Российская академия наук
Сибирское отделение
Институт систем информатики
им. А.П. Ершова

Л.В. Городняя

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

Препринт

Новосибирск 2025

Статья посвящена выработке форм для показа результатов анализа и сравнения особенностей языков, систем и парадигм программирования. Продемонстрирована предлагаемая форма показа результатов сравнения языков программирования, их сходства и различия на примере языка Lisp, наиболее успешных его диалектов (Scheme, Common Lisp, Racket, Clojure) и парадигмы функционального программирования.

Рецензент Т.А. Андреева

Siberian Branch of the Russian Academy of Sciences
A.P. Ershov Institute of Informatics Systems

L.V. Gorodnyaya

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

Preprint

Novosibirsk 2025

The article is devoted to the development of forms for presenting the results of analysis and comparison of features of languages, systems and programming paradigms. The proposed form for presenting the results of comparison of programming languages, their similarities and differences is demonstrated using the example of the Lisp language, its most successful dialects (Scheme, Common Lisp, Racket, Clojure) and the functional programming paradigm.

1. **ВВЕДЕНИЕ**

Статья посвящена результатам очередного этапа разработки методик анализа и сравнения языков, систем и парадигм программирования, продолжающейся в лаборатории информационных систем ИСИ СО РАН им. А. П. Ершова в рамках тематики, связанной с исследованием средств и методов преподавания программирования. Ранее выработана визуально-табличная форма показа категорий семантических систем языка программирования (ЯП) [1]. Теперь начинается проверка парадигмально-семантической методики на конкретных долгоживущих ЯП. Такая проверка требует специальных форм показа результатов анализа и сравнения ЯП. Методика учитывает, что термин «язык программирования» в речевой практике понимается как «входной язык системы программирования, обеспечивающей доступ к определённым аппаратным средствам» (ЯиСП). Поэтому, кроме сравнения конструкций ЯП на уровне синтаксиса и семантики, анализируется реализационные структуры данных, пространства доступных процессов обработки данных и дисциплины доступа к памяти в типовых ЯиСП на уровне прагматической динамики. Учтено, что понимание ЯП всегда опирается на ряд известных, возможно неявных, конструкций, необходимых для реализации системы программирования (СП) и воспринимаемых в практике как неотъемлемая часть языка, в реальности существующего как целостная ЯиСП, составляющие которой взаимосвязаны.

Долгоживущие ЯП обычно расширяют ряд вычислительных возможностей и парадигм программирования подключением стандартных библиотек, пакетов, монад или выделением диалектов. Диалект становится самостоятельным ЯП, наследуя особенности исходной ЯиСП, слегка изменяя и дополняя их. Цель выполненного эксперимента — изучить особенности изменения синтаксиса, семантики и прагматики ЯП в диалектах и наследниках, показать особенности наследования конструкций уровня и синтаксиса, и семантики, и прагматики .

В данной статье приведены результаты сравнения языка Lisp с его успешными диалектами, представленные в форме, показывающей, что унаследовано, что изменено и что дополнено. Выбор языка Lisp для первого эксперимента обусловлен не только чёткостью и лаконизмом его описания [2], но и ростом интереса к функциональному программированию (ФП), регулярно происходящем при смене элементной базы и расширении сферы применения ИТ. Кроме того, Lisp можно характеризовать как ЯП одновременно и низкоуровневый, и сверх высокого уровня в зависимости от уровня решаемых задач. Lisp легко адаптируется к решению новых задач и изобретению лаконичных и эффективных конструкций, иногда не соответствующих шаблонам, навязываемым более популярными ЯП.

Эксперимент, выполненный на материале диалектов Pure Lisp, Scheme, Common Lisp, Racket и Clojure [2-8], появившихся с шагом 10 лет, 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 синтаксиса. При сравнении языка Lisp с его диалектами уделено внимание своду принципов ФП и расслоению ЯиСП на базис, расширение, средства диагностики границ вычислимости и отладки программ, а также связи с внешним миром.

Изложение начинается с описания форм и обозначений для показа разноуровневых различий между диалектами. Затем дана краткая справка о языке Lisp и его диалектах Pure Lisp, Scheme, Common Lisp, Racket и Clojure в порядке их появления. Далее сформулированы выводы о замеченных особенностях формирования диалектов языка Lisp.

Общая сводка целей создания рассмотренных диалектов и достигнутых ими результатов представлена в Приложении 1. Развитие гомоиконного синтаксиса, трансформационной семантики и прагматичной динамики диалектов показана в Приложениях 2-5. Приложение 6 показывает стабильность архитектуры, унаследованной

---

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

диалектами от языка Lisp. Приложение 7 посвящено показу особенностей парадигмы ФП. Материалы по языкам ФП, наследующим идеи языка Lisp, таким как Sisal, F# и Haskell [9-12] дают достаточное основание рассматривать Lisp как базовую математику ФП. Результаты их анализа выходят за пределы данной статьи.

Синтаксис программы назван гомоиконным, если он представляет программу в виде её абстрактного синтаксического дерева (AST). Семантика названа трансформационной, если она отражает эквивалентность разных вычисляемых форм. Динамика вычислений названа прагматичной, если она определена требованиями эффективности и производительности программ и производительности программирования. Парадигма программирования отражает стиль мышления в процессе постановки задачи и программирования её решения.

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

2. Формы показа результатов сравнения ЯП

Традиция описывать ЯП и представлять их формальные определения сложилась во времена, когда каждый язык реализовывали автономно, начиная с прагматики приаппаратного уровня, полной реализации анализаторов семантики и синтаксиса ЯП, возможно с разработкой своего формализма, типа расширенных БНФ. Такие формализмы не претендовали на описание семантики и прагматики из-за их чрезмерного разнообразия.

Теперь в практике реализации новых ЯП сложилась тенденция ограничиваться синтаксической надстройкой над существующими языками без пересмотра или уточнения решений уровня прагматики с небольшими вариациями семантики, что означает выделение небольшого числа известных стереотипов в этой сфере, почти не подверженных изменениям. Основные трудности связаны с тем, что описания ЯП и их стандартов обладают слишком большим объёмом (от семисот до полутора тысяч страниц). Используемые в описаниях ЯП формализмы представляют собственно синтаксис языка с неформальными пояснениями без конкретики наследования конструкций предшествующих языков и решений уровня семантики и прагматики, описываемых средствами естественного языка. Полученные из таких источников сведения требуют проверки на реальных ЯиСП, подверженных развитию.

В порядке эксперимента для показа деталей наследования на уровне синтаксиса между диалектами ЯП предлагается использовать расширение форм Бэкуса-Наура, отражающее в стиле ООП отношение наследования между понятиями нового языка и их одноимёнными определениями в предшественниках, частично дополненное показом особенностей некоторых понятий на уровне семантики и прагматики (Табл. 1). Примеры такого представления конструкций «S-выражение», «Форма», «Функция» для сравнения диалектов языка Lisp приведены в Приложении 2, «ленивых вычислений» и структур данных в таблицах 15 и 16, особенностей «REPL-цикла» в таблице 20 Приложения 5.

[2] http://homelisp.ru/
[3] https://www.jdoodle.com/

Таблица 1.
Расширение БНФ для представления результатов сравнения ЯиСП

Формула	*Примечание*
Старое_понятие.Предшественник	*Используется определение из предшествующего языка*
Старое_понятие!~Шаблон.Предшественник	*Из определения предшествующего языка исключаются фрагменты, соответствующие шаблону*
Одноимённое_понятие	*Используется новое определение, полностью замещающее старое*
Элемент …	*Произвольное число вхождений элемента*
Синтаксис *Семантика* Прагматика ***//Комментарий***	*Уровни определения (Таблица 2)*
〚Формулы над множествами 〛 *// скобки для наглядности перехода к семантике, // к семантике вычислений в конкретном пространстве*	*Семантические системы. Формулы семантики вычислений (Таблица 3)*
□ Операции над состояниями памяти □ *// скобки для наглядности перехода к прагматике, // точнее, к семантике изменения состояний памяти*	*Прагматическая динамика. Дисциплина изменения состояний памяти (Таблица 4)*

В порядке эксперимента некоторые особенности семантики и прагматики показаны с помощью специальных обозначений с небольшими комментариями. Для более точного определения семантики и прагматики лаконичной формы пока не нашлось, тем более, что одни и те же конструкции в диалектах могут перемещаться из прагматики в семантику, из семантики в синтаксис — граница между синтаксисом, семантикой и прагматикой условна, она зависит от сложности конструкций, понятия разных уровней могут входить в общую формулу (Таблица 2).

Таблица 2.
Формы для показа границ между уровнями понятий ЯиСП

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

Строки таблицы 2 выражают разноуровневые составляющие определения ЯП привлечением разных шрифтов для понятий уровня синтаксиса, семантики и прагматики. Синтаксис — жирный шрифт, семантика — жирный курсив, прагматика — обычный подчёркнутый шрифт, комментарий — курсив. Более наглядно, но не столь просто, использовать индексы (верхний, обычный, нижний). Примеры так представленных различий в уровнях и границах вхождения в ЯиСП структур данных диалектов языка Lisp приведены в Приложении 3.

Не все важные особенности ЯП удалось выразить такими компактными формами. При поиске форм для показа результатов сравнения ЯиСП, удобных для оценки выразительной силы ЯП, а также трудоёмкости и продуктивности реализации СП, эффективности и производительности программ, создаваемых на базе ЯиСП, 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 критерии зависят от принципов ФП (универсальность данных, само-применимость определений, равноправие и независимость параметров и единственность результатов функций, гибкость блоков памяти и неизменяемость хранимых значений). Они кратко описаны в Приложении 4. Технически в качестве основного критерия выбрана семантическая декомпозиция определений ЯП, позволяющая показывать различия и дистанцию в понятийной сложности между семантическими системами[4], образующими ЯиСП. Для лаконичного показа различий семантики и прагматики предложенные обозначения немного различаются для разных категорий семантических систем — вычисления, структуры данных, управление вычислениями и обработка памяти.

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 ФП выражается формулой:

V, F, R ⟧ **| R □ F □ V** — «**R** является подмножеством **F** и **F** является подмножеством **V**» или «**R** включено в **F**, а **F** включено в **V**».

Такой, присущий ФП, формат семантики ЯП 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 СП. Операции и программируемые функции **F** включаются в основное множество **V**, а правило применения **R** операций **F** к данным **V** — не более чем одна из функций, возможно программируемая. Разные категории семантических систем (вычисления, структуры данных, управление вычислениями и обработка памяти) могут быть подчинены разным правилам применения **R,** требующим разных систем обозначений (таблицы 3-4).

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

Таблица 3.

Обозначения для показа различий в определении **V** — основного множества структур и типов данных ЯиСП.

⟦*Формула*⟧	*Пояснение: формула для семантических систем заключается в двойные квадратные скобки*
∀ etd	Область определения всех функций опирается на любые элементы множества типов данных **etd.**[5]
∃ BSD etd	Область определения всех функций может использовать любые элементы кумулятивной иерархии[6] базовых структур данных из множества **BSD**[7] над элементами типов данных из **etd**.
V ‖ S V ∩ S	Фильтрация, пересечение множеств для выделения подходящих
□ □ U ∩ ∈ ∉ ⊇ ⊆	Операции над множествами
: => λ (x y)	Отображение, переход, формат представления схемы функции
∧S ′S ©S	Методы обработки форм: eval quote compile
≈	Эквивалентно

Эти обозначения позволяют показывать разницу в пространствах допустимых значений в ЯП и особенности ограничений на представление и выполнение функций в разных семантических системах ЯП. Например, формула **⟦∃ { BSD … } ∀ { etd … }⟧** задаёт пространство допустимых данных, устроенное как кумулятивная иерархия структур данных **BSD** над значениями из элементарных типов данных **etd**. Такие обозначения позволили выразить разницу между пространствами допустимых данных в семантических системах рассмотренных диалектов языка Lisp, создания и обработки элементарных, встроенных и программируемых структур данных в соответствии с принципами универсальности ФП, само-определения функций и структур данных, независимости и равноправия параметров и единственности результатов функции (Приложении 5, Таблицы 8-13, 15-16)

Таблица 4.

Обозначения для показа различий в механизмах обработки памяти в ЯиСП

((Действие))	*Пояснение: действия заключаются в двойные круглые скобки*
< U ; … ; >	Структура блоков памяти
→↓ ↑ ←	Операции над элементами памяти: → инициирование, ↓ запись, ↑чтение, ← удаление. ↑ : переменная => пара: данное с адресом
!@	Чтение произвольного элемента с удалением
@	Адрес произвольного элемента памяти
from ! из	Выбор произвольного элемента памяти
+=	пополнение блока памяти
±	Переход к другой дисциплине функционирования
∅	Пустое множество
#	Число элементов блока памяти

5 Примеры **etd: атом, number, string,** metaD (метаданные) и др.

6 Универсум фон Неймана**,** кумулятивная иерархия множеств

7 Примеры **BSD: list, array, hash, set, structure** и др.

*	Многократное повторение операции, можно ни одного
~◊	Не достижим
(GC …)	Вызов мусорщика из программы
H VM GC	Блок памяти, виртуальная машина, мусорщик
Prog □ Heap	Включение одного блока памяти в другой
Var ∈ call	В хождение элемента в выражение, категорию

Эти обозначения соответствуют неявным действиям, сопровождающим вычисления. Например, □→ ↓ ↑* ←□ представляет действие, сопровождающее применение локальной переменной как рассредоточенную последовательность[8] неявных операций над памятью. Сначала можно завести элемент памяти, ссылка на который связывается с переменной. Потом в элемент памяти можно записать значение. Затем записанное значение можно читать из памяти произвольное число раз в пределах локальной области видимости. После выхода из этой области ссылку на элемент памяти нужно удалить, связь переменной исчезнет. Такие обозначения позволили при сравнении диалектов языка Lisp выразить разницу 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 (Таблицы 14, 15).

3. Немного истории языка Lisp

Идеи Джона Маккарти (*John McCarthy*), воплощённые в языке Lisp, сразу вызвали ревнивую критику как со стороны программистов, так и со стороны математиков. Математиков смущала противоречивость некоторых построений с точки зрения классической математики, например, различие контекстов определения, вызова и вычисления функций[9]. Программисты не могли смириться с отсутствием в языке привычных понятий, начиная с «изменения состояний памяти», а также с непредсказуемо медленной обработкой списков в сравнении с быстрой обработкой векторов.

В середине 1970-ых годов Дана Скотт (Dana S. Scott) опубликовал конструктивную теорию, смягчившую критицизм математиков, построив первую непротиворечивую модель бестипового λ-исчисления[10]. Скептицизм программистов оказался много устойчивее. Например, общее мнение, что Lisp — это интерпретируемый язык, скорее всего, связано с тем, что реализации языка Lisp обычно предоставляют диалоговый стиль работы с программой на базе REPL-цикла и не формирует файл с результатом компиляции, поэтому не заметно, что фрагменты программного кода компилируется по мере необходимости. Такое понимание не исчезло при появлении в 1976 году диалекта Scheme, использующего совмещение чтения программы с её полной компиляцией, но сохраняя диалог, и добавляя неявную эффективную векторную реализацию списков.

Язык Lisp, включая интерпретатор и компилятор, был описан в виде формализма на самом языке Lisp. Lisp позволяет создавать программы, динамически порождающие код, выполнять любые реализационные трюки, строить виртуальные машины, специализированные системы, диалекты и расширяющие язык пакеты. Такой потенциал языка Lisp сложился как ответ на вопрос: «*НАСКОЛЬКО* новые задачи компьютерной

---

[8] **Рассредоточенная последовательность** задаёт порядок выполнения операций, между выполнением которых происходят вычисления, определённые независимо.

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

[10] https://ru.wikipedia.org/wiki/ Непрерывность по Скотту

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 представлять как атомы.

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

— Эксперименты при разработке решений новых задач продуктивнее выполнять в диалоге на базе интерпретаторов, компиляция нужна для достаточно отлаженных программ.

— Списки произвольной длины из элементов любой природы могут быть гомоиконными конструкциями как высокого уровня постановки задачи, так и низкого уровня реализационных 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][11], хотя реализация языка изначально поддерживает основные императивные черты ради привлечения опытных программистов и возможности повышать надёжность 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 (QUOTE, COND, LAMBDA, LABEL) и универсальную функцию EVAL, способную вычислить любое правильно представленное списком выражение, 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, определение 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 обобщённого интерпретатора.

Такая система понятий и средств обработки данных позволила поддержать все семантические и прагматические принципы чисто функционального программирования.

Дж. Маккарти ожидал, что оставшиеся проблемы организации вычислений будут решены в более поздней версии, условно названной Lisp 2, в которую планировал включить обработку многомерных векторов, сопоставление с образцами и организацию параллельных вычислений [14]. Рассказывая о языке Lisp, Дж. Маккарти подчёркивал, что экспериментируя программист может изменять в языке Lisp всё что угодно, кроме константы Nil. К 1962 году была готова 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 её пределами, в нашей стране на БЭСМ-6, ЕС ЭВМ, СМ-4 и других машинах[12]. Составные данные языка Lisp на уровне синтаксиса выглядят как списки

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

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

элементов любой природы, хотя неявно в СП на уровне 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 имён, изменяемые поля используются при организации рекурсии и оптимизации отложенных вычислений, а вектора используются для сопряжения со встроенными машинными процедурами.

В начале 1960-ых Питер Ландин (Peter J. Landin) в своей работе о лямбда-исчислении ввел понятие "call-by-name"[13], использованное в описании языка Algol-60. В 1964 году он предложил машину SECD — это виртуальная и/или абстрактная машина, предназначенная для использования в качестве целевого языка (бэкэнд) при компиляции языков ФП [15]. Вскоре появился Lispkit — реализация чисто функционального диалекта Pure Lisp с лексической областью видимости, разработанного в качестве испытательного и учебного стенда при изучении концепций ФП, включая ранние эксперименты с ленивыми вычислениями [16, 17]. Полученные компилятор и виртуальная машина были легко переносимы. Важный в контексте ленивых вычислений термин «мемоизация» был придуман Дональдом Мичи в 1968 году. В 1971 году Кристофер Стрэйчи предложил термин "call-by-need"[14], предшественник термина «ленивые вычисления».

В 1974 году в Xerox началась разработка аппаратуры и системы машинных команд для аппаратной реализации языка Lisp. Язык Scheme был разработан в 1976 году 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 ЯП. Практически одновременно термин "Lazy evaluation" (ленивые вычисления) был введен в статье "Programming Languages and Their Definition" Кристофера Стрейчи, в статье "A Lazy Evaluator" Питера Хендерсона и Джеймса Х. Морриса и в диссертации Родриго Терье. В 1978 году был представлен язык программирования Lazy ML, который стал первым языком, основанным на парадигме ленивых вычислений. В 1978-1979 годах был разработан язык программирования Hope[15] в Эдинбургском университете Великобритании. Этот язык оказал значительное влияние на ML и Haskell, представленный программистскому сообществу в 1987 году с целью притормозить создание новых языков ФП, их новизна затрудняет работу журналов при решении вопросов о приоритете публикуемых результатов.

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

В последние годы особое внимание привлекает диалект Racket (ранее — PLTScheme), созданный в 1994 году как платформа языково-ориентированного программирования [5]. Это симптом перехода практики программирования от накопления правильности программ на уровне библиотек к уровню создания диалектов и проблемно ориентированных языков (DSL).

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

---

13 «вызов по имени»

14 «вызов по необходимости»

15 Филд А., Харрисон П. Функциональное программирование. Перевод под редакцией В.А. Горбатова. – М. Мир, 1993, 638 с. Содержит описание разных вариантов «Мусорщика».

4. Диалект Scheme для эффективной реализации

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

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

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

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

— От рецептов отложенных функциональных значений, формируемых в точке вызова функции, перейти к формированию замыканий в точке определения функции, заодно и макротехнику ограничить выполнением на этапе компиляции, что сводит её потенциал к привычным возможностям препроцессоров во многих ЯиСП. При компиляции автоматически выполнять оптимизацию рекурсий, сводимых к итерациям.

— 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![16]).

Язык Scheme заимствовал терминологию и синтаксис языка Lisp, несколько изменяя смысл ряда понятий и сужая трактовку почти всех принципов ФП. Из характерных особенностей языка Lisp в языке Scheme поддержана примерно треть. Переход к векторам пошатнул позиции пустого списка и атома Nil — реализация пустых векторов в те годы не практиковалась. Принудительная компиляция программы, теряющая исходное AST, затрудняет возможности динамической оптимизации программ и процессов.

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

5. Common Lisp для производственного применения

Common Lisp был разработан в начале 1980-ых с целью объединения полезных механизмов большого числа разрозненных диалектов языка Lisp [4]. Common Lisp часто сравнивают и 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 преодолено сужение потенциала языка Lisp. Сохраняя и восстанавливая основные концептуальные решения языка Lisp, диалект Common Lisp их дополняет, расширяя методы формирования пространств допустимых процессов.

Common Lisp иногда называют Lisp-2[17], а Scheme — Lisp-1, имея в виду использование раздельных пространств имен для функций и переменных[18]. CLISP проводит различие между временем чтения, временем компиляции, временем загрузки и временем выполнения и позволяет пользовательскому коду также учитывать это различие при выборе желаемого типа обработки программ на нужном этапе. В системе CLISP реализован пакет CLOS, дающий полную поддержку популярной в производстве парадигмы ООП.

---

16 set! работает только на ранее введенных символах, define на любых.

17 Ассоциация с проектом Lisp-2 Дж.Маккарти.

18 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.

Этот диалект резко расширил сферу производственного применения языка Lisp, используя на уровне лексикона семантику доступа к матрицам, хэш-таблицам, программируемым структурам данных, подобным структурам в языке C, изменяемым полям и мультизначениям, удобными для моделирования независимых потоков. Всё 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 определённый тип.

Common Lisp даёт ответ 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 ЯП и приложений.

— Компиляция отдельных функций допускает и компиляцию полной программы без потери исходного списочного представления AST.

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

— Гибкость распределения памяти с возможностью программировать вызов «сборщика мусора» дополнена средствами выяснять время, даты и этапы работы программы, чтобы прогнозировать целесообразность применения тех или иных методов.

— Неизменяемость значений уточняется введением неявного понятия «поле» для работы с изменяемыми системными свойствами символа. Для потребляющих много памяти функций 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 мультизначения. Основное отличие от языка Lisp связано с разделением пространств имён 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 автоматизированного средства доказательств теорем (ACL2) и систем компьютерной алгебры (Аксиома, Maxima).

6. Racket для создания новых языков программирования

В 2010 году название Racket получила очередная версия учебной среды программирования на языке Scheme, разрабатываемая с 1995 в Университете Дьюка для обучения созданию, разработке и реализации новых ЯП, что 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 1970-ых годов, 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 модульной 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 ЯП, 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 ЯиСП. Racket включает макросы на этапе и компиляции, и выполнения с удобными функциями для разработки и отладки. Поддержан диалект Scribble — язык разметки для документации и ряд диалектов, связанных с основными парадигмами программирования, проблемно-ориентированными языками (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[19], 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[20]. 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 соответствует предложенной Д. Кнутом парадигме литературного (грамотного) программирования (literate programming)[21].

— 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 (БНФ).

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

— 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 в качестве значения «ложь» специального булева значения #f вместо пустого списки ( ) или атома Nil. Хотя формально язык Racket называют диалектом языка Scheme, по своим особенностям он ближе к Common Lisp.

7. Clojure — новые горизонты ИТ

Clojure — современный диалект языка Lisp, появился в 2007 году, разработан Ричем Хикки (Rich Hickey), независимым разработчиком ПО, ранее разработавшим dotLisp в

---

19 Архитектура ARM (*Advanced RISC Machine)* — усовершенствованная RISC-машина.

20 Сотрудники фирм-разработчиков компиляторов утверждают, что необходимость ожидать результат компиляции даёт им защищённую нишу времени для продумывания программ.

21 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. Возможно Д. Кнут хотел привлечь внимание к тому, что искусство программирования основывается и на владении естественным языком.

рамках проекта .NET Framework. Clojure наследует особенности языка Lisp, обеспечивающие гибкое и мощное метапрограммирование, поддерживает синтаксическое расширение [6-8] и надежную семантику, дополненную механизмами для программирования приложений на базе распределённых и параллельных процессов.

В этом языке определения функций могут неформально сопровождаться пред- и пост-условиями, 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 используются 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 определённых стандартов.

Динамическая типизация означает, что, кроме статической проверки типов переменных на этапе компиляции, проверяются типы значений, точно известные во время выполнения. Это обеспечивает гибкость и быструю разработку, так как не требуется явного объявления типов переменных. Clojure уделяет большое внимание тестированию (clojure.test, test.check, midje, kaocha, Unit Testing, Integration Testing, Property-based testing), вызывающему большее доверие у практиков, 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, векторы 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 её наполнения[22], допускается использование подчеркивания «_» для обозначения игнорируемой позиции. Исходная структура сохраняется в соответствии с принципом неизменяемости данных.

Реструктурированная последовательность аргументов функции позволяет вместо имён связанных переменных использовать нумерацию параметров, список которых можно рассматривать как вектор[23]. 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 учётом проблем надёжности и безопасности.

[22] Похоже на реорганизацию векторов в языке APL.

[23] Подобно некоторым языкам заданий и макропроцессоров.

В качестве компромисса между идеями чистого ФП и необходимостью изменения состояний при организации параллельных процессов введены указательные переменные — атомы как уникальные указатели превращаются в указательные переменные и рассматриваются как реализационные структуры, обеспечивающие разные дисциплины доступа к памяти и стратегий 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 синтаксис отображений, множеств и векторов. Введены метаданные — кодированный аналог списка свойств, который может быть связан со значением или указательной переменной.

— Можно синхронизовывать выполнение потоков: откладывание, ожидание, обещание, передача, готовность, блокировка (delay, future, promise, deliver, is-done?, deref), контролировать ход вычисления, а также представлять эффективные формы циклов, не использующих 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 сохраняет примерно 80% особенностей языка Lisp, уточняет ряд его решений для удобства представления параллельных процессов и дополняет его заметным комплектом средств, соответствующих развитию элементной базы, поддерживающих отладку взаимодействующих процессов и удостоверение правильности программ. Самое заметное расширение связано с понятием атом. Атом из неявного уникального указателя на список свойств стал явным указательным типом данных, позволяющим поддерживать различные дисциплины обработки памяти, возникающие в разных моделях параллельных вычислений, разнообразие которых непредсказуемо велико. Кроме того, механизм реструктуризации данных распространён на список аргументов, что расширяет форматы определения функций, позволяет отказываться от использования имён связанных переменных[24], что удобно при генерации машинного кода.

8. Итог сравнения диалектов

В ходе эксперимента выяснилось, что результаты сравнения ЯП следует показывать декомпозированными по отдельным особенностям и конструкциям ЯиСП. Обозначения из таблиц 1-4 оказались достаточными для показа синтаксических, семантических и прагматических различий между рассмотренными диалектами, причём с выражением отношения наследования между ЯП. В таблицах 11-16 и 19 Приложений 2-5 удалось показать происходившее при создании диалектов изменение особенностей поддержки семантических принципов «универсальность», «само-определение функций» (рекурсия), «независимость и равноправие параметров функций», «единственность результата функции», а также форматов ветвлений, ленивых вычислений и REPL-цикла. В таблицах 17 и 18 показано произошедшее изменение особенностей поддержки принципов «гибкость границ блоков памяти» и «неизменяемость хранимых в памяти значений». Кроме того, в таблице 20 показана сводка особенности поддержки современного ФП, в которой нашли применения все системы обозначений из таблиц 1-4. Следует отметить, что объём таблицы 20 почти в два раза меньше объёма таблиц 8-10 из Приложения 4, описывающего принципы ФП.

---

[24] Интересно, что в своё время основатель нашей математической школы Н.Н. Лузин не одобрял термин «связанная переменная», пояснял, что это вообще не переменная, потому что её имя можно заменять на другое — смысл формулы не изменится. Своего термина не предложил.

Отмечая разницу в целях создания диалектов языка Lisp, можно заметить, что рассмотренные диалекты обладают стабильным реализационным ядром, использующим конкретный комплект структур данных и механизмов их обработки. В Приложении 6 показано, что вариации структур данных и значений сводятся к пересмотру границ между лексиконом, синтаксисом, семантикой и прагматичной динамикой языка, между периодом компиляции и выполнения программы, а также к вариантам представления значения «ложь» и уточнению понятия «атом» как указательной переменной. Системные прагматические решения по обработке структур данных в новых диалектах из неявных уровня прагматики становятся доступными сначала на уровне абстрактной семантики с помощью функций, затем переходят на уровень конкретного синтаксиса, получают своё представление, что не нарушает семантическую эквивалентность программ, т. к. AST диалектов имеет подобное списочное представление. Семантика вычислений в диалектах по разному взаимодействует с семантикой изменения состояний памяти, что проявляется, как правило, в именовании функций и выборе границ изменяемых данных.

Принципы и понятия ФП в диалектах языка Lisp уточнялись в зависимости от продуктивности и критериев качества программирования в разных областях приложения. Для Scheme это сохранение привычки к присваиваниям и векторам, для Common Lisp — полнота разнообразия структур данных, для Racket — создание специализированных диалектов, освобождающий от нагромождения библиотек, для Clojure — освоение многопроцессорных комплексов и распределённых информационных систем. Более подробно результаты сравнения отражены в препринте [18].

9. Заключение

Вот и завершён обзор результатов сравнения наиболее популярных диалектов языка Lisp и эксперимент по выбору форм для показа результатов анализа и сравнения ЯП. Основная причина проведения такого сравнения с поиском обозримых форм и лаконичных обозначений связана с разработкой методик оценки продуктивности ЯиСП и программируемых решений в отличие от непосредственного измерения эффективности и производительности программ. Полученные результаты образуют основу для определения номенклатуры семантических систем языков ФП и методик оценки продуктивности ЯП. Следует отметить не только новые диалекты языка Lisp, но и выпуски их реализаций — 25 августа 2025 года выпущена очередная реализация Armed Bear Common Lisp (ABCL).

При измерении мощности ЯП по характеристике пространства доступных процессов вычислений самыми мощными являются языки ассемблера. Языки более высокого уровня, включая 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 году, показывают, что в них сохранены основные возможности языка Lisp, реализационные структуры данных, базовые принципы ФП и понятия с небольшими вариациями на злобу дня. Авторы этих диалектов чётко 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.

Результаты сравнения диалектов языка Lisp показывают медленное смягчение программистского скептицизма в отношении к принятым в языке Lisp решениям и парадигме ФП по мере прогресса элементной базы и развития ИТ, что видно по созданию новых ЯП, включающих механизмы ФП как важное преимущество. Семейство Lisp теперь является

одним из старейших и наиболее влиятельных семейств ЯП, его мощность в разных источниках, начиная с Википедий, оценивается от 500 до тысяч ЯП и диалектов. Кроме того, следует учесть языки ФП, 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 не меньшее число диалектов у языков C/C++, Bash, Perl, Python, JavaScript, BASIC, Forth, SQL, Fortran, Pascal, Ada, Assembler.

Следующий эксперимент по представлению результатов сравнения ЯиСП предполагается выполнить на материале языков ФП, таких как Erlang, Sisal, F# и Haskell, рассматриваемых как наследники языка Lisp. Потом интересно выполнить представление результатов сравнения представителей других долгоживущих семейств ЯП, в первую очередь это Fortran и C, сохранившие значимость до наших дней. Отдельная задача − представление результатов сравнения наших ЯП с зарубежными аналогами.

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. McCarthy J., Abrahams P.W., Edwards D. J. et al. LISP 1.5 Programming Manual. The MIT Press, Cambridge, 1963. 106 p.
3. Kent R. Dybvig 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.
7. https://cdn.oreillystatic.com/oreilly/booksamplers/9781449394707_sampler.pdf
8. Отт А. Введение в Clojure — https://alexott.net/ru/clojure/clojure-intro/
9. Differences Clojure with other Lisps — https://clojure.org/reference/lisps/
10. 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.
11. Сошников Д.В. Программирование на F#. — М.: ДМК Пресс, 2011. — 192 с.
12. Душкин Р.В. Функциональное программирование на языке Haskell / М.: ДМК Пресс,, 2008. — 544 с., ил. с. — 1500 экз. — ISBN 5-94074-335-8.
13. Официальный сайт языка Haskell — "О языке" http://haskell.org/aboutHaskell.html
14. John Backus 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/
15. Mitchell, R.W., "LISP 2 Specifications Proposal", Stanford Artificial Intelligence Laboratory Memo No. 21, Stanford, Calif., 1964.
16. Landin, P.J. (January 1964). "The Mechanical Evaluation of Expressions". *Comput. J.* 6 (4): 308—320. *doi*:*10.1093/comjnl/6.4.308*
17. Хендерсон П. Функциональное программирование. Применение и реализация = Functional Programming. — М.: Мир, 1983. — 349 с.
18. Henderson, Peter; Jones, Geraint A.; Jones, Simon B. (1983). The LispKit Manual. University of Oxford Computing Lab. ISBN 0-902928-18-X. https://github.com/hanshuebner/secd/tree/master/lispkit/LKIT-2

19. Городняя Л.В. Lisp и его диалекты // Новосибирск, препринт, 2025 https://www.iis.nsk.su/repository/gorod.14408.
20. Городняя Л.В. Сравнение диалектов языка Lisp. // Материалы конференции "Научный сервис в сети Интернет" https://keldysh.ru/abrau/2025/temp/17.pdf

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. McCarthy J., Abrahams P. W., Edwards D. J. et al. LISP 1.5 Programming Manual. The MIT Press, Cambridge, 1963. 106 p.
3. Kent R. Dybvig 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.
7. https://cdn.oreillystatic.com/oreilly/booksamplers/9781449394707_sampler.pdf
8. Ott A. Vvedeniye v Clojure — https://alexott.net/ru/clojure/clojure-intro/
9. Differences Clojure with other Lisps — https://clojure.org/reference/lisps/
10. Yevstigneyev V.A., Gorodnyaya L.V., Gustokashina Y.U. V. YAzyk funktsional'nogo programmirovaniya SISAL // v sb. «Intellektualizatsiya i kachestvo programmnogo obespecheniya». — Novosibirsk, 1994, — S. 21–42.
11. Soshnikov D.V. Programmirovane na F#. — M.: DMK Press, 2011. — 192 p.
12. Dushkin R.V. Funktsional'noye programmirovaniye na yazyke Haskell / Gl. red. D.A. Movchan;. — M.: DMK Press,, 2008. — 544 p — ISBN 5-94074-335-8.
13. Ofitsial'nyy sayt yazyka Haskell — "O yazyke" http://haskell.org/aboutHaskell.html
14. John Backus 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/
15. Mitchell, R.W., "LISP 2 Specifications Proposal", Stanford Artificial Intelligence Laboratory Memo No. 21, Stanford, Calif., 1964.
16. Landin, P.J. (January 1964). "The Mechanical Evaluation of Expressions". *Comput. J.* 6 (4): 308—320. *doi*:*10.1093/comjnl/6.4.308*
17. Khenderson P. Funktsional'noye programmirovaniye. Primeneniye i realizatsiya = Functional Programming. — M.: Mir, 1983. — 349 p.
18. Henderson, Peter; Jones, Geraint A.; Jones, Simon B. (1983). The LispKit Manual. University of Oxford Computing Lab. ISBN 0-902928-18-X. https://github.com/hanshuebner/secd/tree/master/lispkit/LKIT-2
19. Gorodnyaya L.V. Lisp i yego dialekty // Novosibirsk, preprint, 2025 https://www.iis.nsk.su/repository/gorod.14408 .
20. Gorodnyaya L.V. Sravneniye dialektov yazyka Lisp. // Materialy konferentsii "Nauchnyy servis v seti Internet" https://keldysh.ru/abrau/2025/temp/17.pdf .

Приложение 1

Проблемы, цели и результаты создания языка Lisp и его диалектов

Развитие компьютерных средств и систем программирования сопровождается расширением сфер их применения, а следовательно и изменением актуальности разных проблем, постановкой новых целей и необходимостью решения новых классов задач. Откликом становится создание новых ЯП и их диалектов, отвечающих на запросы своего времени.

Таблица 5.

Задачи, на решение которых откликнулись диалекты языка Lisp

ЯП	*Проблема*
Lisp	«*НАСКОЛЬКО* новые задачи компьютерной обработки информации отличаются от традиционных задач обработки чисел?»
Pure Lisp	«*КАКОЙ* может быть методика обучения программированию на языке Lisp?»
Scheme	«*КАК* сделать функциональное программирование столь же эффективным как привычное императивное?»
Common Lisp	«*ЧТО* может дать функциональное программирование программной индустрии?»
Racket	«*КТО* будет определять языки программирования в будущем?»
Clojure	«*ГДЕ* новые горизонты, в освоении которых помогает продуктивность и моделирующая сила языка Lisp?»

Результаты создания и применения языка Lisp и его диалектов показали продуктивность происходившего языкотворчества, существенно расширившего сферу применения компьютерных и информационных технологий.

Таблица 6.

Достижения диалектов языка Lisp в сфере компьютерной обработки информации

Год	*Результат*
1958	Lisp вывел за пределы традиционных задач обработки чисел,
1962	Pure Lisp показал концепцию чисто функционального программирования (ЧФП), Lisp 1.5 стал типовой системой ФП,
1976	Scheme создал прецедент эффективной реализации ФП,
1984	Common Lisp обеспечил продуктивность производственного применения ФП,
1995	Racket сформировал систему обучения созданию и реализации ЯП
2007	Clojure поддержал полноценное применение ФП в новых ИТ, преимущественно через JVM и развил отдельные механизмы и понятия.

Приложение 2

Показ результатов сравнения синтаксиса S-выражений и синтаксического представления вычисляемых форм и функций[25]

Язык Lisp предлагает предельно лаконичный синтаксис символьного представления любых данных, включая программы, в виде кругло скобочных списков из элементов произвольной природы, разделяемых при необходимости пробелами — (S-выражение). Теперь такое свойство называют «гомоиконность». Lisp обладает гомоиконным синтаксисом, программа представляется непосредственно абстрактным синтаксическим деревом (*abstract syntax tree* — AST) в виде иерархии символьных выражений - S-выражений.

Lisp, Pure Lisp: **S-выражени**е **::= атом** *// идентификаторы, числа, строки*
| **(S-выражение . S-выражение)** *// пары*
| **(S-выражение …)** *// списки через пробел*
Scheme: **S-выражение ::= S-выражение.Lisp | #f | #t** *// булевы значения*
| **[S-выражение.Scheme …]** *// группировка*
Common Lisp: **S-выражение ::= S-выражение.Lisp**
| **&атом** *// разметка видов параметров*
| **:атом** *// ключевые параметры и константы*
Racket: **S-выражение ::= S-выражение.Scheme**
| **#S-выражение.Lisp** *// вектор*
| **{ S-выражение … }** *// хэш-таблица*
Clojure: **S-выражение ::= S-выражение.Racket | true | false**
| **#{S-выражение ... }** *// множество*
| **[S-выражение ...]** *// последовательность*
| **{(ключ => значение) … }** *// хэш, ассоциативный список*
| **((S-выражение \ ",")…)** *// список через запятую*
| ^ meta *//данные, подобные спискам свойств*
| ***#"как в Java"*** *// регулярные выражения*

Списочные S-выражения унаследованы всеми диалектами языка Lisp. К такому представлению данных добавляются квадратные и фигурные скобки, сначала в Scheme и Racket для повышения наглядности, потом в Clojure для выделения множеств, последвателльностей и хэш-таблиц при их переносе с неявного уровня прагматики на явный уровень синтаксиса. Clojure предлагает квадратные сколки для последовательностей п вспоминает, что когда-то элементы списка в языке Lisp разделяли запятыми.

Выделяются литеры **# & : ^ =>** в роли функций, влияющих на смысл атомов или списков. Это говорит о том, что граница между синтаксисом и семантикой условна. Некоторые особенности программируемых конструкций, удаётся достаточно лаконично и понятно представить модифицированными БНФ. Здесь показаны формулы РБНФ, показывающие синтаксические представления функций и вычисляемых форм, иногда сопровождаемые комментариями.

Lisp: ***Форма*** **::= атом** *// идентификатор*
| **'S-выражение** *// константа*
| **(*Функция* S-выражение …)** *// выражение*
Scheme: ***Форма*** **::=** ***Форма.Lisp*** **!~ ( )** *// кроме пустого списка*
Common Lisp: ***Форма*** **::=** ***Форма.Lisp***
Racket: ***Форма*** **::=** ***Форма.Scheme***

[25] Обозначения и раздела 2 Таблица 1

| *// инфиксная арифметика*

Clojure: ***Форма*** **::=** ***Форма.Lisp***

| **[S-выражение …]** *// последовательности*

Во всех диалектах наследуется базовая форма представления выражений языка Lisp, Scheme исключает из неё пустой список, реализационно не сводимый к вектору, что выясняется экспериментально при компиляции, по документации это не заметно. Clojure добавляет к ней явное представление последовательностей (это убирает необходимость в форме Prog).

Аналогично показаны формулы, показывающие синтаксическое представление определений функций, иногда сопровождаемые комментариями.

Lisp: ***Функция*** **::= атом** | Анонимная | Именованная

Анонимная **::= (lambda (атом …)** ***Форма*** **)** *// динамика*

Именованная **::= (*** ***define*** **Имя** Анонимная **)** *// статика*

Scheme: ***Функция*** **::=** ***Функция.***Lisp

| **(set! атом.**Lisp Анонимная **)** *// обновлённое*

Анонимная **::= (lambda (атом … [. атом])** ***Форма*** **)**

// динамика и возможен серийный параметр после точки, последний

Common Lisp: ***Функция ::=*** ***Функция.***Lisp

| *// неявный полиморфизм по виду параметров*

| **#’** ***Функция*** *// функциональная константа*

Именованная **::= (*** ***defun*** **Имя (атом …)** ***Форма*** **)**

// другой формат определения

Racket: ***Функция*** **::=** ***Функция.***Lisp

| ((*case-lambda* S-выражение [(*Предикат* …) (*Функция* ...)] …)

// полиморфизм по предикату

| ((*case-lambda* [(*Предикат* …) (*Функция* ...)] …)

| ((*match-lambda* [(*Шаблон*…) (*Функция* ...)] …)

// полиморфизм по шаблону — другой синтаксис

| ((*match-lambda* S-выражение [(*Шаблон* …) (*Функция* ...)] …)

Clojure: ***Функция*** **::=** ***Функция.***Lisp

Анонимная **::= (fn [Атом …]** ***Форма*** **)**

// другие скобки и ключевое слово

Именованная **::= (*** ***defn*** **Атом** Анонимная **)** *// другое ключевое слово*

С точностью до ключевых слов диалекты наследуют общую схему представления определений функций с небольшими вариациями. Диалект Scheme позволяет выделять серийный параметр для функций с произвольным числом параметров и изменять ранее объявленные определения (set!).

Common Lisp вводит префикс #’ для функциональных значений.

Racket делает явным полиморфизм (lambda-case) и вводит генераторы лексеров/парсеров (match-lambda) для определения диалектов ЯП..

Clojure выделяет список параметров квадратными скобками, подчёркивая механизм реструктуризации — можно использовать номера параметров вместо имён связанных переменных. Кроме того, производит смену имён некоторых функций (lambda-fn, define-defun-defn и др.).

Приложение 3

Показ результатов сравнения разноуровневых структур данных

Расширенные БНФ внешне не различают понятия уровня синтаксиса, семантики или прагматичной динамики. Чтобы показать изменение уровня вхождения структур данных в ЯиСП разных диалектов (динамиика → *семантика*→ синтаксис) используются описанные в разделе 2 формы, отражающие с помощью шрифтов или форматирования показывать уровень вхождения понятия в конкретном ЯП.

Язык **::= Синтаксис *Семантика*** Динамика *// Комментарий*

Такая форма означает, что обычным шрифтом в ней обозначается язык, жирным шрифтом обозначается понятие, имеющее синтаксическую форму на базовом уровне, жирный курсив используется для понятий, реализуемых как вызовы семантических функций, подчёркивание применяется для представления неявных понятий уровня прагматической динамики, а простой курсив применяется для комментария, отделяемого двумя косыми чертами. Таким образом, можно показать как одно и тоже содержание распределяется по разным уровням ЯиСП.

Lisp::**списки**, вектора множества хэш-функции изменяемые поля
// явные списки, остальное внутри прагматики
*// ассоциативные списки, списки программируемых свойств**

Scheme:: ***списки*** **вектора** множества хэш-функции изменяемые поля
// списочная форма вектора, выглядящего как список
// ассоциативные списки и списки системных свойств

Common Lisp:: **списки *вектора множества хэш-функции*** изменяемые поля
// функции конструирования структур данных, моделируемых списками
*// ассоциативные списки, списки программируемых свойств**

Racket:: **списки вектора** множества хэш-функции изменяемые поля
// ввод векторов в виде списков с префиксом #
// ассоциативные списки, списки свойств,

Clojure:: **списки вектора множества хэш-функции** метаданные изменяемые поля
//явный синтаксис для всех структур данных
*// ассоциативные списки, списки программируемых свойств**

Приведённые формулы показывают, что в языке Lisp главное — списки, на уровне синтаксиса представлены только списки, вектора, множества, хэш-функции и изменяемые поля присутствуют неявно в прагматике и динамике вычислений, ассоциативные списки и списки программируемых свойств произвольного состава поддержаны функциями уровня семантики. Список как основная структура данных языка позволяет на этапе экспериментов моделировать другие структуры, включая вектора, множества, хэш-таблицы, неявно существующие в реализации.

При переходе к диалекту Scheme вектора представляются явно, хотя выглядят как списки, множества, хэш-функции, изменяемые поля присутствуют неявно в прагматике, списки свойств используются на уровне семантики для имён функций и переменных, нет программируемых свойств. Вектора взяты за основу реализационных структур ради эффективности. Scheme внешне наследует синтаксис списков, но неявно реализует их как вектора, а списки присутствуют как расширение в отдельном модуле.

Common Lisp возвращается к спискам в качестве основной структуры, вектора, множества, хэш-функции (ассоциативные списки, списки программируемых свойств) поддержаны функциями уровня семантики, изменяемые поля введены неявно, на уровне

реализации. Common Lisp дополняет работу со списками функциями конструирования и обработки других структур данных по их списочной модели. Списки, возможно представляющие многомерные вектора, множества и хэш-таблицы, обрабатываются функциями доступа к их реализации. Кроме того, на уровне реализации добавлено представление чисел машинной точности.

Racket использует явное представление и списков и векторов, остальное наследует от Scheme. Списки и пары дополнены обработкой других структур и типов данных. К обычным структурам данных (вектора, множества, хэш-таблицы, структуры) на уровне семантики Racket добавляет классы, объекты, потоки, регулярные выражения (regular expressions), дату и время (date/time) и различает точные (exact) и неточные (inexact) числа.

Clojure вводит явный синтаксис для всех структур данных. Семантическая поддержка ассоциативных списков и списков программируемых свойств* допускает произвольный состав, на уровне прагматичной динамики появляются неизменяемые метаданные как эквивалент списков свойств. Списки как основа реализации Clojure дополняются синтаксисом для векторов, множеств, хэш-таблиц, программируемых структур данных и обогащаются транзакционной памятью, реструктуризацией структур данных и динамическими указательными переменными. На уровне семантики Clojure поддерживает реструктуризацию любых последовательностей, включая списки аргументов, транзакционную память как в базах данных и понятие атома из уникального указателя становится указательной переменной, обеспечивающей разные дисциплины доступа к памяти и стратегий многопоточности (atom, ref, agent).

То же самое можно выразить в более наглядной форме:

Язык:: **Синтаксис** ***Семантика*** Прагматика *//Комментарий,*

Вышеописанные формы приобретают вид:

Таблица 7.
Показ разноуровневости структур данных

Lisp:: **списки** ***(ассоциации свойства*)*** вектор множество хэш-функции поля *// явные списки, остальное внутри прагматики или семантики*
Scheme:: ***списки свойства*** **вектор** множество хэш-функции ассоциации поля *// списочная форма вектора*
Common Lisp:: **списки** ***вектор множество хэш-функции (ассоц. свойства*)*** поля *// функции конструирования структур данных, моделируемых списками*
Racket: **списки вектор** ***свойства*** множество хэш-функции ассоц.списки поля *// ввод векторов в виде списков с префиксом #*
Clojure: **списки вектор множества хэш-функции** ***ассоциации свойства**** метаданные поля *//явный синтаксис для всех структур данных*

Такие формулы показывают, как одно и то же множество структур данных распределено по разным уровням ЯиСП в диалектах языка Lisp.

Приложение 4

Принципы функционального программирования

Таблица 8.

Характеристика и особенности принципов парадигмы ФП на уровне синтаксиса

Правило	*Пояснение*
Гомоиконный синтаксис	Любой информации для компьютерной обработки можно дать символьное представление — S-*выражение*, в котором произвольная неделимая элементарная информация представляется *атомами,* а представление составной информации можно конструировать так, что из иерархии таких представлений можно извлекать любые составляющие части. Обычно это *бинарные деревья*. Отсутствие информации символизируется атомом Nil.
Универсальность символьных форм	Любые данные в программах можно представлять как символьные выражения — символьные формы, а все понятия, связанные с программированием обработки данных, рассматривать как функции или применение функций. Функции в свою очередь, как любые данные, представимы в виде символьных форм. Функции и их параметры могут иметь имена, образующие области видимости - иерархия.
Самоопределение функций (рекурсия).	Все понятия программирования можно рассматривать как функции или применение функций и любая обработка информации программируется как определение или применение функций. Возможны универсальные функции, работающие на любых данных, включая представления самих функций (рекурсия). Самоопределение функций и структур данных позволяет определения делать лаконичными. Самоопределение функций по существу связано с особой категорией — предикатами, позволяющими выяснять принадлежность значения параметра функции конкретному типу данных, что символизируется отличием от пустого списка, атома Nil.

Таблица 9.

Характеристика и особенности принципов парадигмы ФП на уровне семантики

Правило	*Пояснение*
Трансформационная семантика	Семантические принципы (независимость и равноправие параметров, единственность результата функции) адресованы программисту, реализация ЯП их поддерживает полностью, с небольшими вариациями в диалектах. Один и тот же результат может быть получен разными функциями, что позволяет трансформировать программы, заменяя представления вызовов функций на их результаты или на представление эквивалентных функций.

Независимость и равноправие параметров функции	Параметры функции не зависят ни друг от друга, ни от порядка и времени их вычисления. Возможны разные категории функций. Кроме обычных функций, применяемых к заранее вычисленым значениям аргументов, существуют макросы, применяемые к представлениям аргументов без их предварительного вычисления. Такая разница позволяет управлять вычислениями и программировать или моделировать любые схемы вычислений, включая параллельные или ленивые — отложенные или опережающие.
Единственность результата функции	Для организации потока вычислений и трансформации программ удобно, чтобы любая функция вырабатывала результат, причём ровно один. При необходимости большего числа результатов их можно объединять в один список, а отсутствие результатов можно выражать значением «пустой список» — Nil. Применяемая к конкретным аргументам функция вырабатывает определённый результат, в повторном вычислении которого нет необходимости.

Таблица 10.

Характеристика и особенности принципов парадигмы ФП на уровне прагматики

Правило	*Пояснение*
Прагматичная динамика	Прагматические принципы (гибкость распределения памяти, неизменяемость хранимых значений) обеспечивает реализация, программист учитывает их как данность, хотя система может предлагать альтернативы в силу принципа универсальности, поддерживать функции управления памятью и вычислениями.
Гибкость распределения памяти	Невозможно заранее так распределить память, чтобы в динамике выполнении программы не возникала необходимость его изменить. Чтобы исключить необоснованный простой памяти, нужна поддержка изменения границ между блоками памяти по необходимости или по запросу из программы. Такую работу выполняет специальная программа «сборки мусора» (*garbage collection* — GC) либо автоматически, либо как вызов функции GC.
Неизменяемость хранимых значений	Хранимые значения могут многократно использоваться в процессах вычислений, поэтому без необходимости нет оснований их изменять. При вычислениях новые значения размещаются в свободной памяти без нарушения состояния памяти аргументов или ранее сохранённых структур данных. Указатели не являются значениями языка, поэтому на них принцип неизменяемости не распространяется. Для хранения изменяемых указателей выделяются специальные изменяемые поля, возможность изменения состояния которых оговаривается заранее. Например, это указатели на поколения параметров рекурсии или указатель на замыкание функционального значения, содержащего свободные переменные.

Приложение 5[26]

Семантика поддержки принципов функционального программирования

Программа — абстрактное синтаксическое дерево (AST), представленное в виде иерархии символьных выражений — S -выражений.

Таблица 11.

Показ поддержки семантического принципа «универсальность»

⟦*Формула* ⟧ : ⟦∃ BSD ... ⟧ ⟦∀ etd ... ⟧	*Пояснение:* Пространство допустимых данных устроено как кумулятивная иерархия базовых структур данных BSD над значениями из встроенных типов данных etd.
Lisp: **данное ::= S-выр || ⟦∃ BSD etd || ⟦□ etd ⟧** **etd ::= {atom number string}** **BSD ::= list**	Неделимую информацию можно представлять как атом произвольной природы или числа и строки. Составная информация представляется над ними как иерархия списков.
Scheme: **данное ::= данное.**Lisp **etd ::= etd.Lisp !~ atom | { *id* array port }** **BSD ::= { list array }**	Диалект Scheme ограничивает пространство допустимых данных и добавляет в типы данных идентификаторы, вектора и порты.
Common Lisp: **данное ::= данное.**Lisp **etd ::= etd.Lisp | { [number]** поле **}** **BSD ::= BSD.Lisp | {*array hash set structure*}**	Common Lisp к пространству допустимых данных языка Lisp, добавляет конечные числа ([N]) и изменяемые поля. Поддерживает доступ к многомерным массивам, хэш-таблицам, множествам, структурам данных.
Racket: **данное ::= данное.**Common Lisp **etd ::= etd.**Scheme | **{ *object port* document }** **BSD ::=BSD.**Common Lisp	Racket нацелен на формирование навыков создания DSL-языков, включая подготовку документации, что соответствует предложенной Д. Кнутом парадигме литературного (грамотного) программирования (literate programming).
Clojure: **данное ::= данное.**Racket **etd ::= etd.**Racket | metaD **BSD ::= BSD.**Racket | **{array set hash structure}**	Clojure переводит на уровень синтаксиса представления векторов, множеств, отображений (хэш-таблиц) и других структур данных, добавляя к встроенным типам данных неявные метаданные.

Функции, их синтаксис показан в Приложении 2, могут быть само-определимыми.

[26] Обозначения из раздела 2, Таблица 3

Таблица 12.[27]
Показ принципа «само-определение функций» (рекурсия)

⟦*Правильные формулы* ⟧	*Пояснение*
Lisp: **F ::= *Функция*** || ⟦∧**S-выр … => S-выр** ⟧ Lisp: **M ::= *Функция*** || ⟦**S-выр … =>** **(S-выр □ F ↕ S-выр … => S-выр *)** ⟧ *// преобразует программу в любое время* Lisp: **P ::= *Функция*** || ⟦**P ⊂ F** ⟧ || ⟦∧**S-выр … => { Nil | S-выр }** ⟧ *// результат трактуется как булево значение*	Множество функций распадается на три категории: F, M, P. F могут обрабатывать и вырабатывать любые данные. Макросы M вырабатывают произвольные формы, включая определения функций. Результатом предиката P может быть Nil или что-то другое.
Scheme: **F ::= F**.Lisp Scheme: **P ::= *Функция*** || ⟦∧**S-выр … => { #f | S-выр }** ⟧ Scheme: **D ::= *Функция*** || ⟦∧**S-выр …** **=> #<u><unspecified></u>** ⟧ *// вывод данных* Scheme: **M ::= ⟦©*Функция*: ©S-выр …** **=> { S-выр *Функция*}**⟧	Результатом предиката P может быть *#f* или что-то другое. Выделяется категория «процедуры» D, не вырабатывающие значений. «Гигиенические» макросы работают только на этапе компиляции.
Common Lisp: **F ::= F.**Lisp **| { *do do* dolist dotimes loop for until dosequence mapc mapcar mapcan mapcon map iterate dohash make-** }** *// циклы, отображения, обработка структур данных*	Common Lisp, наследуя семантику функций языка Lisp, предлагает комплект функций, выполняющих работу разнообразных схем циклов и обработчиков структур данных,
Racket: **F ::= F.**Scheme **| ⟦(λ ([атом : *Тип*] [(атом S-выр)] …))** ⟧ *// типы аргументов и значения по умолчанию* **| ⟦(*define/contract* (F атом ...)** **(-> *Тип* …) *Форма*)** ⟧ *// типизация формата функции, переменные* *// и результат вызова* Racket: ***Tun* ::= { *integer symbol string* … }**	Добавлена возможность объявлять типы параметров и давать им значения по умолчанию. Определения функций могут содержать контракт - объявления типов данных результата функции, её параметров и вычисляемых форм.
Clojure: **M ::= M.**Common Lisp **F ::= F.**Common Lisp **| { *require import slurp spit ServerSocket* }**	Clojure берёт функции и макросы от Common Lisp и вводит ряд функций, специфичных для организации процессов в сетях.

[27] Обозначения из раздела 2 таблица 4

Параметры функций обладают равноправием и независимостью.

Таблица 13.

Показ поддержки особенностей принципа «независимость и равноправие параметров» функций

⟦*Правильные формулы* ⟧	*Пояснение*
Lisp: ***Форма*** **||** ⟦**((λ (x y) (E … x … y …)) A B)** ≈ **((λ (y x) (E … x … y …)) B A)** □ **{ E A B }** ∈ **BSD.**Lisp ⟧	Параметры функции не зависят друг от друга, их можно вычислять в любом порядке, если они вычислимы или макросы.
Scheme: ***Форма*** **::=** ***Форма.***Lisp **||** ⟦**((λ (x y) (E …)) A List (B))** ≈ **((λ (x . y) (E…)) A B1 B2 … Bk)** □**{ E A B}** ∈ **BSD.**Scheme ≈ **BSD.**Lisp **!~D** ⟧	Выделен серийный параметр, он последний после « **.** ».
Common Lisp: ***Форма*** **::=** ***Форма.***Lisp **|** ⟦**(λ (x** ***&opt y &kee*** **k** ***&rest z)*** ***(E … x … y … k … z …))*** ⟧ // *виды параметров: факультатив, ключи, серии*	Различаются виды параметров, можно указать позиционные, необязательные, ключевые и серийные в соответствующем порядке.
Racket: ***Форма ::= Форма.***Scheme **|** (***Define*** **(F x …)** ***(match*** **x [P1 E1] … [Pk Ek]))** **|** (***Define*** **(F pat …) (E …))** // *шаблоны аргументов* **|** { ***match values values-ref let-values*** }	При определении функций можно вместо имен параметров использовать сопоставление аргументов с образцами.
Clojure: ***Форма ::= Форма.***Common Lisp **|** ⟦(***Defn*** **F [x …]** { ***:pre*** **[P …]** ***:post [P…] } E)*** **|** { ***assert pre post AssertionError*** } ⟧	Введён формат и специальные функции для проверки правильности определений функции независимо от контроля типов данных

Для организации потока вычислений и трансформации программ удобно, чтобы любая функция вырабатывала результат, причём ровно один. При необходимости большего числа результатов их можно объединять в один список, а отсутствие результатов можно выражать значением «пустой список» — Nil. Применяемая к конкретным аргументам функция вырабатывает определённый результат, в повторном вычислении которого нет необходимости. Любое подвыражение можно вынести из представления формы в параметр функции.

Таблица 14.

Показ поддержки особенностей принципа «единственность результата функции»

⟦*Правильные формулы* ⟧	*Пояснение*
Lisp: ***form*** **||** ⟦ **(E … (S …) …)** ≈ **((λ (x) (E … x …)) (S …))** □ **{ E S }** ∈ **BSD.**Lisp ⟧	Любое подвыражение может сделать аргументом функции или макроса.

Scheme: ***Форма* ::= *Форма.***Lisp **|| ⟦ ((λ (x) (E … x …)) (S …)) ≈ (E … (S …) …) || S ∉ D** **□ { E S } ∈ BSD.**Scheme **≈ VS.**Lisp **!~D ⟧**	Вызовы процедур, не дающих результат, не могут быть аргументами функций. В диалекте Scheme процедуры взаимодействия с внешними устройствами используются только на внешнем уровне.
Common Lisp: ***Функция* ::= *Функция.***Lisp **| ⟦F => *value* [v1 v2 … vK] ⟧** *// мультизначение*	Результат функции может быть мультизначением для организации параллельных вычислений.
Racket: ***Функция* ::= *Функция.***Scheme ***| (Define (F x ...) (values …))*** *// список результатов* ***| (Define-values (v ...) (E …))*** *// параметры-результаты* **| { *match values values-ref let-values* }**	Возможна выработка мультизначений, выбор из них конкретного и связывания мультизначения с именами.
Clojure: ***Функция* ::= *Функция.***Common Lisp **|| ⟦@атом … => {@атом | S-выр } ⟧** **| { *swap! compare-and-set! alter dosync commute ref-set reset!* }** ***Tun* ::= *Tun.***Racket **| {*atoms refs agents*}** *// Reference Types — указательные типы*	Введены специальные функции для работы с атомами как с указательными переменными, которые можно изменять, не изменяя значения, на которые они указывали.

Принципы функционального программирования не налагают ограничений на схему управления вычислениями, что допускает организацию и выполнения ленивых вычислений

Таблица 15.

Показ поддержки выполнения ленивых вычислений (Lazy evaluation)

Правило БНФ	*Пояснение*
Lisp, Pure Lisp: Lazy ::= { quote eval }	Базис языка включает блокировку (quote) и вычисление (eval) и наследует от λ-исчисления, что параметры фенкции могут вычисляться в любое время
Scheme: ***Lazy ::= Lazy.***Lisp **| *((define (delay expr)*** **(lambda ()** **(eval *expr (the-env))*** **| *(define (force promise) (promise))*)**	Консервативное расширение базиса приостановка (delay) и возобновление вычислений (delay) выполнено на базе eval, перенесённой в другой модуль.
Common Lisp: ***Lazy* ::= *Lazy.Lisp*** **| *(lazy lazy-cons force)***	Синтаксический сахар в виде макроса (lazy), ленивой пары (lazy-cons) и результата (force)

Racket: Lazy ::= Lazy.Scheme | (delay force)	Неконсервативное расширение (delay force) встроено как машинные процедуры
Clojure: ***Lazy* ::= *Lazy.***Lisp **| (*lazy-seq range map filter* …)**	Введены ленивые последовательности (lazy-seq) из произвольных выражений и стандартные функционалы их обработки (range map filter и др)

Поддержка разных схем управления вычислениями опирается на ряд встроенных структур данных, дополняющих структуры уровня синтаксиса.

Таблица 16.

Показ особенностей реализация типов и структур данных (ТСД)

Правило БНФ	*Пояснение*
Lisp: **ТСД ::= { Nil атом числа строки S-выражения вектор множество хэш }** свойства **::= { *PNAME APVAL *SUBR FSUBR EXPR FEXPR ... }***	Вектора, множества, хэш-таблицы неявно существуют в реализации, их можно моделировать с помощью списков. Свойства атома могут быть многократными и программируемыми.
Scheme: **ТСД ::= ТСД.**Lisp **!~{Nil атом} | порт | #f | идентификатор | вектор** свойства **::= {** указатель имя видимость окружение изменяемость связывание **}**	За основу реализационных структур взяты вектора. Фиксированы свойства идентификаторов
Common Lisp: **ТСД ::= ТСД.**Lisp **| |N| | поле** // *ограниченные числа, изменяемые поля* **| { *вектор множество хэш* … }** свойства: **{*name value package symbol macro*** special type status ***documentation ... }***	Добавлены числа машинно-зависимого размера и изменяемые поля. Многомерные вектора, множества, хэш-таблицы и другие структуры данных конструируются по списочной модели. Свойства однократные, программируемые.
Racket: **ТСД ::= ТСД.**Scheme **| { date/time double regexp множество** хэш-таблицы структуры классы объекты функции потоки параметры} свойства**::= { *name value plist print-name function documentation type category default-value validator location source contract* … }**	К обычным структурам данных добавлены классы, объекты, потоки, регулярные выражения и различие точных и неточных чисел.
Clojure: **ТСД ::= ТСД.**Racket **|{ & more :as all :or […] (…)** // *реструктуризация списка параметров* ***Collections* Numbers true false Strings *Keywords Symbols* Vectors Lists Maps Sets Functions Records атомы агенты** deftype указатели } свойства**::= {** author version line column file arglists macro const doc private tag **ATOM REF VAR COLLECTION и т.д. в JVM }**	Введена реструктуризация структур данных, работающая с любой последовательностью, включая список параметров, транзакционная память как в базах данных, динамические переменные, обеспечивающие разные дисциплины доступа к памяти и стратегии многопоточности (atom, ref, agent). Атом - указательная динамическая переменная.

Особенности поддержки работы с памятью требуют представления разных механизмов её функционирования, переключения дисциплины обработки и выяснения достижимости элементов памяти «◊». Здесь изменяется формат понятия «семантическая система», используется вариант с разделёнными составляющими **F** и **R** — **⟦ V, F; R ⟧**, в котором правило применения операций **R** не рассматривается как операция из **F**, следовательно, его нельзя перепрограммировать, оно защищено от изменений**.**

Таблица 17.

Показ поддержки принципа «гибкость границ блоков памяти»

((*Дисциплина* **))**	*Пояснение*
Lisp: memory **::= (({**Heap Free**}** **|** Heap **((**Free **=))** **| ±** { дисциплина1 **::=** **⟦**Heap**; (→↓ ↑*); **очередь**⟧** **⟦**Free**; ←; **действие **⟧** **| (cons** X Y**) += (**Heap **(*set* !**Free (X **.** Y))) # Free **= 0 =>** GC *// auto* *// переключается дисциплина* дисциплина2 **::=** **⟦**Heap**; (↑* ←); **очередь **⟧** **⟦**Free**; →; **действие **⟧ }))** **| GC** GC::= **((((** x ε Heap (~◊ x => (*delete* Heap x) (cons x Free)) **))**	Heap ⋂ Free не пересекаются. Выполнение функции **cons** сопровождается изъятием элемента из Free, записью в него своего результата и размещением элемента в Heap. При опустошении Free автоматически запускается GC, меняющий дисциплину {Heap Free} на перенос недостижимых из программы ячеек блока Heap в блок Free. И в том и в другом случае граница между блоками изменяется. →↓ ↑← — операции над памятью: инициирование, запись, чтение, удаление и * —многократное повторение операции. Чтение ↑ вырабатывает указатель на пару с адресом. Формы (F … (**GC** …) …) могут содержать вызов мосорцика. **GC** можно вызывать из программы. .
Scheme: memory **::=** memory.Lisp !~**GC** | GC	Диалект Scheme использует только автоматическое, реализационно зависимое освобождение памяти GC.
Common Lisp: memory **::=** memory.Lisp **| { *date room …* }**	Можно указывать интервалы запуска **GC**, выяснять время, даты и этапы работы программы.
Racket: memory **::=** memory.Common Lisp **|** *// Оптимизированная сборка* *// мусора с управлением* *// производительностью кода.*	Включен поколенческий **GC**, минимизация пауз и функция, дающая сведения о потреблении памяти. Поддержаны ленивые последовательности для потенциально бесконечных данных. Имеются средства рефакторинга и измерения производительности.
Clojure: memory ::= memory.Common Lisp **|** *// JVM сборщики мусора* *// (G1, ZGC, Shenandoah …)*	Используются постоянные структуры данных, переходы (persistent data structures, transients), System/gc. VisualVM, JProfiler) для выявления утечек памяти и неэффективного использования памяти.

Неизменяемость значений на уровне ЯП не исключает необходимость изменения данных на уровне ЯиСП. Иначе нет возможности поддерживать принцип «гибкость границ» — необходимо часть памяти перемещать из одного блока в другой и менять дисциплину функционирования памяти. Для обеспечения таких особенностей проводится разница между

значениями и указателями. Неизменяемость поддерживается для хранимых значений, а указатели на значения, в порядке исключения, можно изменять.

Таблица 18.
Показ особенностей поддержки принципа «неизменяемость данных», более точно — «неизменяемость хранимых в памяти значений»

((*Правило* **))**	*Пояснение*
Lisp: Heap **::=** **((↕ (((F x) □ x ∈** Heap **(x)) (F x) =)))** ↕{Rec Clo ***Prog***} □ Heap Δ Rec = **{**FnRec **… };** **±** (→↓↑*)* // *поколения* Δ Clo = **{**FnVal **… }; ±** →↓↑* ↓↑* ↕ FnVar **∈** call // *функ. значение* Δ ***Prog*** = **{**Var **… }; ±** ↓ (↑* ↓*)* ↕ Var **∈** ***Prog***.2 // *переменные* Δ **{** ***rplaca rplacd*** **}: ±** ↑ ↓ **))**	Память для результатов функции отличается от памяти аргументов. Выделяются области памяти для организации рекурсии, замыканий и рабочих переменных функции ***Prog***, допускающие изменения. Исключения: При реализации рекурсии (Rec) и ленивых вычислений (Clo) для эффективности заранее выделяются изменяемые поля (rplaca и rplacd). Внутри ***Prog*** функции ***set*** и ***setq*** меняют состояния рабочих переменных без внешнего побочного эффекта. Изменение рабочих переменных действует в локальной области видимости вызова функции.
Scheme: Heap **::=** Heap**.**Lisp **!~ { *Prog rplaca rplacd*}** **|{** ***set! set-car! set-cdr! vector-set! string-set!*** **}** **((** ↕{Rec Clo**}** **((** Heap Δ **{*set! set-car! set-cdr! vector-set! String-set!* }: {±** ↑ ↓**))** **}))**	Мутирующие процедуры (set! set-car! set-cdr! vector-set! string-set!) изменяют переменные, элементы вектора и строки. Можно заменять ранее введённые определения функций (define) без возможности вернуться к прежним.
Common Lisp: Heap **::=** Heap**.**Lisp **((** ↕ поле **: (**↓ **(**↑* ↓**)*)** // *очередь* Δ **{ *setf* }: ±** ↑ ↓ **))**	Введено неявное понятие «изменяемое поле» для работы с изменяемыми свойствами символа, деструктивными аналогами ряда функций, setf для переменных и полей, массивов строк, defparameter и defvar, изменяемых слотов из CLOS.
Racket: Heap **::=** Heap**.**Scheme **((** ↕ {поля **вектора** хеши структуры **}** **: (**↓ **(**↑* ↓**)*)** // *очередь* Δ **{ *mutable-struct*}:±** ↑ ↓ **))**	*Введены мутирующие структуры данных и функции-мутаторыt.* *Макрос изменяет код при компиляции и вычислении.* . □ □
Clojure: Heap **::=** Heap**.**Common Lisp **|** метаданные . **((** ↕ {@Refs **@**dosync **@Atoms** Java-объекты **}: ±** (↓ ↑* ↓*) // *очередь* Δ **{ *def* }: ±** ↑* ↓* // *повторные определения* **))**	Введена транзакционная память и реструктуризация последовательностей, изменяющие ссылки на неизменяемые значения, транзакции, указательные переменные. Функция def может изменять определения, а set! - значения измененных,

Таблица 19.
Показ механизмов управление вычислениями, ветвления

Правило	*Пояснение*
Lisp: ветви **::= форма** **| ⟦↕{ (cond (P[28] s-выр) …)** **(*and* P …)** **(*or* P …) } ⟧** Lisp: P **::= ⟦форма ↧ F : x => {Nil | s-выр}⟧**	Конструктор **cond** строит произвольное число ветвей, объединяет возможности условного выражения и переключателя. Логические функции **and** и **or** фактически работают как ветвления — до первого истинного или ложного выражения (логика Маккарти)
Scheme: ветви **::=** ветви**.**Lisp **| { (if P s-выр s-выр)** **(*case* форма** ((значение **…) s-выр) …** **)** **(*unless* s-выр …)** **(*when* P s-выр …) }** P **::= форма | ⟦↕ F : x => {#f | s-выр} ⟧**	Механизм ветвлений дополняется условным выражением if в качестве основной формы, переключателем по значению case и циклами when и unless. #f - логическое значение «ложь» вместо Nil.
Common Lisp: ветви **::=** ветви**.**Scheme **| { (*ecase* форма ((**тип **…) s-выр) …)** *// нет иначе* **(*typecase* s-выр ((**тип **…) s-выр) … (T s-выр)** **)** **P ::= P.**Lisp тип **::= { *typep boundp fboundp keywordp compiled-function-p oddp plusp complexp subtypep listp consp atom null arrayp hash-table-p packagep stream-p pathnamep structure-object-p … }**	Формат ветвление из Scheme, но логические значения из Lisp. Имеется переключатель, сообщающий об отсутствии истинного предиката и выбор по типу значения. Имеются предикаты на все встроенные и реализационно различимые типы и структурв значений.
Racket: ветви **::=** ветви**.**Scheme **| (*match* s-выр [s-выр s-выр] …)** тип **::= { *exact? inexact? zero? positive? negative? even? odd? pair? list? null? vector? hash? set? boolean? procedure? port? input-port? output-port? syntax? eof-object? void? … }**	Добавлено сопоставление с образцом. Имеются предикаты на все встроенные и реализационно различимые типы и структурв значений.
Clojure: ветви **::=** ветви**.**Scheme **| { (*if-let* [атом s-выр] s-выр …)** **(*when* предикат s-выр …)** **(*when-let* [атом s-выр] s-выр …** **)** **(*if-not* предикат s-выр s-выр)** **(*when-not* предикат s-выр …)** **(*condp* предикат s-выр …)** тип **::= { *type? instance? number? integer? float?ratio? complex? keyword?symbol? string?vector?list? map? set? character? seq? fn? coll? record? bigint? bigdec? char? Ifn? … }**	Добавлены виды ветвлений, по разному формирующие результаты и различно структурирующие параметры.

Интерактивный стиль взаимодействия с ЯиСП по разному определён не только в разных диалектах, но и при взаимодействии с разными периферийными устройствами. Здесь подразумевается взаимодействие со стандартными вводом/выводом.

[28] См. Таблица 12

Таблица 20.
Показ различий REPL-цикла: интерпретации и компиляция

Правило	*Пояснение*
Lisp: REPL-цикл **::=** **(**LOOP **(*print* (eval (*print* (*read*))))** **)** // *(READ-EVAL-PRINT LOOP)*	Печать результата чтения, его вычисление и печать полученного значения. При выполнении EVAL может быть вызвана функция COMPILE, выполняющая частичную компиляцию отдельных функций.
Scheme: REPL-цикл **::=** REPL-цикл.Lisp (loop (set! x **(read)) (displayln x)** (set! x (compile x)) (set! y (run **x)) (displayln y)**)	Вместо функции EVAL выполняет встроенную функцию COMPILE, теряя исходный код.
Common Lisp: REPL-цикл **::=** REPL-цикл.Lisp **| (*loop* (*print* (eval** **(*compile* (*print* (*read*))))))** // *возможна компиляция полной* // *программы.*	REPL-интерфейс использует быструю компиляцию в машинный код для немедленного вычисления и вывода результата, поддерживает интерактивную работу с кодом. Возможен дизассемблированный листинг кода.
Racket: REPL-цикл **::=** REPL-цикл.Scheme | // *REPL-интерфейс использует* // *JIT-компиляцию в байт-код* // *или JVM.*	Выделен ряд диалектов, соответствующих уровням способностей студентов (Minimal Racket, Racket, Lazy Racket, Typed Racket и др.) с разными схемами организации вычислений.
Clojure: REPL-цикл **::=** REPL-цикл.Common Lisp **| {** nREPL CIDER Cursive **}** // *поддержка отладки,* // *горячей загрузки кода и* // *интеграции с инструментами* // *разработки и верификации.*	Это "пульт управления" работающей системы, работает поверх JVM, JavaScript (ClojureScript) или .NET (ClojureCLR). Делает исполняемые файлы. Добавлены варианты, позволяющие исследовать состояние программы (@state), синхронизовывать выполнение потоков: откладывание, ожидание, обещание, передача, готовность, блокировка (delay, future, promise, deliver, is-done?, deref), представляет эффективные формы циклов, не использующих стек, включая поддержку тестирования и верификации.

Приложение 6

Показ стабильности ядра ЯиСП для языка Lisp и его диалектов

Обычно при реализации ЯиСП происходит декомпозиция на слои, такие как базовые средства, макросы для расширения ЯП, инструменты проверки границ вычислимости и средства взаимодействия с внешним миром, с периферийными устройствами. Отчасти такую декомпозицию можно выразить подобно синтаксису с послойным перечислением семантических функций, поддерживающих вычисления, хранение данных в памяти, управление вычислениями и обработку структур данных.

Lisp: базис **::= {NIL eq cons car cdr atom}**
макро **::= {lambda *define* quote cond ...}**
границы **::= {eval error *trace untrace compile* ...}**
периферия **::= {print read *REPL prog* set setq rplaca rplacd *put get compile* gensym GC ...}**

Для языка Lisp характерен лаконичный базис, обеспечивающий работу с S-выражениями, комплект макросов, достаточный для расширения ЯП, проверка границ вычислимости опирается на универсальную функцию eval, способную для любого S-выражения выработать его значение или сообщить с помощью функции error причину содержащейся в нём ошибки. При отладке можно использовать функции trace и untrace, управляющие выводом данных о вычисляемых функциях и их аргументах. На периферии внимания включается богатая коллекция средств для связи с внешним миром и программирования работ, отклоняющихся от принципов ФП.

Scheme: базис **::=** базис.Lisp **!~NIL !~eq | eq? | #f | #t** *// из-за векторов*
макро **::=** макро.Lisp **!~cond | {if case delay force let let* letrec … define-syntax let-syntax letrec-syntax syntax-rules}**
границы **::= {compile exit raise eqv? equal? ...}** *// из-за компиляции*
модули **::=** периферия.Lisp **| {*list eval* set! set-car! set-cdr! display *do map for-each exists for-all* …}** *// библиотеки*

Диалект Scheme, наследуя базис языка Lisp, исключает из него имя предиката «eq» и заменяет его на «eq?», исключает атом NIL и вместо него вводит булево значение #f , наследуя макро — средства расширения языка, убирает макрос «cond», вместо «cond» вводит «if» и «case», «compile» переводит в свои средства контроля границ, в которые включает ряд функций, удобных для применения компилятором. Особо выделены синтаксические макросы, работающие на этапе компиляции. Наследуя периферию, сводит её к комплекту модулей и процедур, не вырабатывающих результат.

Common Lisp: базис **::=** базис.Lisp
макро **::=** макро.Lisp **!~*define***
| *defun* | defmacro | {if setf *delete nconc nsubst …*
***mapcar mapcan mapcon make- multiple- funcall* function**
***loop declare gethash defmacro* … make-* multiple-* typep ...}**
границы **::=** границы.Lisp **| {*try catch throw step*}**
периферия **::=** периферия.Lisp
| { *apropos dribble describe* open close **… *CLOS*}** *// пакеты*

Диалект Common Lisp наследует все архитектурные составляющие языка Lisp, включая «макро», переименовывает «define» в «defun», к средствам контроля границ и взаимодействия с периферией добавляет ряд практичных функций и пакет «CLOS» для поддержки ООП.

Racket: базис **::=** базис.Scheme **| {**delay force**}** *// встроены*
макро **::=** макро.Scheme **| {mcons mcar ...set-mcar! .. vector … box unbox set-box!}**
границы **::=** границы.Scheme **| {error break check-expect try/except try-catch trace untrace printf displaylnDrRacket Racket Debugger raco check-syntax}**
модули **::=** модули.Scheme **| {read display newline printf Opn-input-file … port-count-bytes ... file-exists? close-input-port … delete-file}**
| диалект

Диалект Racket наследует все архитектурные составляющие языка Scheme, добавляя к ним ряд функций, характерных для Common Lisp и систему диалектов разного назначения.

Clojure: базис **::=** базис.Lisp **!~{eq car cdr} | {eq? first rest} | {true false}**
макро **::=** макро.Clisp **!~{lambda define} | {fn defn def … if # let case reset! swap! atom ref agent promise future thread-safe deliver apply map filter reduce loop/recur quote do when}**
границы **::=** границы.Lisp
| {try catch throw finally is-done? Number? ... vector? list? Set? map? Coll?}
периферия **::=** периферия.Clisp **| {println defprotocol defonce deftype defrecord take drop take-while drop-while}**

Диалект Clojure наследует базис и границы от языка Lisp, а макро и периферию от диалекта Common Lisp, переименовывая (eq car cdr lambda define) в (eq? first rest fn defn) соответственно. Ко всем составляющим добавляется ряд функций, нацеленных на повышение продуктивности программирования для современных ИТ на базе JVM.

Вывод: Полученные описания архитектуры ЯиСП диалектов показывают, что наибольшее отклонение от языка Lisp произошло при создании Scheme, в более поздних диалектах объём наследования архитектурных решений языка Lisp возрос до практически полного восстановления. Выполнено развитие понятия «атом» и введены в реализацию ЯиСП механизмы пакетов, областей видимости, создания диалектов, реструктуризации, транзакционная память и метаданные, удобные для разработки современных приложений ИТ.

Приложение 7

Лисповское диалектное абстрагирование семантики вычислений от семантики изменения состояний памяти

Каждый из диалектов произвёл определённое уточнение особенностей ФП без отказа от его принципов, впервые поддержанных в реализации языка Lisp.

Таблица 21.

Показ особенностей современной парадигмы ФП, её связь с понятиями ЯП

Принцип	*Формула*	*Примечание*
Гомоиконный синтаксис	Lisp**: ТСД.**Lisp **::= S-выражение** *// достаточно для представления всех конструкция ЯП*	
Универсальность.	ФП**: V::= ⟦∃ { ТСД.**ЯП **}** □ **{ тип.ЯП metaD.ЯП }}⟧**	Синтаксис подобен Clojure для всех структур и типов данных, включая реализационные.
Само-определение функций (рекурсия)	ФП**:** F **::= ⟦V … => V** **↕ *(Define* F (λ (x ...) (… (F ...) … F…)) ⟧** ФП**:** M **::= ⟦V … => (F : V => V)** **↕ *(Defmacro* M (x ...) (… (M ...) … M …)) || {M} ⊂ F ⟧** ФП**:** P **::= ⟦V … => Nil | #f | V** **↕ {P} ⊂ F ⟧**	Синтаксис Common Lisp и Clojure, синтаксический сахар ***Define*** и ***Defmacro*** может быть переименован с добавлением ряда аналогов **{ *dosequence map mapc mapcar mapcan mapcon iterate dohash*** и др. **}**
Частичные функции, **ветвления** и циклы	ФП**:** ветвление **::=** **⟦форма.**ЯП **⊂** **{ (cond (P форма) …)** **(*and* P …) (*or* P …) } ⟧**	Имена **cond, *and, or*** могут быть изменены. Возможны циклы **{** ***do do* dolist dotimes loop for until while*** и др. **}**
Трансформационная семантика	Существуют разные определения функций и представления фыорм, результаты которых совпадают на одинаковых комплектах аргументов в одинаковых контекстах.	
Независимость и равноправие параметров функции	ФП**: форма ::= форма.**ЯП **↨ ((λ (x y) (E … x … y…)) A B)** ≈ **((λ (y x) (E … x … y …)) B A) ⟧** **↨ ⟦** (E … (S …) …) ≈ **((λ (x) (E … x …)) (S …)) ⟧**	Эквивалентность: Перестановочность параметров и аргументов. Вынесение подвыражений в аргумент.
Единственность результата функции	ФП**:** F **::= F.**ЯП **| ⟦V … => V [v1 v2 … v**K **] ⟧** *// мультизначения* ФП**: форма ::= форма.**ЯП **| ({ *:pre* [P …] *:post* [P…] }** **форма)**	Семантика Common Lisp с проверкой правильности от Clojure

Ленивые вычисления	ФП**:** Lazy::= F**.**ЯП ↕ ⟦ F□ {quote eval delay force**}** ⟧	Нужны блокировка, вычисление и/или приостановка, возобновление
REPL	ФП**:** REPL-цикл**.**ЯП **| (*loop* (print (eval (*compile*** **(print (read**))))))	Возможна компиляция полной программы без потери кода.
Прагматичная динамика	Поддержка принципов ФП использует исключения, обработка которых необходима для обеспечения продуктивности программирования и производительности программ.	
Гибкость распределения памяти	ФП**:** memory **::=** memory**.**Common Lisp ФП**: GC::= GC.**Racket **| GC.**JVM	Переключение дисциплины функционирования памяти и варианты мусорщика
Неизменяемость хранимых значений	ФП**:** Heap **::=** Heap**.**Clojure	От Clojure добавлены метаданные, транзакции, указательные переменные и асинхронность.

Приложение 8

Перечень таблиц, показывающий результаты сравнения диалектов языка Lisp