УДК 004.43(042.4)

ФОРМЫ ДЛЯ ПОКАЗА РЕЗУЛЬТАТОВ СРАВНЕНИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ НА ПРИМЕРЕ ДИАЛЕКТОВ ЯЗЫКА LISP

Л. В. Городняя[0000-0002-4639-9032]

Институт систем информатики им. А. П. Ершова СО РАН,
г. Новосибирск, Россия
Новосибирский государственный университет, г. Новосибирск, Россия

lidvas@gmail.com

Аннотация

Статья посвящена выработке форм для показа результатов анализа и сравнения особенностей языков, систем и парадигм программирования. Предлагаемая форма продемонстрирована на примере результатов сравнения языка Lisp, наиболее успешных его диалектов (Scheme, Common Lisp, Racket, Clojure) и парадигмы функционального программирования на разных уровнях определения языков и систем программирования. Форма позволила лаконично показать наследование ряда особенностей языка Lisp и их развитие в диалектах на уровне конкретного синтаксиса, абстрактной семантики и системной прагматики.

Ключевые слова*: язык программирования, Lisp, Scheme, Common Lisp, Racket, Clojure, функциональное программирование, сравнение языков программирования, конкретный синтаксис, абстрактная семантика, системная прагматика.*

ВВЕДЕНИЕ

Очередной этап разработки методики анализа и сравнения языков, систем и парадигм программирования потребовал специальных форм для лаконичного представления и показа результатов применения этой методики. Статья посвящена текущим исследованиям, продолжающимся в Лаборатории информационных систем Института систем информатики им. А. П. Ершова Сибирского отделения Российской академии наук (СО РАН) в рамках тематики, связанной с методами преподавания программирования, требующими для контроля успехов в обучении оценки продуктивности программирования и производительности

программ. Ранее была выработана визуально-табличная форма показа категорий семантических систем языка программирования, затрагивающая кроме абстрактной семантики механизмы системной прагматики [1]. Теперь начинается проверка разработанной парадигмально-семантической методики на конкретных долгоживущих языках программирования.

Изложение начинается с описания форм и обозначений для демонстрации разноуровневых различий между диалектами языка программирования. Затем дана краткая справка о языке Lisp и его диалектах Pure Lisp (1962), Scheme (1976), Common Lisp (1984), Racket (1994) и Clojure (2007) в порядке их появления. Далее сформулированы выводы о замеченных особенностях создания диалектов языка Lisp.

Гомоиконный конкретный синтаксис программы представляет программу в виде ее абстрактного синтаксического дерева (abstract syntax tree – AST), позволяющего применять автоматизированную генерацию распознавателей принадлежности программы языку программирования. Трансформационная абстрактная семантика отражает эквивалентность разных форм представления программ и данных, дающую основания для оптимизирующих преобразований программ. Приаппаратная системная прагматика вычислений подчинена требованиям эффективности и производительности кода программ, включая проблемы безопасности и надежности. Парадигма программирования отражает стиль мышления в процессе постановки задачи, способствующий продуктивному программированию ее решения. Взаимодействие синтаксиса, семантики, прагматики и парадигм можно рассматривать как логическую интерпретацию диалектных абстракций языка программирования, представление которой требует специальных форм, показывающих архитектуру языка.

Методика учитывает, что термин «язык программирования» в речевой практике понимается как «входной язык системы программирования, обеспечивающей доступ к определенным аппаратным средствам». Такое понимание потребовало специальных форм показа результатов анализа и сравнения, отражающих перемещение сквозных понятий на разные уровни реализации языка программирования. Поэтому, кроме сравнения конструкций языка программирования на уровне конкретного синтаксиса и абстрактной семантики, проанализированы структуры данных, пространства доступных процессов обработки данных

и дисциплины доступа к памяти в типовых языках программирования на уровне системной прагматики. Учтено, что понимание языка программирования всегда опирается на ряд известных, возможно неявных конструкций, необходимых для его реализации в системе программирования и воспринимаемых в практике как неотъемлемая часть языка, в реальности существующего как целостный комплекс, составляющие которого взаимосвязаны.

Долгоживущие языки программирования обычно расширяют ряд вычислительных возможностей и парадигм программирования подключением стандартных библиотек, пакетов, монад или выделением диалектов, повышающих продуктивность программирования. Диалект становится самостоятельным языком, наследуя особенности исходного языка программирования, слегка изменяя и дополняя их. Цель выполненного эксперимента – изучить особенности изменения конкретного синтаксиса, абстрактной семантики и системной прагматики языка программирования в диалектах и наследниках, показать особенности наследования конструкций уровня и синтаксиса, и семантики, и прагматики. При сравнении выделяются диалектные абстракции, работающие как метапонятия, смысл которых немного варьируется в диалектах при сохранении архитектуры языка и поддержанных им парадигм.

В статье приведены результаты сравнения языка Lisp с его успешными диалектами. Результаты представлены в форме, показывающей, что унаследовано, что отвергнуто, что изменено и чем дополнено. Выбор языка Lisp для первого эксперимента обусловлен не только четкостью и лаконизмом его описания [2], но и ростом интереса к функциональному программированию, регулярно происходящим при смене элементной базы и расширении сферы применения информационных технологий. Кроме того, Lisp можно характеризовать как язык программирования одновременно и низкоуровневый, и сверхвысокого уровня в зависимости от уровня решаемых задач. Lisp легко адаптируется к решению новых задач и изобретению лаконичных и эффективных конструкций, не всегда соответствующих шаблонам, навязываемым более популярными языками программирования. Такой спектр возможностей языка позволяет определять логическую интерпретацию сквозных понятий, показывающую общность решений, связанных с языком, – архитектуру языка.

Эксперимент выполнен на материале диалектов Pure Lisp, Scheme, Common Lisp, Racket и Clojure [2–8], появившихся с шагом в 10 лет. Анализ этих диалектов показал различие целей их создания и механизмов достижения целей при минимальных изменениях семантики и прагматики исходного языка Lisp. Более заметны изменения на уровне лексикона[1] и конкретного синтаксиса, что удобно для выделения диалектных абстракций. При сравнении языка Lisp с его диалектами уделено внимание своду принципов функционального программирования и расслоению языка программирования на базис, расширение, средства диагностики границ вычислимости и отладки программ, а также средства связи процесса вычислений с внешним миром.

Для оценки особенностей диалектов были использованы примеры реализации отдельных конструкций языка Lisp и его диалектов, проверенные на системе HomeLisp[2], платформе jdoodle.com[3] и других онлайн-компиляторах.

ФОРМЫ ПОКАЗА РЕЗУЛЬТАТОВ СРАВНЕНИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

Традиция описывать языки программирования и представлять их формальные определения сложилась во времена, когда каждый язык реализовывали автономно, начиная с прагматики решений уровня аппаратуры, полной реализации анализаторов синтаксиса и семантики языка программирования, возможно с разработкой своего формализма, типа расширенных форм Бэкуса – Наура (БНФ). Такие формализмы не претендовали на описание контекстнозависимых особенностей абстрактной семантики и системной прагматики из-за чрезмерного разнообразия новых архитектур и развития методов реализации языков программирования.

С тех пор в практике реализации новых языков программирования сложилась тенденция ограничиваться синтаксической надстройкой над существующими языками без пересмотра или уточнения решений уровня прагматики, ограниченной RISC-архитектурой[4], с небольшими вариациями семантики, что

[1] Под термином «лексикон» понимается множество имен доступных функций.

[2] http://homelisp.ru/

[3] https://www.jdoodle.com/

[4] RISC-архитектура – reduced instruction set computer

означает выделение небольшого числа известных стереотипов в этой сфере, почти не подверженных вариациям. Это позволяет представлять наследование конструкций между языками и выполнять логическую интерпретацию различий в языке программирования в терминах диалектной абстракции.

Основные трудности представления логической интерпретации связаны с тем, что описания языков программирования и их стандартов обладают слишком большим объемом (от 700 до 1500 страниц). Формализмы, используемые в описаниях языка программирования, представляют собственно конкретный синтаксис языка с неформальными пояснениями без показа границ наследования конструкций предшествующих языков и решений уровня абстрактной семантики и системной прагматики, описываемых средствами естественного языка. Сведения, полученные из таких источников, требуют проверки на реальных языках программирования, подверженных развитию.

В порядке эксперимента для показа деталей наследования на уровне конкретного синтаксиса между диалектами языка программирования предложено использовать специальное расширение форм Бэкуса – Наура [9], отражающее отношение наследования между понятиями разных диалектов и их одноименными определениями в предшественниках, частично дополненное показом особенностей некоторых понятий на уровне абстрактной семантики и системной прагматики (табл. 1). Примеры такого представления конструкций «S-выражение», «Форма», «Функция», «ленивые вычисления», структуры данных и особенностей «REPL-цикла»[5], полученных при сравнении диалектов языка Lisp, приведены в препринтах [22, 23]. В эксперименте удалось показать с помощью таких обозначений некоторые особенности абстрактной семантики и системной прагматики с небольшими комментариями.

Для более полного показа лаконичной формы пока не нашлось, одни и те же конструкции могут перемещаться из прагматики в семантику, из семантики в синтаксис – граница между синтаксисом, семантикой и прагматикой условна.

[5] REPL-цикл – название основного рабочего цикла, определяющего обработку программ, сокращение от Read Eval Print Loop.

Табл. 1. Расширение БНФ для показа результатов сравнения языков программирования

Формула	Примечание
Диалект: понятие	Объявление понятия в диалекте
Старое_понятие**.**Предшественник	Используется определение из предшествующего языка
Старое_понятие**!~**Шаблон**.**Предшественник	Из определения предшествующего языка исключаются фрагменты, соответствующие шаблону
Одноименное_понятие	Используется новое определение, полностью замещающее старое
Элемент …	Произвольное число вхождений элемента, возможно ни одного
_* «Symbol» [понятие]	Последовательность литер, кроме Symbol – продолжение после ошибки
⊥! «строка-диагноз» **[** понятие **]**	Сообщение диагноза с приемом дополнения в диалоге
Синтаксис Семантика Прагматика // Комментарий	Уровни определения (табл. 2)
[[Формулы над множествами]] // скобки для наглядности перехода с семантике вычислений	Семантические системы. Семантика вычислений в конкретном пространстве (табл. 3)
((Операции над состояниями памяти)) // скобки для наглядности перехода к прагматике изменения памяти	Системная прагматика. Дисциплина изменения состояний памяти (табл. 4)

Кроме показа наследования такие формулы позволяют определять продолжающие и диагностические грамматики. Понятия разных уровней могут входить в общую формулу (табл. 2).

Табл. 2. Формы для показа границ между уровнями понятий языка программирования

Формула	*Примечание*
Элемент ::= { **Синтаксис** ***Семантика*** Прагматика *// Комментарий* }	*Разные шрифты*
Элемент ::= **{** **Синтаксис** ***Семантика*** Прагматика *// Комментарий* **}**	*Разные уровни*

Строки табл. 2 выражают разноуровневые составляющие определения языка программирования привлечением разных шрифтов для понятий уровня **синтаксиса**, ***семантики*** и прагматики. **Синтаксис** – жирный шрифт, ***семантика*** – жирный курсив, прагматика – обычный подчеркнутый шрифт, *комментарий* – курсив. Более наглядно использовать индексы ($^{\text{верхний}}$, обычный, $_{\text{нижний}}$). Примеры так представленных различий в уровнях и границах вхождения в языки программирования структур данных диалектов языка Lisp приведены в препринте [23].

Не все важные особенности языка программирования удалось выразить такими компактными формами. При поиске форм для показа результатов сравнения языков программирования, удобных для оценки выразительной силы языка программирования, а также трудоемкости и продуктивности его реализации, эффективности и производительности программ, создаваемых на базе языка программирования, учтена зависимость от последовательности критериев принятия решений по декомпозиции программ, что не является однозначным, зависит от парадигм программирования и классов решаемых задач. Для диалектов языка Lisp последовательность критериев зависит от принципов функционального программирования (таких как универсальность данных, самоприменимость определений, равноправие и независимость параметров и единственность результатов функций, гибкость границ блоков памяти и неизменяемость хранимых значений). Технически в качестве основного критерия выбрана

семантическая декомпозиция определений языков программирования, позволяющая показывать различия и дистанцию в понятийной сложности между похожими семантическими системами[6], образующими язык программирования. Для лаконичного показа различий абстрактной семантики и системной прагматики предложенные обозначения немного различаются для таких категорий семантических систем, как вычисления, структуры данных, управление вычислениями и обработка памяти – они обладают разными шаблонами определения функций.

Семантическая система – это тройка **[[V, F, R]]**, где:

V – основное множество данных, возможно бесконечное;

F – набор операций, возможно принадлежащих множеству **V,** расширяемый программируемыми функциями;

R – варианты правил применения операций **F** к данным из **V**, возможно входящих в **F**, представимые как данные из **V**, возможно программируемые как функции.

Компактная форма взаимосвязей составляющих семантических систем функционального программирования выражается формулой

$$[[V, F, R]] \mid R \subset F \subset V$$

где «**R** является подмножеством **F** и F является подмножеством **V**» или «**R** включено в **F**, а **F** включено в **V**».

Такой формат семантики языка программирования, присущий функциональному программированию, поддерживает передачу опыта программирования в форме диалектов и пакетов со своими правилами их интерпретации. В процессе программирования новых функций, расширяющих **F**, возможна разработка новых вариантов **R** и новых семантических систем. Такие взаимосвязи между понятиями позволяют формализовать и развивать правила **R** применения операций **F** к данным **V**, включая кумулятивные (накопительные) эффекты между составляющими системы программирования. Представления операций и программируемых функций **F** включаются в основное множество **V**, а правило применения **R** операций **F** к данным **V** – не более чем одна из функций, возможно

[6] С. С. Лавров предложил понятие «семантическая система» как расширение понятия «алгебраическая система» заданием явного правила R применения операций к данным.

программируемая. Различные категории семантических систем (вычисления, структуры данных, управление вычислениями и обработка памяти) могут быть подчинены разным правилам применения **R**, требующим различных систем обозначений (табл. 3 и 4).

Эти обозначения позволяют показать различие в пространствах допустимых значений и особенности ограничений на представление и выполнение функций в разных семантических системах языка. Например, формула [[∃ { BSD[7] … } ∀ { etd[8] …]] задает пространство допустимых данных, устроенное как кумулятивная иерархия структур данных BSD над любыми значениями из множеств элементарных типов значений etd. На уровне абстрактной семантики языка программирования определение etd представляется как набор предикатов, распознающих принадлежность значения к конкретному типу или виду элементарных значений. Определение BSD кроме предикатов содержит набор конструкторов и деструкторов, связанных с неявными функциями доступа к памяти и с диагностикой ошибочных значений.

Табл. 3. Обозначения для показа различий в определении **V** – основного множества структур, видов и типов данных языка программирования.

[[Формула]]	*Пояснение: Формула для семантических систем заключается в двойные квадратные скобки*
∀etd	Область определения всех функций опирается на любые элементы множества типов данных etd[9]
∃ BSD etd	Область определения всех функций может использовать любые элементы кумулятивной иерархии[10] базовых структур данных из множества BSD[11] над элементами типов данных из etd
V ‖ S V ∩ S	Фильтрация, пересечение множеств для выделения подходящих
⊂ ⊃ U ∩ ∈ ∉	Операции над множествами

[7] Базовые структуры данных.

[8] Элементарные типы значений.

[9] Примеры etd: атом, number, string, metaD (метаданные) и др.

[10] Универсум фон Неймана**,** кумулятивная иерархия множеств.

[11] Примеры BSD: list, array, hash, set, structure и др.

[[Формула]]	*Пояснение: Формула для семантических систем заключается в двойные квадратные скобки*
: => λ (x y)	Отображение, переход, формат представления схемы функции
^S ’S ©S	Методы обработки форм: eval quote compile
≈ $\equiv_s$ $\equiv_w$	Эквивалентно
∉!	Ошибочное значение
⊥!	Неопределенное значение

Такие обозначения позволили выразить различие между пространствами допустимых данных, создания и обработки элементарных, встроенных и программируемых структур данных в соответствии с принципами функционального программирования [22, 23]. Предложенные формулы могут использоваться как уточнение анализатора текстов программ для оптимизирующей компиляции и проверки условий семантической корректности программ и компиляторов.

Табл. 4. Обозначения для показа механизмов обработки памяти в языке программирования

((Действие))	*Пояснение: Действия заключаются в двойные круглые скобки*
< U ; …>	Структура блоков памяти
V ↕ S V ‖ S	Из V выбираются такие, как S
Δ S	За исключением **S**
→ ↓ ↑ ← ↔	Операции над элементами памяти: → инициированные, ↓ запись, ↑ чтение[12], ← удаление, ↔ обмен данными.
!@	Чтение произвольного элемента с удалением из структуры данных
@	Адрес произвольного элемента памяти из структуры данных
from ! из	Выбор произвольного элемента памяти из структуры данных
+=	Пополнение блока памяти
±	Переход к другой дисциплине функционирования

[12] ↑: переменная **=>** пара: данное с адресом

((Действие))	*Пояснение: Действия заключаются в двойные круглые скобки*
∅	Пустое множество
#	Число элементов блока памяти или структуры данных
*****	Многократное повторение операции, может ни одного
~◊	Недостижимый из программы элемент памяти
(GC …)	Вызов мусорщика из программы
H VM GC	Блок памяти, виртуальная машина, мусорщик
Prog ⊂ Heap	Включение одного блока памяти в другой
Var ∉ call	Вхождение элемента в выражение или категорию функций

Эти обозначения могут соответствовать неявным действиям, сопровождающим вычисления над структурами данных. Например, дисциплина обработки памяти, определяемая формулой ((→ ↓ ↑* ←))**,** задает действия, сопровождающие применение локальной переменной, как рассредоточенную последовательность[13] неявных операций над памятью. Сначала разрешено завести элемент памяти (**→**), ссылка на который связывается с переменной. Потом в элемент памяти разрешается записать значение (**↓**). После этого записанное значение можно читать из памяти произвольное число раз (**↑***) в пределах локальной области видимости. После выхода из этой области следует удалить ссылку на элемент памяти (**←**), связь переменной с элементом памяти исчезнет, он становится недостижимым из программы. Перед выполнением операции над памятью возможна проверка, является ли операция допустимой в последовательности, определяющей дисциплину обработки памяти. При определении языка программирования на уровне системной прагматики в таких обозначениях можно задавать и другие условия корректности работы с памятью, неявно сопровождающей семантические функции. В результате общее определение языка программирования можно выразить как комплект проекций – схем, выражающих возможность присоединять ассоциированные определения абстрактной семантики и системной прагматики к определению конкретного синтаксиса для автоматической генерации компилятора.

[13] Рассредоточенная последовательность задает порядок выполнения операций, между выполнением которых происходят вычисления, определенные независимо.

Такие обозначения позволили при сравнении диалектов языка Lisp выразить различие между семантическими системами поддержки обработки памяти и хранимых в ней значений в соответствии с принципами гибкости границ блоков памяти, использующей функцию GC – вызов мусорщика, и неизменяемости хранимых значений, адреса которых достижимы из программы [23]. Общая схема определения приобретает вид

Синтаксис [⟦ ***Семантика*** ⟧] [**((** Прагматика **))**]

Конструкциям уровня абстрактной семантики и системной прагматики могут соответствовать разные шаблоны кодогенерации, включая функционально эквивалентные шаблоны для отладчика, компилятора и интерпретатора, представления которых требуют другой, специальной макротехники.

Такие формулы можно использовать как представление дисциплины работы с памятью, контролируемой и на этапе компиляции, и в процессе исполнения программы, что может способствовать обеспечению надежности и безопасности программ.

НЕМНОГО ИСТОРИИ ЯЗЫКА Lisp

Идеи Джона Маккарти (John McCarthy), воплощенные в языке Lisp, сразу вызвали ревнивую критику со стороны как программистов, так и математиков. Математиков смущала противоречивость некоторых построений с точки зрения классической математики, например различие контекстов определения, вызова и вычисления функций[14]. Программисты не могли смириться с отсутствием в языке привычной техники, начиная с «изменения состояний памяти», а также с непредсказуемо медленной обработкой списков в сравнении с быстрой обработкой векторов.

К середине 70-х годов XX в. Дана Скотт (Dana S. Scott) опубликовал конструктивную теорию, смягчившую критицизм математиков, построив первую непротиворечивую модель бестипового λ-исчисления[15]. Скептицизм программистов оказался более устойчивее. Например, общее мнение, что Lisp – это интерпретируемый язык, скорее всего, связано с тем, что реализации языка Lisp

[14] https://en.wikipedia.org/wiki/Funarg problem/ – статья о Funarg-проблеме.

[15] https://ru.wikipedia.org/wiki/ – непрерывность по Скотту.

обычно предоставляют диалоговый – интерактивный стиль работы с программой на базе REPL-цикла и не формируют файл с результатом компиляции, поэтому не заметно, что фрагменты программного кода компилируются по мере необходимости. Такое мнение не исчезло при появлении в 1976 г. диалекта Scheme, использующего совмещение чтения программы с ее полной компиляцией, но, сохраняющего диалог и добавляющего неявную эффективную векторную реализацию списков.

Уже в начале 60-х годов XX в. язык Lisp, включая интерпретатор и компилятор, был описан в виде формализма на самом языке Lisp. Lisp позволяет создавать программы, динамически порождающие код, выполнять любые системные трюки, строить виртуальные машины, специализированные системы, диалекты и пакеты, расширяющие язык. Такой потенциал языка Lisp дает ответ на вопрос: «НАСКОЛЬКО новые задачи компьютерной обработки информации отличаются от традиционных задач обработки чисел?». Основные тезисы ответа представлены следующими утверждениями.

– Любой информации можно дать символьное представление, числа – частный случай символьного представления, элементарные данные другой природы можно представить как атомы.

– Все понятия программирования можно рассматривать как функции или применение функций. Переменные, операторы или команды – не более чем разные категории функций.

– Эксперименты при разработке решений новых задач продуктивнее выполнять в диалоге на базе интерпретаторов. Компиляция полезна для достаточно отлаженных программ.

– Списки произвольной длины из элементов любой природы могут быть гомоиконными конструкциям как высокого уровня постановки задачи, так и низкого уровня системных решений. Вектора, множества, таблицы и другие структуры данных на этапе экспериментов можно моделировать с помощью списков.

Концепции языка Lisp со временем кристаллизовались как парадигма функционального программирования [13][16], хотя реализация языка изначально

[16] Обзор литературы о функциональном программировании – https://alexott.net/ru/fp/books/

поддерживает основные императивные черты ради привлечения опытных программистов и возможности повышать надежность и эффективность программ.

Для быстрого ознакомления с идеями языка Lisp Дж. Маккарти выделил семантический базис – диалект Pure Lisp, включающий в себя пять функций обработки списков (CONS, CAR, CDR, EQ, ATOM), четыре конструктора выражений и функций (QUOTE, COND, LAMBDA, LABEL)[17] и универсальную функцию EVAL, способную вычислить любое выражение, правильно представленное списком, что определяет границы вычислимости без использования семантики изменения состояний памяти, без глобальных переменных. Пары функций QUOTE и EVAL достаточно для поддержки разных схем вычислений, включая ленивые вычисления, оптимизации программ и любую макротехнику как основу метапрограммирования, определения интерпретаторов и компиляторов. Диалект Pure Lisp дал ответ на вопрос «КАКОЙ может быть методика обучения программированию на языке Lisp?». Ответ выглядит следующим образом:

– Выделить краткую базовую семантику, достаточную для определения остальных конструкций языка (выражения, ветвления, рекурсивные функции).

– Дать примеры их символьного представления и применения при решении знакомых задач.

– Показать типовые решения некоторых задач с помощью отображений, ленивых вычислений и метапрограммирования.

– Привести определения интерпретатора и компилятора для изучаемого Pure Lisp на уровне калькулятора, а потом расширить определение Lisp-калькулятора до обобщенного интерпретатора и компилятора.

Такая система понятий и средств обработки данных позволила поддержать все семантические и прагматические принципы чисто функционального программирования, позднее реализованного как основная парадигма программирования на базе ленивых вычислений в языках ML (1973), Hope (1980), Haskell (1990) и др.

Дж. Маккарти ожидал, что оставшиеся проблемы организации вычислений будут решены в более поздней версии, условно названной Lisp 2, в которую

[17] QUOTE – блокировка, COND – ветвление, LAMBDA – безымянная функция, LABEL – именование.

планировал включить обработку многомерных векторов, сопоставление с образцами и организацию параллельных вычислений [11]. Рассказывая о языке Lisp, Дж. Маккарти подчеркивал, что, экспериментируя, программист может изменять в языке Lisp все что угодно, кроме константы Nil[18] [12]. К 1962 г. были готовы версия Lisp 1.5 и описание реализации системы, ставшей преемником самого раннего языка Lisp [2]. Это описание языка стало основой для создания Lisp-систем как в США, так и за их пределами, в нашей стране на БЭСМ-6, ЕС ЭВМ, СМ-4 и других машинах[19]. Сложные данные языка Lisp на уровне синтаксиса выглядят как списки элементов любой природы, хотя неявно в системе программирования на уровне системной прагматики языка поддержаны и другие структуры данных, такие как вектора, множества, хэш-таблицы и изменяемые поля, ставшие явными в более поздних диалектах. Хэш-таблица применяется для идентификации атомов, множество моделируется как список различимых параметров функции, размеченное множество используется при организации списков свойств атомов и пространств имен, изменяемые поля используются при организации рекурсии и оптимизации отложенных или ленивых вычислений, а вектора – для сопряжения со встроенными машинными процедурами.

В начале 60-х годов XX в. Питер Ландин (Peter J. Landin) в работе о λ-исчислении [12] ввел понятие "call-by-name"[20], использованное в описании языка Algol-60. В 1964 г. он предложил машину SECD – виртуальную и/или абстрактную машину, предназначенную для использования в качестве целевого языка (бэкэнд) при компиляции языков функционального программирования [12]. Вскоре появился Lispkit – реализация чисто функционального диалекта Pure Lisp с лексической областью видимости, разработанного в качестве испытательного и учебного стенда при изучении концепций функционального программирования, включая ранние эксперименты с ленивыми вычислениями [13, 14]. Полученные компилятор и виртуальная машина были легко переносимы. Важный

[18] Это утверждение Джон Маккарти произносил в конце декабря 1968 г. в кабинете А. П. Ершова в Новосибирске в цикле лекций, посвященных языку Lisp и верификации программ.

[19] https://www.computer-museum.ru/histsoft/lisp_sorucom_2011.htm

[20] «вызов по имени»

в контексте ленивых вычислений термин «мемоизация» был придуман Дональдом Мичи в 1968 г. [15]. В 1971 г. Кристофер Стрэйчи предложил термин "call-by-need"[21], предшественник термина «ленивые вычисления» [16].

В 1974 г. в Xerox началась разработка аппаратуры и системы машинных команд для аппаратной реализации языка Lisp. Язык Scheme был разработан в 1976 г. в MIT в рамках проекта по созданию Lisp-машин [3]. Появилось доказательство, что так называемая «неэффективность языка Lisp» обусловлена не свойствами языка, а особенностями компьютеров и методов реализации языка программирования. Практически одновременно термин "Lazy evaluation" (ленивые вычисления) был введен в статье "Programming Languages and Their Definition" Кристофера Стрейчи [17], в статье "A Lazy Evaluator" Питера Хендерсона и Джеймса Х. Морриса [16]. В 1978 г. был представлен язык программирования Lazy ML, который стал первым языком, основанным на парадигме ленивых вычислений. В 1978–1979 гг. был разработан язык программирования Норе[22] в Эдинбургском университете Великобритании. Этот язык оказал значительное влияние на Haskell, представленный программистскому сообществу в 1987 г. с целью притормозить создание новых языков функционального программирования, их новизна затрудняет работу экспертов при решении вопросов о приоритете публикуемых результатов.

В 1984 г., через 8 лет после Scheme, появился диалект Common Lisp – мультипарадигмальный язык общего назначения, дополняющий традиционные динамические решения языка Lisp механизмом статического связывания переменных и раздельных пространств имен, специальных средств программирования макросов, функционалов, пакетов и лексических замыканий функций [4]. В 1995 г. Common Lisp был стандартизован ANSI.

В последние годы особое внимание привлекает диалект Racket (ранее – PLT Scheme), созданный в 1994 г. как платформа языково ориентированного программирования [5]. Это симптом перехода практики программирования

[21] «вызов по необходимости»

[22] *Филд А., Харрисон П.* Функциональное программирование. Пер. под редакцией В. А. Горбатова. М. Мир, 1993, 638 с. Содержит описание различных вариантов «мусорщика».

от накопления правильности программ на уровне библиотек к уровню создания диалектов и проблемно ориентированных языков.

В 2007 г. появился Clojure – современный диалект языка Lisp, предлагающий решения по верификации программ и параллельному программированию [6–8].

ДИАЛЕКТ Scheme ДЛЯ ЭФФЕКТИВНОЙ РЕАЛИЗАЦИИ

Язык Scheme был разработан в 1976 г. в MIT в рамках проекта по созданию Lisp-машин [3]. Scheme ответил на вопрос «КАК сделать функциональное программирование столь же эффективным как императивное?». Ответ включал следующее:

– Разграничить универсальность символьных вычислений отказом от представления значений любой природы, а списки свойств символа оставить только для переменных и функций.

– За основу системных структур данных взять вектора, а списки использовать при необходимости как синтаксическое расширение в отдельном модуле.

– Чтение списочного представления программы, фактически являющегося представлением ее абстрактного синтаксического дерева, совместить с принудительной компиляцией, а универсальную функцию EVAL (интерпретатор) вынести из базиса во вспомогательный модуль.

– От рецептов отложенных функциональных аргументов, формируемых в точке вызова функции, перейти к формированию замыканий в точке определения функции, заодно и макротехнику ограничить выполнением на этапе компиляции, что сводит ее потенциал к привычным возможностям препроцессоров во многих языках программирования. При компиляции автоматически выполнять оптимизацию рекурсий, сводимых к циклам.

– Смягчить неизменяемость данных, опираясь на унификацию присваиваний и определений функций, выполненную к тому времени при разработке языка Algol 68, а заодно и разрешить изменять значения системных свойств символа (define = set![23]).

[23] set! работает только на ранее введенных символах, define на любых.

Язык Scheme заимствовал терминологию и синтаксис языка Lisp, несколько изменив смысл ряда понятий и сузив трактовку почти всех принципов функционального программирования. Из характерных особенностей языка Lisp в языке Scheme поддержана примерно треть. Самым заметным отклонением от языка Lisp является введение булевых значений #f и #t вместо использования константы Nil или пустого списка в качестве значения «ложь»[24]. Переход к векторам пошатнул позиции пустого списка и атома Nil – реализация пустых векторов в те годы не практиковалась. Принудительная компиляция программы, теряющая исходное абстрактное синтаксическое дерево, затрудняет возможности динамической оптимизации программ и процессов. Такая компиляция устраняет повторную интерпретацию программ, что является оптимизацией многократно используемых программ.

Можно считать, что так выделилось минимальное ядро грамматики языка Lisp. Теперь существуют реализации Scheme на JVM.

Common Lisp ДЛЯ ПРОИЗВОДСТВЕННОГО ПРИМЕНЕНИЯ

Common Lisp был разработан в начале 80-х годов XX в. с целью объединения полезных механизмов большого числа разрозненных диалектов языка Lisp [4]. Common Lisp часто противопоставляют языку Scheme – это два самых популярных диалекта Lisp. Scheme предшествовал Common Lisp и исходил не только из той же традиции Lisp, но и от тех же разработчиков. Гай Стил, вместе с которым Джеральд Джей Сассман разработал Scheme, возглавлял комитет по стандартизации Common Lisp, в котором было преодолено сужение потенциала языка Lisp. Сохраняя и восстанавливая основные концептуальные решения языка Lisp, диалект Common Lisp их дополняет, расширяя методы формирования пространств допустимых процессов.

[24] Вопреки предостережению Джон Маккарти: «Программист может изменять в языке Lisp все что угодно, кроме константы Nil». Математиков смущало, что Nil одновременно и атом, и список. В теории удобнее, чтобы любое данное принадлежало ровно одному типу.

Common Lisp иногда называют Lisp-2[25], а Scheme – Lisp-1, имея в виду использование раздельных пространств имен для функций и переменных[26]. CLISP проводит различие между временем чтения, временем компиляции, временем загрузки и временем выполнения программы и позволяет пользовательскому коду также учитывать это различие при выборе желаемого типа обработки программы на нужном этапе. В системе CLISP реализован пакет CLOS, дающий полную поддержку популярной в производстве парадигмы ООП.

Этот диалект резко расширил сферу производственного применения языка Lisp, используя на уровне лексикона семантику доступа к многомерным матрицам, хэш-таблицам, программируемым структурам данных, подобным структурам в языке C, изменяемым полям и мультизначениям, удобными для моделирования независимых потоков. Все это внешне представляется как списки, но реализовано как эффективные структуры данных, доступные через функции. Массивы могут содержать любой тип значений в качестве элемента, смешивать разные типы в одном массиве или могут быть специализированы, содержать только определенный тип.

Common Lisp ответил на вопрос: «ЧТО может дать функциональное программирование программной индустрии?». Ответ включает следующие дополнения:

– Универсальность символьной обработки и структур данных неограниченного размера дополнена средствами обработки конечных чисел и структур данных, типичных для большинства языков программирования и приложений.

– Компиляция отдельных функций допускает и компиляцию полной программы без потери исходного списочного представления абстрактного синтаксического дерева.

– Самоопределение в форме рекурсии обогащено разнообразием схем циклов, превосходящих по возможностям типовые схемы.

– Гибкость распределения памяти с возможностью программировать вызов «сборщика мусора» дополнена средствами выяснять время, даты и этапы

[25] Ассоциация с проектом Lisp-2 Дж. Маккарти [11].

[26] Язык Lisp распался на два семейства – Lisp-1 и Lisp-2, различаемые по отношению к статике и динамике, возможностям применения списков свойств и роли атома NIL = ( ) в логике управления вычислениями.

работы программы, чтобы прогнозировать целесообразность применения тех или иных методов.

– Неизменяемость хранимых значений уточняется введением неявного понятия «поле» для работы с изменяемыми системными свойствами символа.

– Для функций, потребляющих много памяти, предоставляются их деструктивные аналоги, приспособленные к более эффективной обработке данных (conc-*nconc, subst-nsubst, reverse-nreverse, union-nunion, mapcar-mapcon и др.*)

– Единственность результата функции расширяется на мультизначения, поддерживающие переход к многозначным функциям и организации параллельных вычислений.

Введены понятия «поле», «пакет» и «мультизначение». Основное отличие от языка Lisp связано с разделением пространств имен в зависимости от их назначения и включения в разные пакеты для решения отдельных классов задач.

Common Lisp поддерживает средства динамического анализа и использовался в разработке автоматизированных средств проверки доказательств теорем (ACL2) и систем компьютерной алгебры (Аксиома, Maxima).

СРЕДА Racket ДЛЯ СОЗДАНИЯ НОВЫХ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

Очередная версия учебной среды программирования на языке Scheme, разрабатываемая с 1995 г. в Университете Дьюка для обучения созданию, разработке и реализации новых языков программирования, в 2010 г. получила название Racket [5, 20]. Это означает переход интересов от продуктивности программирования к продуктивности разработки компиляторов.

Образовательная направленность повлияла на общую структуру языка Racket как систему диалектов, соответствующих уровням обучения. Это заодно позволило в реализации языка сохранить решения языка Scheme, принятые под прессом компьютерных характеристик середины 70-х годов XX в., и дополнить системную поддержку языка Racket в соответствии со значительно усовершенствованными возможностями современной элементной базы и новыми требованиями ИТ.

Разработчики диалекта Racket выявили ключевые недостатки Scheme, затрудняющие разработку крупных и надежных систем, а именно отсутствие мо-

дульной системы, слабую поддержку обработки исключений, метапрограммирования и расширяемости, ограничения в типизации и структурах данных, отсутствие способов для построения безопасных и масштабируемых систем. Для преодоления таких недостатков Racket был создан как диалект языка Lisp и используется для реализации языков программирования, компиляторов и интерпретаторов, образовательных систем и платформ, языков для обучения и преподавания, включая обучающие и экспериментальные языки. Оставаясь наследником Scheme, Racket делает ставку на практичность, расширяемость и богатство инструментов, уделяя больше внимания удобству использования и обучения, поддержку метапрограммирования и инструментальных средств для разработки языков программирования. Racket включает макросы на этапе и компиляции, и выполнения с удобными функциями для разработки и отладки. Поддержаны диалект Scribble – язык разметки для документации и ряд диалектов, связанных с основными парадигмами программирования, проблемноориентированными языками (*domain-specific language – DSL*) и приложениями ИТ.

Производительность Racket обеспечена JIT-компилятором и механизмом «сборки мусора» с поддержкой поколений объектов. Включена поддержка мелкозернистого параллелизма. Имеется учебная версии Minimal Racket без пакетов, поддержка байт-кода и JIT-компиляции для архитектуры ARM[27], а также быстродействующий Typed Racket и другие диалекты. Разработана собственная виртуальная машина. В экспериментах по разработке разных версий Racket обнаружилось, что компиляция не всегда повышает производительность программ, но может способствовать продуктивности программирования[28]. Система макросов в Racket используется для создания полных языковых диалектов, затрагивая семантику. Racket отвечает на вопрос «КТО будет определять языки программирования в будущем?».

Ответ достигается следующими решениями:

– Универсальность символьных вычислений распространена на приобретение профессиональных навыков, включающих разработку документации, что

[27] Архитектура ARM (*Advanced RISC Machine)* – усовершенствованная RISC-машина для мобильных устройств.

[28] Сотрудники фирм-разработчиков компиляторов утверждают, что необходимость ожидать результат компиляции дает им защищенную нишу времени для продумывания программ.

соответствует предложенной Д. Кнутом парадигме литературного (грамотного) программирования (literate programming)[29].

– Среди диалектов выделены языки, соответствующие уровням способностей, навыков и знаний студентов (Minimal Racket, Racket, Lazy Racket, Typed Racket и др.).

– Независимость параметров поддержана механизмом сопоставления с образцами, достаточными для представления грамматик с переводом.

– Самоопределение и рекурсивные функции подкреплены практикой применения генерации лексеров/парсеров, а также определением языков на уровне абстрактного синтаксического дерева.

– Гибкость распределения памяти сопровождается средствами рефакторинга, тестирования и измерения производительности кода.

– Единственность результатов функции расширена средствами организации асинхронных процессов, подразумевающих мультизначения.

Racket примерно на треть наследует решения языка Scheme и на две трети возвращается к исходным решениям языка Lisp. Самое заметное отличие диалекта Racket от семантики языка Lisp – использование специального булева значения #f в качестве значения «ложь» вместо пустого списка ( ) или атома Nil. Хотя формально язык Racket называют диалектом языка Scheme, по своим особенностям он ближе к Common Lisp.

Clojure – НОВЫЕ ГОРИЗОНТЫ ИТ

Современный диалект языка Lisp Clojure появился в 2007 г., разработан Ричем Хикки (Rich Hickey), независимым разработчиком ПО, ранее разработавшим dotLisp в рамках проекта **.**NET Framework. Clojure наследует особенности языка Lisp, обеспечивающие гибкое и мощное метапрограммирование, поддерживает синтаксическое расширение [6–8] и надежную семантику, дополненную механизмами системной прагматики для программирования приложений на базе

[29] http://www.literateprogramming.com/ – методика профилактики кризиса ухода исполнителя и разбухания описаний, создаваемых техническими писателями. Не исключено, что методика не стала массовой из-за высокого темпа развития ИТ, опережающего созревание речевой практики. Возможно, Д. Кнут хотел привлечь внимание к тому, что искусство программирования основано и на владении естественным языком.

распределенных и параллельных процессов.

В этом языке определения функций могут неформально сопровождаться пред- и постусловиями, что помогает проверке утверждений об аргументах и полученных результатах, а также позволяет при тестировании проверять инварианты функций. Clojure, как язык программирования, не предоставляет встроенных методов верификации программ, но для обеспечения корректности программ использует различные подходы и инструменты, включающие средства типизации и спецификации (clojure.spec, malli), динамический контроль данных и тестирование (clojure.test, test.check), помогающие обнаруживать неожиданности. Поддержана интеграция с внешними инструментами для формальной верификации с возможностью автоматического тестирования свойств, такими как Rosette, Z3, SMT (satisfiability modulo theories) или ACL2, проверки типов данных (clj-kondo, Eastwood), а также статические анализаторы для проверки безопасности кода или соблюдения определенных стандартов.

Динамическая типизация означает, что, кроме статической проверки типов переменных на этапе компиляции, проверяются типы данных, точно известные во время выполнения, что существенно повышает надежность и безопасность вычислений. Заодно это обеспечивает гибкость и быструю разработку, так как не требует явного объявления типов переменных. Clojure уделяет большое внимание тестированию (clojure.test, test.check, midje, kaocha, Unit Testing, Integration Testing, Property-based testing), вызывающему у практиков больше доверия, чем верификация.

Статический анализ выявляет потенциальные ошибки, нарушения стиля кодирования, неиспользуемый код и константные вычисления (эквивалент чисто функционального программирования), поддерживает автодополнения для более раннего обнаружения ошибок. Доступны библиотеки, позволяющие определять контракты (core.contracts, lucid.policy) для функций, и автоматическая генерация документации. Возможна явная реструктуризация структур данных, работающая с любой последовательностью, включая:

- списки, вектора и последовательности Clojure;
- любые коллекции, реализующие java.util.List (например, ArrayLists и LinkedLists);
- Java-массивы;

- строки, реструктурированные как списки символов;
- списки аргументов функций.

Реструктуризация означает переход от ранее созданной структуры данных к структуре с другим методом доступа при сохранении ее наполнения[30], допускается использование подчеркивания «_» для обозначения игнорируемой позиции. Исходная структура сохраняется в соответствии с принципом неизменяемости хранимых значений. Реструктурированная последовательность аргументов функции позволяет вместо имен связанных переменных использовать нумерацию параметров, список которых можно рассматривать как вектор[31]. К первому аргументу функции можно обращаться, просто используя «%».

Параллельное программирование использует транзакционную память, как в базах данных, а также агентов и разные виды указательных переменных. Поддержаны ленивые последовательности, вспомогательные процессы и введено несколько неявных понятий для поддержки параллелизма и программирования своих структур данных с учетом проблем надежности и безопасности.

В качестве компромисса между идеями чисто функционального программирования и необходимостью изменения состояний при организации параллельных процессов введены указательные переменные – атомы из уникальных указателей на списки свойств атома превращаются в указательные переменные и рассматриваются как системные структуры, обеспечивающие разные дисциплины доступа к памяти и стратегии многопоточности (atom, ref, agent).

Clojure отвечает на вопрос «ГДЕ новые горизонты, в освоении которых помогает продуктивность и моделирующая сила языка Lisp?». Ответ опирается на следующее.

– Универсальность символьных вычислений распространена на явный конкретный синтаксис отображений, множеств и векторов. Введены метаданные – кодированный аналог списка свойств, который может быть связан со значением или указательной переменной.

– Можно синхронизовать выполнение потоков: откладывание, ожидание, обещание, передача, готовность, блокировка (delay, future, promise, deliver, is-

[30] Похоже на реорганизацию векторов в языке APL.

[31] Подобно некоторым языкам заданий и макропроцессоров.

done?, deref), контролировать ход вычисления, а также представлять эффективные формы циклов, не использующие стек.

– Неизменяемость данных при необходимости изменений превращена в транзакционную память, как в базах данных, поддержаны агенты и разные виды динамических и указательных переменных.

– Единственность результата функции дополнена возможностью проверки утверждений об аргументах и полученных результатах, при тестировании можно проверять инварианты функций и использовать внешние методы верификации программ.

Диалект Clojure наследует примерно 80% особенностей языка Lisp, уточняет ряд его решений для удобства представления параллельных процессов и дополняет его заметным комплектом средств, соответствующих современной элементной базе и поддерживающих отладку взаимодействующих процессов и удостоверение правильности программ. Самое заметное расширение связано с понятием «атом». Атом из неявного уникального указателя на список свойств атома стал явной указательной переменной, позволяющей поддерживать различные дисциплины обработки памяти, возникающие в разных моделях параллельных вычислений, разнообразие которых непредсказуемо велико. Кроме того, механизм реструктуризации данных распространен на список аргументов, что расширяет форматы определения функций, позволяет отказываться от использования имен связанных переменных[32], что удобно при генерации машинного кода.

ИТОГ СРАВНЕНИЯ ДИАЛЕКТОВ

В ходе эксперимента выяснилось, что результаты сравнения языков программирования следует показывать декомпозированными по отдельным особенностям и конструкциям языка программирования. Обозначения из табл. 1–4

[32] Интересно, что в свое время основатель нашей математической школы Н. Н. Лузин не одобрял термин «связанная переменная», он пояснял, что это вообще не переменная, потому что ее имя можно заменять на другое – смысл формулы не изменится. Своего термина не предложил. *Лузин Н. Н.* Интеграл и тригонометрический ряд. Изд. 2-е. М.: Гостехиздат, 1951.

оказались достаточными для показа наиболее очевидных синтаксических, семантических и прагматических различий между рассмотренными диалектами, причем с выражением отношения наследования. Удалось показать происходившее при создании диалектов изменение особенностей поддержки семантических принципов «универсальность», «самоопределение функций» (рекурсия), «независимость и равноправие параметров функций», «единственность результата функции», а также форматов ветвлений, ленивых вычислений и REPL-цикла. Кроме того, показано произошедшее изменение особенностей поддержки принципов «гибкость границ блоков памяти» и «неизменяемость хранимых в памяти значений» [23].

Отмечая различие в целях создания диалектов языка Lisp, можно заметить, что рассмотренные диалекты обладают стабильным системным ядром, использующим конкретный комплект структур данных и механизмов их обработки. Показано, что вариации структур данных и значений сводятся к пересмотру границ между лексиконом, синтаксисом, семантикой и прагматикой языка, к различию возможностей периода компиляции, выполнения и отладки программы и приоритетов между областями видимости символов, а также к вариантам представления значения «ложь» и расширению понятия «атом» от уникального указателя на список свойств атома до понятия «указательная переменная». Системные решения по обработке структур данных в новых диалектах из неявных уровнях системной прагматики становятся доступными сначала с помощью функций на уровне лексикона абстрактной семантики, затем получают представление на уровне конкретного синтаксиса. Это не нарушает функциональную эквивалентность программ, абстрактное синтаксическое дерева всех диалектов имеет подобное списочное представление. Семантика вычислений в диалектах по-разному взаимодействует с прагматикой изменения состояний памяти, что проявляется, как правило, в именовании функций и выборе границ изменяемых данных.

Принципы и понятия функционального программирования в диалектах языка Lisp уточнялись в зависимости от роли продуктивности программирования и критериев качества программ в разных областях приложения. Для Scheme – это эффективность и сохранение привычки к присваиваниям и векторам, для

Common Lisp – разнообразие структур данных, для Racket – создание специализированных диалектов, освобождающих от нагромождения библиотек, для Clojure – освоение многопроцессорных комплексов и распределенных информационных систем. Более подробно результаты сравнения представлены в препринтах [22, 23].

Разнообразие целей и решений, принятых в рассмотренных диалектах, позволило сформулировать особенности диалектного абстрагирования семантики вычислений от семантики изменения состояний памяти. Каждый из диалектов произвел определенное уточнение особенностей функционального программирования без отказа от его принципов, впервые поддержанных в реализации языка Lisp. Семантические и прагматические принципы функционального программирования (универсальность представления данных и программ, самоприменимость, равноправие параметров и единственность результатов функций, гибкость границ памти и неизменяемость хранимых значений) дополнились конструкциями ветвлений для представления частичных вычислений, ленивых вычислений для управления временем вычислений и REPL-циклом[33] для поддержки удобной отладки программ.

Гомоиконный конкретный синтаксис поддерживает универсальность представления данных и программ с помощью общих структур данных, удобных для самоопределения рекурсивных функций и структур данных, границы которых могут быть заданы как частичные функции с помощью ветвлений и циклов. Начиная с символьных представлений с помощью S-выражений языка Lisp и списков, достаточных для моделирования любых структур данных, диалекты предложили конкретный синтаксис для векторов, множеств и хэш-функций, присутствовавших на уровне системной прагматике:

(S-выражение …) *// список через пробел в языке Lisp.*
(S-выражение «**.**» S-выражение) *// пара в языке Lisp.*
((*S-выражение «.» S-выражение*) …)
// ассоциативный список *в языке Lisp.*
(*индикатор S-выражение* …)
// список свойств атома *в языке Lisp.*

[33] REPL-цикл — Read Eval Print Loop.

```
[S-выражение … ]
        // группировка или вектор в диалекте Scheme.
[S-выражение … ]
        // последовательность в диалектах Racket и в Clojure
#{S-выражение ... }   // множество в диалекте Clojure
{ ( ключ => значение ) … }
    // хэш-таблица или ассоциативный список в диалекте Clojure.
```

На уровне трансформационной абстрактной семантики существуют разные определения функций и представления форм, результаты которых совпадают на одинаковых комплектах аргументов в одинаковых контекстах. Это дает основания для оптимизирующих эквивалентных преобразований программ, включая исключение константных (чисто функциональное программирование) или дублирующих вычислений, перестановочность параметров и аргументов, вынесение подвыражений в аргументы, преобразование рекурсий в циклы, отложенные или ленивые вычисления, а также различные средства проверки правильности программ, предложенные в диалекте Clojure. Развитие вариантов REPL-цикла при отладке программ показывает целесообразность использования частичной или полной компиляции наряду с интерпретацией без потери исходного кода программы. Такие варианты предложены в диалектах Scheme, Common Lisp и Racket.

На уровне системной прагматики поддержка принципов функционального программирования использует исключения, обработка которых необходима для обеспечения продуктивности программирования и производительности программ. В динамике возникает переключение дисциплины функционирования памяти и вызов вариантов мусорщика с оптимизацией. Поддержка параллелизма и асинхронности привела в диалекте Clojure к расширению понятия атом до указательной переменной, а требование производительности программ повлекло поддержку метаданных и транзакций.

ЗАКЛЮЧЕНИЕ

Проведенные исследования форм для представления и обзора результатов сравнения наиболее популярных диалектов языка Lisp показали возможность лаконичного показа наследования их особенностей на уровне синтаксиса,

семантики и прагматики. Основная причина проведения такого эксперимента, а также поиска обозримых форм и выбора кратких обозначений связана с разработкой методик оценки продуктивности языка программирования и программируемых решений в отличие от непосредственного измерения эффективности и производительности программ. Произошло выделение понятия «диалектные абстракции», удобного при декомпозиции определений языка программирования на автономные составляющие, и понятия «логическая интерпретация», допускающего независимое варьирование и развитие сквозных понятий в процессе эволюции ИТ с сохранением общей архитектуры языка.

Сравнение диалектов языка Lisp показывает медленное смягчение программистского скептицизма в отношении к принятым в языке Lisp решениям и парадигме функционального программирования по мере прогресса элементной базы и развития ИТ, что видно по созданию новых языков программирования, рекламирующих включение механизмов функционального программирования как важное преимущество.

Полученные результаты образуют основу для определения номенклатуры семантических систем языков функционального программирования. Следует отметить не только новые диалекты языка Lisp, но и выпуски их реализаций – 25 августа 2025 г. выпущена очередная реализация Armed Bear Common Lisp (ABCL) [25].

При измерении мощности языков программирования как характеристики пространства доступных процессов вычислений самыми мощными являются языки ассемблера. Языки более высокого уровня, даже языки управления заданиями в операционных системах, теряют часть такого пространства ради удобства представления процессов обработки данных и управления ими. Такая потеря отчасти компенсируется моделированием, влекущим снижение эффективности ради продуктивности. Диалекты Scheme, Common Lisp, Racket и Clojure обладают реализацией на JVM, что говорит об их равномощности, они предоставляют одно и то же пространство процессов вычислений.

Сравнение диалектов Scheme, Common Lisp, Racket и Clojure, последний из них появился в 2007 г., показывает, что в них сохранены основные возможности языка Lisp, системные структуры данных, базовые принципы функционального программирования и понятия с определенными вариациями на злобу дня.

Авторы этих диалектов четко называют свои диалекты вариантами языка Lisp. Появление названий “Racket” и “Clojure” не отменяет роль так названных диалектов в успешной адаптации языка Lisp к новым поколениям программистов и новым возможностям аппаратуры. Знакомство с языками функционального программирования, наследующими идеи языка Lisp, такими как Sisal, F# и Haskell [19, 20, 26, 27], дает достаточное основание рассматривать Lisp как базовую математику функционального программирования. Результаты их анализа выходят за пределы настоящей статьи.

Семейство Lisp теперь является одним из старейших и наиболее влиятельных семейств языков программирования, его мощность в различных источниках, начиная с Википедий, оценивается от пятисот до тысяч языков программирования и диалектов. Кроме того, следует учесть языки функционального программирования, наследующие основные идеи языка Lisp, они постоянно разрабатываются, улучшаются и появляются новые, их число также оценивается в сотни или тысячи. Во многих источниках Lisp называют чемпионом по числу диалектов и наследников, хотя встречаются и утверждения, что сравнимое число диалектов имеется у языков C/C++, Bash, Perl, Python, JavaScript, BASIC, Forth, SQL, Fortran, Pascal, Ada, Assembler.

Следующий эксперимент по отладке методики анализа и сравнения языков программирования предполагается выполнить на материале языков функционального программирования, таких как Erlang, Sisal, F# и Haskell, рассматриваемых как наследники языка Lisp. Кроме того, интересно показать результаты сравнения представителей других долгоживущих семейств языков программирования, в первую очередь это Fortran и C, сохранивших значимость до наших дней. Отдельная задача – сравнение наших ЯП с зарубежными аналогами.

Благодарности

Автор искренне благодарен Андрею Валентиновичу Климову за ценные рекомендации по улучшению стиля изложения и поиску форм для представления результатов сравнения ЯП, Николаю Вячеславовичу Шилову, Игорю Сергеевичу Ануурееву и Борису Леонидовичу Файфелю за интерес к языку Lisp и стимулирующие вопросы.

СПИСОК ЛИТЕРАТУРЫ

1. *Городняя Л.В.* О представлении результатов анализа языков и систем программирования. Научный сервис в сети Интернет: труды XX Всероссийской научной конференции (17–22 сентября 2018 г., г. Новороссийск). М.: ИПМ им. М.В. Келдыша, 2018.

2. *McCarthy J. Abrahams P. W., Edwards D. J. et al.* LISP 1.5 Programming Manual. The MIT Press, Cambridge, 1963. 106 p.

3. *Dybvig K.R.* The Scheme Programming Language. https://www.scheme.com/tspl4/

4. *Graham P.* ANSI Common Lisp. Prentice Hall, 1996. 432 p.

5. The Racket Reference. https://docs.racket-lang.org/reference/

6. Clojure Programming. OReilly.com. Retrieved 2013-04-30. https://cdn.oreillystatic.com/oreilly/booksamplers/9781449394707_sampler.pdf

7. *Omm A.* Введение в Clojure. https://alexott.net/ru/clojure/clojure-intro/

8. Differences Clojure with other Lisps. https://clojure.org/reference/lisps/

9. *Backus J.W.* The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference // Proceedings of the International Conference on Information Processing. UNESCO. 1959. P. 125–132.

10. *Backus J*. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs // 1977 ACM Turing Award Lecture, p. 621–641.

11. *Mitchell R.W.* LISP 2 Specifications Proposal. Stanford Artificial Intelligence Laboratory Memo No. 21, Stanford, Calif., 1964.

12. *Лавров С.С., Силагадзе Г.С.* Входной язык и интерпретатор системы программирования на базе языка ЛИСП для машины БЭСМ-6. М.: ИТМ и ВТ АН СССР, 1969.

13. *Landin P.J.* The Mechanical Evaluation of Expression // Comput. J. 1964. Vol. 6, No. 4. P. 308–320. https://doi.org/10.1093/comjnl/6.4.308

14. *Хендерсон П.* Функциональное программирование. Применение и реализация. М.: Мир, 1983. 349 с.

15. *Henderson P., Jones G.A.; Jones S.B.* The LispKit Manual. University of Oxford Computing Lab. 1983.
https://github.com/hanshuebner/secd/tree/master/lispkit/LKIT-2

16. *Michie D.* 'Memo' Functions and Machine Learning" (PDF). Nature. 1968. Vol. 218 (5136), P. 19–22. Bibcode:1968Natur.218...19M.
https://doi.org/10.1038/218019a0. S2CID 4265138

17. *Strachey Ch.* Fundamental Concepts in Programming Languages // Higher-Order and Symbolic Computation. 2000. Vol. 13, No. 1–2. P. 11–49.[34]

18. *Henderson P., Morris JH.* A lazy evaluator. Symposium ACM Sigact-Sigplan sur les principes des langages de programmation // DBLP, Proceedings of the 3rd ACM SIGACT-SIGPLAN symposium on Principles on programming languages (POPL), 1976. P. 95–103.

19. *Душкин Р.В.* Функциональное программирование на языке Haskell. М.: ДМК Пресс, 2008. 544 с., ил.

20. Официальный сайт языка Haskell – "О языке"
http://haskell.org/aboutHaskell.html

21. From PLT Scheme to Racket. Racket-lang.org. Retrieved 2011-08-17. https://docs.racket-lang.org/guide/intro.html Welcome to Racket

22. *Городняя Л.В.* Lisp и его диалекты. Новосибирск, препринт, 2025. https://www.iis.nsk.su/repository/gorod.14408

23. *Городняя Л.В.* Формы для показа результатов сравнения языков программирования на примере диалектов языка LISP.

[34] Предварительные публикации: *Strachey Christopher.* Programming Languages and Their Definition; *Strachey Christopher.* Fundamental Concepts in Programming Languages, 1967

www.iis.nsk.su/files/preprint/gorodnyaya-2025-forms_0.pdf?ysclid=mk9e9ot2mp144838343

24. *Городняя Л.В.* Сравнение диалектов языка Lisp // Материалы конференции «Научный сервис в сети Интернет», 2025. https://keldysh.ru/abrau/2025/temp/17.pdf

25. Armed Bear Common Lisp (ABCL). https://armedbear.common-lisp.dev/

26. *Евстигнеев В.А., Городняя Л.В., Густокашина Ю.В.* Язык функционального программирования SISAL // В сб. «Интеллектуализация и качество программного обеспечения». Новосибирск, 1994. С. 21–42.

27. *Сошников Д.В.* Программирование на F#. М.: ДМК Пресс, 2011. 192 с.

FORMS FOR DISPLAYING THE RESULTS OF COMPARISON OF PROGRAMMING LANGUAGES USING THE EXAMPLE OF DIALECTS OF THE LISP LANGUAGE

L. V. Gorodnyaya[0000-0002-4639-9032]

A. P. Ershov Institute of Informatics Systems, Novosibirsk, Russia
Novosibirsk State University, Novosibirsk, Russia

lidvas@gmail.com

Abstract

This article focuses on developing forms for presenting the results of analyzing and comparing the characteristics of programming languages, systems, and paradigms. The proposed form is demonstrated through a comparison of the Lisp language, its most successful dialects (Scheme, Common Lisp, Racket, Clojure), and the functional programming paradigm across different levels of language and system definition. The form allows for a concise presentation of the inheritance of several features of the Lisp language and their evolution in its dialects, at the levels of concrete syntax, abstract semantics, and implementation pragmatics."

***Keywords**:* programming language, Lisp, Scheme, Common Lisp, Racket, Clojure, functional programming, comparison of programming languages, concrete syntax, abstract semantics, implementation pragmatics.

REFERENCES

1. *Gorodnyaya L.V.* O predstavlenii rezul'tatov analiza yazykov i sistem programmirovaniya. Nauchnyy servis v seti Internet: trudy XX Vserossiyskoy nauchnoy konferentsii (17–22 sentyabrya 2018 g., g. Novorossiysk). M.: IPM im. M.V. Keldysha, 2018.

2. *McCarthy J. Abrahams P. W., Edwards D. J. et al.* LISP 1.5 Programming Manual. The MIT Press, Cambridge, 1963. 106 p.

3. *Dybvig K.R.* The Scheme Programming Language.
URL: https://www.scheme.com/tspl4/

4. *Graham P.* ANSI Common Lisp. Prentice Hall, 1996. 432 p.

5. The Racket Reference. URL: https://docs.racket-lang.org/reference/

6. Clojure Programming. OReilly.com. Retrieved 2013-04-30. URL: https://cdn.oreillystatic.com/oreilly/booksamplers/9781449394707_sampler.pdf

7. *Ott A.* Vvedeniye v Clojure.
URL: https://alexott.net/ru/clojure/clojure-intro/

8. Differences Clojure with other Lisps.
URL: https://clojure.org/reference/lisps/

9. *Backus J.W.* The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference // Proceedings of the International Conference on Information Processing. UNESCO. 1959. P. 125–132.

10. *Backus J*. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs // 1977 ACM Turing Award Lecture, p. 621–641.

11. *Mitchell R.W.* LISP 2 Specifications Proposal. Stanford Artificial Intelligence Laboratory Memo No. 21, Stanford, Calif., 1964.

12. *Lavrov S.S., Silagadze G.S.* Vkhodnoy yazyk i interpretator sistemy programmirovaniya na baze yazyka LISP dlya mashiny BESM-6. M.: ITM i VT AN SSSR, 1969.

13. *Landin P.J.* The Mechanical Evaluation of Expression // Comput. J. 1964. Vol. 6, No. 4. P. 308–320. https://doi.org/10.1093/comjnl/6.4.308

14. *Khenderson P.* Funktsional'noye programmirovaniye. Primeneniye i realizatsiya = Functional Programming. M.: Mir, 1983. 349 p.

15. *Henderson P., Jones G.A.; Jones S.B.* The LispKit Manual. University of Oxford Computing Lab. 1983.
URL: https://github.com/hanshuebner/secd/tree/master/lispkit/LKIT-2

16. *Michie D.* 'Memo' Functions and Machine Learning" (PDF). Nature. 1968. Vol. 218 (5136), P. 19–22. Bibcode:1968Natur.218...19M.
https://doi.org/10.1038/218019a0. S2CID 4265138

17. *Strachey Ch.* Fundamental Concepts in Programming Languages // Higher-Order and Symbolic Computation. 2000. Vol. 13, No. 1–2. P. 11–49.

18. *Henderson P., Morris JH.* A lazy evaluator. Symposium ACM Sigact-Sigplan sur les principes des langages de programmation // DBLP, Proceedings of the 3rd ACM SIGACT-SIGPLAN symposium on Principles on programming languages (POPL), 1976. P. 95–103.

19. *Dushkin R.V.* Funktsional'noye programmirovaniye na yazyke Haskell / Gl. red. D.A. Movchan. M.: DMK Press, 2008. 544 p.

20. Ofitsial'nyy sayt yazyka Haskell. "O yazyke"
http://haskell.org/aboutHaskell.html

21. From PLT Scheme to Racket. Racket-lang.org. Retrieved 2011-08-17.
URL: https://docs.racket-lang.org/guide/intro.html Welcome to Racket

22. *Gorodnyaya L.V.* Lisp i yego dialekty. Novosibirsk, preprint, 2025.
URL: https://www.iis.nsk.su/repository/gorod.14408

23. *Gorodnyaya L.V.* Formy dlya pokaza rezul'tatov sravneniya yazykov programmirovaniya na primere dialektov yazyka LISP.
URL: www.iis.nsk.su/files/preprint/gorodnyaya-2025-forms_0.pdf?ysclid=mk9e9ot2mp144838343

24. *Gorodnyaya L.V.* Sravneniye dialektov yazyka Lisp // Materialy konferentsii "Nauchnyy servis v seti Internet", 2025.
URL: https://keldysh.ru/abrau/2025/temp/17.pdf

25. Armed Bear Common Lisp (ABCL). URL: https://armedbear.common-lisp.dev/

26. *Yevstigneyev V.A., Gorodnyaya L.V., Gustokashina Yu.V.* Yazyk funktsional'nogo programmirovaniya SISAL // v sb. «Intellektualizatsiya i kachestvo programmnogo obespecheniya». Novosibirsk, 1994. S. 21–42.

27. *Soshnikov D.V.* Programmirovane na F#. M.: DMK Press, 2011. 192 p.

СВЕДЕНИЯ ОБ АВТОРЕ

ГОРОДНЯЯ Лидия Васильевна – к. ф.-м. н., старший научный сотрудник Института систем информатики им. А.П. Ершова СО РАН, доцент Новосибирского государственного университета, специалист в области системного программирования и образовательной информатики.

Lidia Vasiljevna GORODNYAYA – Senior Researcher at the A. P. Ershov Institute of Informatics Systems, Siberian Branch of the Russian Academy of Sciences, Associate Professor at the Novosibirsk State University, specialist in system programming and educational informatics.

email: gorod@iis.nsk.su

ORCID: 0000-0002-4639-9032

Материал поступил в редакцию 6 января 2026 года