

- LMacro - класс, представляющий лисповский макрос в терминах Common Lisp.

Основными методами пользовательского интерфейса этих классов являются Call() и Apply(). Оба метода получают на вход список фактических параметров. Call() производит, если необходимо, вычисление параметров, после чего вычисляет тело функции (или другого функционального объекта). Результат вычисления возвращается как объект типа LReference по значению.

Метод Apply() отличается от метода Call() тем, что вычисление параметров перед исполнением тела функции не производится. Различие действует только для обыкновенных функций (написанных на Си++ или Лиспе), т.е. для классов LCFunction и LLambda. Для специальных форм, макросов и nlambda-функций вызовы Apply() и Call() семантически эквивалентны[5]. Метод Apply() предназначен для применения при реализации функционалов, т.е. когда фактические параметры даны в уже вычисленном виде. Напротив, метод Call() используется при вычислении форм лисп-программы, когда параметры формы подлежат вычислению.

Для создания специальной формы вводитсся класс описания данной специальной формы, наследуемый непосредственно от LForm. В таком классе достаточно переопределить метод Call()[6].

---

[5]Эта эквивалентность введена за неимением лучшего. Вообще говоря, применения Apply() для этих классов следует избегать.

[6]В реальной программе при создании специальных форм и C-функций желательно также переопределять метод TextRepresentation() для удобства отладки.

Для описания обыкновенной функции, реализованной на языке Си++, создается прямой потомок класса LCFunction. В таком наследнике достаточно переопределить защищенный метод DoCall(), принимающий на вход параметры в уже вычисленном виде, причем не как список, а как вектор объектов LReference. Этот защищенный метод вводится в классе LCFunction.

В настоящей версии библиотеки отсутствуют средства создания лисп-макросов на языке Си++. При необходимости такие средства несложно ввести, однако осмысленность такого шага вызывает определенные сомнения.

## 4.8 Средства описания библиотечных функций

Под термином «библиотечная функция» в данной статье понимается функция или специальная форма языка Лисп, реализованная библиотекой InteLib (на языке Си++).

Как уже говорилось в предыдущем параграфе, для описания библиотечной функции или специальной формы создается новый класс, наследуемый, соответственно, от LCFunction или от LForm, в котором перекрываются соответствующим образом виртуальные методы. Например, функция CONS описывается с помощью класса LFunctionCons, наследуемого от LCFunction и перекрывающего метод DoCall()[7].

Поскольку декларации классов для всех библиотечных функций, относящихся к одной категории, совершенно одинаковы, для декларации таких классов созданы макроопределения, собранные в файле libinfo.hpp. Этот файл не содержит ничего, кроме этих макроопределений, и поддерживает два режима трансляции - обычный (используемый при трансляции самой библиотеки и программ, написанных с ее помощью) и режим генератора конфигурации транслятора. Второй режим включается определением символа `LISP2CPP_TRANSLATOR_INVOKED`. Предполагается, что во втором режиме файл будет подвергнут только препроцессированию[8]. Заголовочные файлы тех модулей библиотеки InteLib, в которых описываются библиотечные лисп-функции, также оформляются с учетом возможности работы во втором режиме. Препроцессирование этих файлов дает на выходе набор директив транслятора ILL (см.§4.12), описывающий текущее состояние библиотеки.

Например, описание библиотечной функции CONS имеет вид

```
DECLARE_CFUNCTION(LFunctionCons,"#<FUNCTION CONS>",
                                          "CONS","CONS")
```

где первые два параметра используются при описании класса (имя класса и текстовое представление его объектов, используемое методом TextRepresentation), тогда как третий и четвертый предназначены для генерации директивы транслятора, которая в данном случае будет иметь вид

```
(DECLARE-FUNCTION "LFunctionCons" "CONS" "CONS")
```

(см. §4.12).

Для использования функции необходим один экземпляр класса, описывающего данную функцию (например, класса LFunctionCons). Необходимо также дать функции имя,

[7]На самом деле, перекрывается еще и метод TextRepresentation(), но для нормального функционирования достаточно перекрыть только DoCall().

[8]Для транслятора Gnu C++ это достигается с помощью ключа -E.

то есть завести объект класса LTermSymbol, ссылающийся на объект-функцию как на функцию, ассоциированную с данным символом. Для удобства таких описаний введен шаблонный класс LFunctionalSymbol, наследуемый от LSymbol. Параметром шаблона является класс, описывающий библиотечную функцию. При инстанциации порождается класс-наследник LSymbol, конструктор которого создает в динамической памяти экземпляр соответствующего класса функции и устанавливает его в качестве функции, ассоциированной с порождаемым символом. Например, если в программе необходима лисп-функция CONS, достаточно поместить в программе следующее описание:

```
LFunctionalSymbol <LFunctionCons> CONS("CONS");
```

## 4.9 Применение лисп-функций

Методы Call(), Apply() и DoCall(), описанные в классах функциональных S-выражений, не возвращают значений (имеют тип void). Для возврата используются методы класса LLispThread. Ссылка на объект этого класса передается в вышеперечисленные функции как параметр. Возврат результата вычисления лисп-функции возможен в одном из трех режимов:

- Обычный возврат. Вызывается метод RegularReturn().
- Возврат по ссылке. Вызывается метод ReferenceReturn(). В этом случае LLispThread запоминает адрес объекта LReference, ссылка на который ему передана.
- Остаточный возврат. Вызывается метод TailReturn(). В этом случае предполагается, что итоговым значением вычисляемой формы должно быть значение того S-выражения, которое передано в метод TailReturn() в качестве параметра.

За обработку последней ситуации отвечает метод Evaluate() класса LDotPair.

## 4.10 Дополнительные типы S-выражений

Кроме перечисленных выше типов S-выражений существуют потомки класса LTerm, описывающие вспомогательные S-выражения, к которым относятся, прежде всего, следующие:

- LTermStream - поток ввода/вывода. Может представлять обычный файл либо поток, связанный с текстовым буфером;
- LHashTable - хэш-таблица, связывающая произвольные ключевые S-выражения с произвольными S-выражениями в качестве значений;
- LPackage - пакет символов (таблица, позволяющая отыскивать символы по имени, а так же создавать новые символы);
- LClassicAtom - потомок класса LTermString, отличающийся только текстовым представлением. Используется вместо символов в функциях, порождающих символы (READ, MAKESYMBOL и т.п.) в случае, если по каким-то причинам недоступен пакет символов, в котором можно было бы породить символ. В отличие от объектов класса LTermSymbol, LClassicAtom не способен хранить значение, иметь ассоциированную функцию или лексическое значение.

Полностью иерархия классов библиотеки InteLib, представляющих S-выражения, показана на рис. 3

- *LTerm*
  - *LTermInteger*
  - *LTermFloat*
  - *LTermString*
    - *LClassicAtom*
  - *LTermLabel*
    - *LTermSymbol*
  - *LDotPair*
  - *LForm*
    - *LCFunction*
      - ... Лисп-функции, написанные на C++
    - *LLispForm*
      - *LLambda*
      - *LNLambda*
      - *LMacro*
    - ... Специальные формы
  - *LTermStream*
  - *LHashTable*
  - *LPackage*
  - ...

Рис. 3: Иерархия классов, представляющих S-выражения

## 4.11 Пример кода

Проиллюстрируем сказанное на примере. Рассмотрим функцию на языке Лисп, сравнивающую два дерева на предмет изоморфности их топологии (сравнение без учета значений в листьях).

```
(defun isomorphic (tree1 tree2)
 (cond ((atom tree1) (atom tree2))
       ((atom tree2) NIL)
       (t (and (isomorphic (car tree1)
                           (car tree2))
               (isomorphic (cdr tree1)
                           (cdr tree2))))))
```

Эквивалентный код на языке C++ может выглядеть так:

```
#include "intelib.h"
LSymbol ISOMORPHIC("ISOMORPHIC");
void LispInit_isomorphic() {
  static LSymbol TREE1("TREE1");
  static LSymbol TREE2("TREE2");
  (L|DEFUN, ISOMORPHIC, (L|TREE1, TREE2),
    (L|COND,
      (L|(L|ATOM, TREE1), (L|ATOM, TREE2)),
         (L|(L|ATOM, TREE2), NIL),
         (L|T, (L|AND,
           (L|ISOMORPHIC, (L|CAR, TREE1),
                          (L|CAR, TREE2)),
           (L|ISOMORPHIC, (L|CDR, TREE1),
                          (L|CDR, TREE2)))))).Evaluate();
}
```

Следует особо отметить, что приведенный выше фрагмент кода является кодом на языке Си++, не требующим какого-либо дополнительного препроцессирования. Таким образом, не выходя за пределы языка Си++, мы получаем возможность применения парадигм языка Лисп.

## 4.12 Вспомогательный транслятор

Несмотря на наглядность и простоту формирования лисповских конструкций средствами описанной библиотеки, большие фрагменты кода на Лиспе удобнее писать в традиционном синтаксисе языка Лисп. Чтобы позволить это, разработан специальный транслятор с Лиспа на Си++, получивший название ILL[9]. Очевидно, что правила такой трансляции достаточно просты. Получаемый в результате трансляции код на языке Си++ пригоден для чтения и модификации человеком.

Транслятор получает на вход один или более файлов, содержащих код в традиционном синтаксисе языка Лисп. Все полученные файлы рассматриваются как составляющие одного модуля (т.е. результат трансляции нескольких файлов эквивалентен результату трансляции их конкатенации).

Ходом трансляции можно управлять с помощью директив транслятора, объединяемых в блоки. Блок директив оформляется как форма верхнего уровня языка Лисп, имеющая первым элементом символ с именем `%%%`. Таким образом, для совместимости с другими трансляторами языка Лисп (например, диалекта Common Lisp) необходимо в окружении описать макрос с именем `%%%`, принимающий на вход произвольное количество параметров и разворачивающийся, например, в пустой список[10]. Кроме символа `%%%`, блок содержит одну или несколько собственно директив - списков вида `(DIRECTIVE-NAME param1 param2 ...)`. Примеры директив будут приведены ниже.

Результатом трансляции являются два файла, содержащие код на языке Си++ - собственно файл модуля и заголовочный файл к нему. В модуле описываются два класса, используемые для инкапсуляции лисповских символов и потому условно называемые

[9]Сокращение слов InteLib Lisp

[10]Использование макроса, а не функции, предохраняет нас от попыток интерпретатора вычислить параметры директивы, что может привести к ошибкам.

«пакетами» (packages). Один из пакетов инкапсулирует описываемые в данном модуле имена библиотечных лисповских функций (CAR, CONS и т.п.), второй - имена лисповских переменных, введенных пользователем. Для удобства последний класс описывается как наследник первого. Также этот класс содержит функции, соответствующие формам верхнего уровня исходной лисп-программы. Эти функции вызываются конструктором класса.

Кроме того, модуль описывает некоторые лисп-символы как доступные извне (публичные). Такие символы описываются вне пакетов в файле кода и снабжаются описанием с директивой extern в заголовочном файле. Указание транслятору описать тот или иной символ как публичный дается директивой (DECLARE-PUBLIC <symbol>). Также возможно вообще избежать описания символа в модуле, предполагая, что он доступен в другом модуле как публичный. Это делается директивой (DECLARE-EXTERNAL <symbol>).

Наконец, модуль описывает функцию инициализации, основная задача которой - породить экземпляр пакета символов. Эта функция должна быть вызвана главной программой перед началом работы с модулем.

При трансляции кода с языка Лисп в язык Си++ возникает проблема различия пространств имен идентификаторов. Множество литер, допустимых в именах лисповских символов, существенно больше, чем множество литер, разрешенных в идентификаторах Си++. В именах лисповских символов, кроме традиционных латинских букв и цифр, допускаются литеры +, -, *, /, &, ^, %, $, @, !, ~, :, ;, ., ?, а также, вообще говоря, любые другие литеры (при использовании экранирования). Кроме того, символ в Лиспе может начинаться с цифры. Лексема интерпретируется как число, если она может быть так проинтерпретирована, в противном случае она интерпретируется как символ.

Для построения взаимно однозначного отображения лисповских имен символов на строгое подмножество допустимых идентификаторов C++ используется тот факт, что имена лисповских символов при переводе во внутреннее представление приводятся к верхнему регистру и, таким образом, symbol, SYMBOL и SyMBoL - это один и тот же символ, имеющий внутреннее представление "SYMBOL". В диалекте Common Lisp существует возможность задать явно литеру, входящую во внутреннее представление - например, Sym\\bol - это символ, внутреннее представление имени которого - "SYMbOL". На практике эта возможность используется крайне редко и мы попросту будем считать, что в нашем диалекте, используемом как входной язык транслятора, она отсутствует. Это позволит зарезервировать латинские буквы нижнего регистра и провести трансляцию имен по следующим правилам:

- Латинские буквы приводятся к верхнему регистру;
- цифры и знак подчеркивания сохраняются без изменений;
- прочие символы заменяются словами, состоящими из латинских букв нижнего регистра в соответствии с некоторой таблицей представления литер; эта таблица формируется директивами DECLARE-CHAR;
- если имя символа начинается с цифры, перед этой цифрой ставится некоторая латинская буква нижнего регистра, например d;
- некоторые из полученных имен заменяются на другие в соответствии с таблицей замены имен во избежание конфликтов имен (например, лисповский символ

null, означающий стандартный системный предикат, преобразуется в идентификатор lNULL во избежание конфликта с макросом NULL, по традиции означающим в Си и Си++ нулевой указатель). Такое преобразование задается директивой DECLARE-NAME, например (DECLARE-NAME "NULL" "lNULL").

Приведем несколько примеров преобразования имен:

```
     Lisp      |      C++
-------------------------------
   Foobar      |  FOOBAR
   read-char   |  READdashCHAR
   -           |  minus
   <           |  lt
   char<       |  CHARlt
   s025        |  S025
   025s        |  d025S
   null        |  lNULL
   file        |  lFILE
```

Помимо лисповских символов, транслятор порождает и другие сущности, нуждающиеся в именовании - в частности, необходимы имена для порождаемых файлов, для классов, инкапсулирующих символы, и для инициализирующей функции.

Используемые соглашения об именованиях основаны на понятии имени модуля. Имя модуля задается директивой MODULE-NAME. Порождаемые файлы получают по умолчанию имена <module_name>.hxx и <module_name>.cxx (соответственно заголовочный файл и файл реализации). Функция инициализации модуля по умолчанию называется LispInit_<module_name>. Класс, инкапсулирующий имена библиотечных функций, по умолчанию именуется PackageLisp_<module_name>, а класс, инкапсулирующий символы, введенные пользователем - PackageUser_<module_name>. Все эти имена могут быть изменены с помощью директив транслятора.

Еще одна важная группа директив транслятора используется для сообщения транслятору о лисповских функциях и символах, реализованных библиотекой, а также об именах заголовочных файлов модулей библиотеки, которые необходимо подключить. Это директивы DECLARE-FUNCTION, DECLARE-SYMBOL и ADD-LIB-HEADERS. Файл, содержащий набор директив, описывающих текущее состояние библиотеки, генерируется автоматически с помощью препроцессора языка Си (см. §4.8).

# 5 InteLib Lisp как диалект Лиспа

Код Си++, представляющий средствами рассмотренной библиотеки аналог кода на языке Лисп, не требует никакого нестандартного препроцессирования и, таким образом, остается кодом, принадлежащим языку Си++, несмотря на использование парадигматики языка Лисп. Если не принимать во внимание рассмотренный транслятор[11], то при работе с библиотекой InteLib никакого языка, кроме Си++, не используется.

[11] такое допущение вполне корректно, поскольку транслятор носит вспомогательную функцию, введен для удобства и его использование никоим образом не обязательно

Тем не менее, при программировании с помощью рассмотренной библиотеки приходится мыслить в терминах языка Лисп. Следует отметить, что средства библиотеки реализуют аналог некоего гипотетического варианта языка Лисп, не являющегося, строго говоря, ни строгим подмножеством, ни строгим надмножеством ни одного из известных существующих диалектов.

Описанный в §4.12 транслятор реализует (воспринимает на входе) именно этот диалект Лиспа. Однако даже в случае, если бы транслятора ILL не существовало, диалект можно было бы рассматривать как прообраз C++-конструкций, строящихся с помощью библиотеки.

Построенный диалект получил название InteLib Lisp. В-основном он строился как сильно усеченное подмножество Common Lisp, однако строгим подмножеством Common Lisp не является, поскольку в InteLib Lisp включены некоторые возможности, не предусмотренные в Common Lisp. В частности, это

- функции типа NLAMBDA;
- точечные списки формальных параметров функций для описания функций с переменным числом параметров;
- возможность применения операции `+` к строкам в качестве операции конкатенации;
- S-выражения типов LTermLabel и ClassicAtom (см. §4.10);
- функции файлового ввода-вывода, семантически аналогичные функциям ANSI C.

Остальные отличия InteLib Lisp от Common Lisp состоят, как правило, в отсутствии тех или иных возможностей. В частности:

- Отсутствуют так называемые ключевые слова (keywords), т.е. символы-константы, имена которых начинаются с двоеточия. В Common Lisp такие символы играют особую роль (в частности, при интерпретации списков формальных параметров). В InteLib Lisp эта особая роль не реализована.
- Не производится интерпретация особых ключевых символов в списках аргументов `&rest`, `&key`, `&optional` и `&aux`. Построить функцию с переменным числом параметров в InteLib Lisp возможно, используя точечный список параметров или одиночный символ вместо списка параметров, аналогично тому, как это делается в языке Scheme [3]. Ввести локальную переменную можно с помощью формы `LET`, так что отсутствие `&aux` не снижает изобразительной мощности. Что же касается «ключевых слов», то они не введены из соображений упрощения реализации.
- Отсутствует теоретико-множественная иерархия типов S-выражений в том виде, в котором она есть в Common Lisp. Библиотека InteLib поддерживает некоторое количество типов S-выражений и предоставляет пользователю возможность введения своих типов, однако никакие типы не рассматриваются как надмножества или подмножества других типов. Любое S-выражение имеет один и только один тип. Функции `COERCE` в InteLib Lisp нет. Это ограничение также введено из соображений упрощения реализации.

- Имеется всего два числовых типа - целые числа и числа с плавающей точкой. Размер и точность, используемые для этих типов, определяется с помощью опций библиотеки.

- Отсутствует специальный тип для терминальных символов. Для представления терминального символа используется строка длиной в 1 элемент. Таким образом, в InteLib Lisp S-выражения `#\A` и `"A"` эквивалентны.

- Отсутствуют структуры и массивы. Реализация таких типов была бы несложной, но по сути своей они чужды Лиспу. Для работы с такими типами данных в мультипарадигмальном проекте следует, по-видимому, пользоваться императивными конструкциями, естественными для Си++.

- Строка символов, за неимением массивов и механизмов работы с ними, представляет собой единую и неделимую единицу данных. В связи с этим функция `CHAR` не может использоваться в качестве аргумента формы `SETF`.

- Отсутствует большинство конструкций, использующих определенные символы в качестве специальных меток (например, форма `LOOP`, использующая символы while, until и др.). Это обусловлено тем, что при реализации библиотеки минимизировалось, насколько возможно, количество символов, описываемых самой библиотекой (см. §4.4), а для реализации одной только формы `LOOP` пришлось бы описать в библиотеке около 20 объектов класса LSymbol, навязывая, таким образом, программисту особенности именования этих объектов. Это, впрочем, не означает, что описание таких конструкций невозможно.

- Отсутствуют формы декларирования типов S-выражений.

- Отсутствуют (не реализованы к настоящему моменту) механизмы block/return и catch/unwind.

- Отсутствуют средства передачи и приема множественных значений.

- Символы в языке InteLib Lisp не имеют так называемых списков свойств (property lists).

- Отсутствуют конструкции «частичного квотирования» (конструкты «обратный апостроф» и «запятая» в Common Lisp).

- В библиотеке отсутствует возможность лексического ассоциирования функции или макроса с символом. Соответственно, в диалекте нет форм `FLET` и `MACROLET`.

- Как уже говорилось в §4.6, принцип реализации функции `SETF` в библиотеке существенно отличается от принятого в Common Lisp. Одно из основных отличий, полученных на уровне диалекта, состоит в невозможности описания функций, которые, будучи примененными в первом аргументе `SETF`, отличались бы по функциональности от своих обычных версий. Это привело, в частности, к весьма ограниченной применимости функции GETHASH.

Необходимо отметить, что при работе над проектом не ставилось цели достичь совместимости с существующими диалектами Лиспа. Дело в том, что современные диалекты во многом обусловлены замкнутостью культуры Лиспа на себя. Стремление сделать язык как можно более универсальным привело к появлению в Лиспе конструкций, по сути не имеющих ничего общего не только с функциональным программированием, но и с изначальной лисповской парадигматикой вообще.

Напротив, библиотека InteLib и поддерживаемый ею диалект Лиспа с самого начала разрабатывались в качестве дополнительных языковых средств для использования в проектах, основным языком которых является императивно-объектный Си++. Это обстоятельство делает ненужным поддержку в библиотеке и диалекте средств, чуждых для лисповской парадигматики и пришедших в Лисп из других (прежде всего - императивных) языков.

В конечном счете, основной целью проекта была не возможность использовать Лисп как таковой в программе на Си++, а обогащение языка Си++ парадигмами, развившимися вокруг Лиспа. Язык Лисп в этом контексте интересен не сам по себе, а исключительно как носитель парадигм программирования, недоступных или слабо доступных в языке Си++.

# 6 Соображения по поводу языка Рефал

Аналогичная методика может быть применена и к другим языкам искусственного интеллекта. Опишем, как планируется реализовать с помощью описываемой методики функциональность языка Рефал.

## 6.1 Краткий обзор языка Рефал

В языке Рефал программа строится из функций, каждая из которых задает алгоритм преобразования (через сопоставление с образцами и замену) входных данных, представленных в виде некоторой «обобщенной символьной строки». Кроме терминальных символов, обобщенная строка может содержать так называемые символы-метки и символы-числа. И те и другие можно рассматривать как расширение алфавита вычислителя. Кроме того, в обобщенной строке могут встречаться подстроки, заключенные в круглые скобки (так называемые структурные скобки) и в угловые скобки. Последние используются для обращения к функциям. Таким образом, выражения, обрабатываемые рефал-машиной, представляют собой в общем случае деревья данных произвольной топологии, состоящих из текстов в произвольным образом расширенном алфавите, содержащие указания о необходимых вычислениях в виде обращений к функциям.

При вычислении выражения рефал-машина выбирает самое левое и самое внутреннее обращение к функции. Таким образом, выражение, поданное на вход рефал-функции, никогда не содержит обращений к функциям. После вычисления функции в исходном выражении вычисленное обращение заменяется на результат вычисления, после чего рефал-машина повторяет действия по поиску самого левого и внутреннего функционального обращения. Работа рефал-машины считается нормально завершенной, если в вычисляемом выражении не остается обращений к функциям. Такое выражение, полученное в результате вычислений, считается результатом работы рефал-программы.

Помимо предопределенных (системных) функций, встроенных в рефал-машину, рефал-машина имеет функции, определенные пользователем, которые и составляют рефал-программу. Каждая такая функция состоит из одного или более правил преобразования вида <образец> = <замена>. Образец представляет собой выражение, содержащее любые допустимые в Рефале данные (кроме, естественно, обращений к функциям), а также так называемые рефал-переменные. Все данные представляют сами себя, а переменные представляют некоторые допустимые множества значений. Всего существует 3 типа переменных: S-переменные, обозначающие отдельный символ (терминальный или символ-метку); T-переменные, обозначающие терм (символ либо выражение в структурных скобках); E-переменные, обозначающие любое выражение, в том числе пустое. Каждое вхождение переменной в образец имеет свой идентификатор (номер или имя) и, возможно, указания на дополнительные ограничения множества символов, которые могут составлять выражение, соответствующее данному вхождению переменной.

Правая часть правила преобразования («замена») представляет собой произвольное рефал-выражение, в которое могут входить обращения к функциям, а также переменные, входящие в образец данного правила. В случае, если входные данные функции успешно сопоставляются с образцом некоторого правила, входящего в функцию, результат выполнения функции строится из правой части «сработавшего» правила путем замены входящих в него переменных выражениями, которые были сопоставлены этим переменным в образце.

## 6.2 Средства представления рефал-выражений на языке Си++

Рассмотрим принципы представления конструкций языка Рефал средствами Си++. Древовидные данные, состоящие из строк терминальных символов, символов-меток, символов-чисел и структурных скобок (то есть выражения, допустимые на входе в рефал-функцию), очевидным образом изоморфны лисповским деревьям, состоящим из строковых констант, символов и чисел. Например, рефальское выражение `'abc' symb ('def' 25)` соответствует в смысле рассматриваемого изоморфизма лисповскому `("abc" symb ("def" 25))`[12]. Этот изоморфизм позволяет использовать для представления рефальских выражений уже существующие средства, разработанные для Лиспа. Приведенное выше выражение можно представить на Си++ как `(L|"abc",SYMB,(L|"def",25))`.

Для представления рефальских угловых скобок, обозначающих вызовы функций (так называемые «активные списки»), введем новый класс RfCall, наследуемый от LTerm и инкапсулирующий обычный список. Для построения таких активных списков введем класс, аналогичный описанному ранее LListConstructor. Назовем его, например, RfListConstructor и опишем его единственный экземпляр с именем R. Также введем унаследованный от LReference класс RfCallReference, предназначенный для работы с активными списками. Основное отличие RfCallReference от LReference состоит в измененной семантике операции «запятая», которая в данном случае предполагает, что

[12]Здесь необходимо особо заметить, что изоморфизм является достаточно условным. На самом деле, рефальские объектные выражения изоморфны подмножеству лисповских S-выражений, а именно множеству списков, в которых нигде не встречается двух символьных строк подряд. В реализуемой версии библиотеки атомарные S-выражения считаются не соответствующими никаким выражениям языка Рефал и при попытке использования в качестве таковых вызывают ошибку. Что касается списков с двумя и более строковыми константами подряд, то такие списки считаются эквивалентными спискам, в которых следующие подряд константы заменены на их конкатенацию.

левым операндом является не списочное S-выражение, а выражение типа RfCall. Соответственно, добавление нового элемента производится к инкапсулированному в него списку.

Теперь рефальское `<fun1 'a' sym1 'b'>` может быть представлено в виде `(R|FUN1,"a",SYM1,"b")`.

Для представления рефальских переменных опишем специальный класс RfVariable, также наследуемый от LTerm. Строго говоря, никакими свойствами терма рефал-переменная не обладает, и такое наследование преследует единственную цель - позволить использовать для построения правил преобразования уже существующие средства.

Класс RfVariable необходимо снабдить методами, используемыми в процессе сопоставления выражений с образцами. Сам класс RfVariable следует сделать абстрактным, а переменные конкретных типов описывать с помощью наследуемых от RfVariable классов. Таким образом, можно, при необходимости, расширить множество типов рефал-переменных, введя, например, переменные, способные соответствовать только заданному регулярному выражению и т.п.

Для представления рефал-функций следует ввести класс RfFunction, наследуемый от LForm. Вычисление рефал-вызова аналогично вычислению лисповской спецформы, т.к. предварительного вычисления параметров производить не следует. Тело рефал-функции можно представить в виде списка предложений, каждое из которых, в свою очередь, представляется точечной парой, CAR которой содержит левую часть предложения, а CDR - правую.

Для примера рассмотрим рефал-функцию «палиндром», приведенную в [14]:

```
Pal {
                  = True;
      s.1         = True;
      s.1 e.2 s.1 = <Pal e.2>;
      e.1         = False;
    }
```

Эта функция может быть представлена на Си++ приблизительно следующим образом:

```
LSymbol PAL("PAL");
LSymbol True("True");
LSymbol False("False");
RfVariable_E e_1("e.1"), e_2("e.2");
RfVariable_S s_1("e.1");
void InitRefalFunctionPal()
{
  LReference the_function =
     new RfFunction((L|
           (L)                ^ (L| "True"),
           (L| s_1)           ^ (L| "True"),
           (L| s_1, e_2, s_1) ^ (L| (R| PAL, e_2)),
           (L| e_1)           ^ (L| "False")
                   ));
  PAL->SetFunction(the_function);
}
```

Библиотечные функции языка Рефал также можно представить с помощью классов, наследуемых от LForm. Важно понимать, что рефал-функция обязана возвращать допустимое рефал-выражение, то есть, в терминах Лиспа, список. Результатом вычисления выражения, в котором встречаются несколько функциональных вызовов подряд, будет их конкатенация. Например, вычисление рефальского выражения `<+ 1 2><+ 3 4>` даст `3 7`. В терминах библиотеки первое выражение выглядит как `(L|(R|Plus,1,2),(R|Plus,3,4))`, а второе - как `(L|3,7)`.

## 6.3 Интерпретация в терминах Рефала

Поскольку алгоритм вычисления конструкции в семантике Рефала существенно отличается от вычисления той же конструкции в семантике Лиспа, использование для этой цели метода `Evaluate()` невозможно (если только мы не согласимся на переписывание большинства классов лисп-иерархии). В этой связи логичным представляется введение функции, внешней по отношению к уже существующей иерархии, которая бы осуществляла альтернативную интерпретацию (то есть альтернативное отображение множества S-выражений на самое себя). Эта функция может быть названа, к примеру, `RefalMachineCall`.

Одной из основных операций, осуществляемых виртуальной машиной Рефала, является сопоставление выражения с образцом. На практике достаточно часто возникают ситуации, когда требуется именно эта операция сама по себе, а не исполнение полноценной рефал-программы. В связи с этим целесообразно сделать доступной для пользователя функцию, реализующую эту операцию (например, `RefalMatch`). По этой же причине пользователю следует предоставить доступ к классу, инкапсулирующему контекст (множество связей рефал-переменных со значениями).

Целесообразно также предусмотреть возможность вызова из рефальских конструкций существующих лисповских функций. При этом следует помнить о различии между множеством всех S-выражений и множеством S-выражений, допустимых как представление рефал-выражения. Кроме того, необходимо учитывать различие семантик двух языков. Проиллюстрируем возникающую проблему на примере. Рассмотрим рефал-выражение

```
<CAR (1 2 3)><CAR (10 20 30)><CAR (100 200 300)>
```

где CAR обозначает стандартную лисп-функцию. Логично ожидать, что результатом вычисления такой конструкции должно стать рефал-выражение `1 10 100`, или, иначе говоря, S-выражение `(1 10 100)`. Однако не все так просто. Первый активный вызов, если не принять специальных мер, вернет S-выражение `1`, второй - `10`, третий - `100`. Все три, следует отметить, не являются допустимыми в качестве рефал-выражения, что уже означает возникновение ошибочной ситуации. Но даже если бы все три результата случайно оказались принадлежащими допустимому множеству, - например, если бы исходный вызов выглядел как

```
<CAR ((1) 2 3)><CAR ((10) 20 30)><CAR ((100) 200 300)>
```

то даже в этом случае полученный результат несколько отличался бы от того, чего мы ожидали (`(1 10 100)` вместо `((1)(10)(100))`.

Причина этого в семантическом различии интерпретаций двух языков. Как уже отмечалось, результат вычисления конкатенации нескольких рефал-выражений есть *конкатенация* результатов, то есть, в терминах Лиспа, результат применения функции `APPEND` к полученным результатам, тогда как в Лиспе в аналогичной ситуации традиции подсказывают ожидать чего-то похожего на результат функции `MAP` (список, построенный из результатов).

Преодолеть проблему можно, введя возможность помечать особым образом те активные списки, которые вызывают лисповскую функцию. При вычислении такого списка полученный результат, прежде чем быть возвращенным вызывающей функции, инкапсулируется в список из одного элемента. Для этого в классе RfListConstructor можно предусмотреть дополнительный оператор, например, `||`, который делал бы то же самое, что и штатный оператор `|`, отличаясь от него только выставлением соответствующего признака у порождаемого активного списка. Класс, представляющий активный список, должен уметь отрабатывать этот признак соответствующим образом.

## 6.4 Трансляция с языка Рефал

Для перевода кода с языка Лисп в конструкции Си++ был создан транслятор, достаточно простой по своей структуре. Аналогичным образом можно создать транслятор для Рефала, причем простота синтаксиса языка Рефал позволяет надеяться, что такой транслятор будет еще проще.

Наконец, реализация языка Рефал в виде надстройки над конструкциями языка Лисп открывает возможность построения дополнительного языка, имеющего как конструкции Лиспа в их первоначальной роли, так и активные списки Рефала с их стратегией вычисления. Следует отметить, что при имеющихся средствах для построения такого языка достаточно написать еще один сравнительно простой транслятор, не внося никаких изменений в библиотеку.

# 7 Заключение

Предложенный метод позволяет, не выходя за рамки существующего языка Си++, использовать при разработке програмы парадигмы, импортированные из других языков, в частности - из языков Лисп и Рефал.

В настоящее время завершена и готовится к изданию лисповская часть библиотеки InteLib, а также транслятор с языка Лисп в конструкции Си++. Параллельно ведется работа над рефальской частью библиотеки. Предполагается, что законченный вариант библиотеки сделает возможным проектирование архитектуры программных систем с учетом возможности использования альтернативных языковых средств без значительных затрат ресурсов на интеграцию фрагментов кода, выполненных на разных языках.

Возможно применение аналогичной методики и к другим языкам, в частности - к языкам Пролог и Плэнер. При этом в качестве базового языка может быть использован не только Си++. В перспективе возможно построение аналогичных библиотек и для других языков, обладающих подобными возможностями, например, для языка Ада95.

# 8 Благодарности

Автор хотел бы поблагодарить Ирину Анатольевну Волкову за вычитывание, правку и рецензирование рукописи.

Автор признателен Владимиру Геннадиевичу Абрамову за советы и рекомендации по тексту статьи.

Автор также благодарит Елену Игоревну Большакову и своего научного руководителя Игоря Геннадиевича Головина за внимание, уделенное данному направлению исследований.

# Список литературы


[1] И. Братко. Программирование на языке Пролог для искусственного интеллекта. М.: Мир, 1990

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

[3] Kelsey R. at al. Revised$^5$ report on Algorithmic Language Scheme.

[4] Kernigan, B.W., Ritchie, D.M.: The C Programming Language. - Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1978

[5] Дж. Макаллистер. Искусственный интеллект и Пролог на микроЭВМ. Пер. с английского. М.: Машиностроение 1990.

[6] McCarthy J. Recursive functions of symbolic expressions and their computation by machine. Communications of the ACM, 3 (1960), 184-195

[7] Mitchell R. Abstract Data Types and Ada. Prentice Hall, 1996.

[8] Müller M., Müller T., Van Roy P.. 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.

[9] В. Н. Пильщиков. Язык ПЛЭНЕР. М.:Наука, 1983.

[10] Robinson J. Logic programming - past, present and future. New Generation Computing, 1, 1983, p.107-121.

[11] Steele G. L. Common Lisp the Language, 2nd edition. Digital Press, 1990.

[12] Stolyarov A., Bolshakova E., Building Functional Techniques into an Object-oriented System. Knowledge-Based Software Engineering (proceedings of the 4th JCKBSE, Brno, Czech Republic, 2000), IOS Press, 2000. p.101-106

[13] Stroustrup B. The C++ Programming Language, 3rd edition. Addison-Wesley. 1997.

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

[15] А. Филд, П. Харрисон. Функциональное программирование. Пер. с английского. М.:Мир, 1993

[16] П. Хендерсон. Функциональное программирование. Применение и реализация. Пер. с английского. М.:Мир, 1983.

[17] Э. Хювенен, Й. Сеппянен. Мир Лиспа. В 2х томах. Пер. с финского. М.:Мир, 1990

[18] Дж. Элти, М. Кумбс. Экспертные системы: концепции и примеры. Пер. с английского. М.: «Финансы и статистика», 1987.